The great book

for ESP32forth

version 1.3 - 22 October 2023

Author

- Marc PETREMANN petremann@arduino-forth.com

Collaborators

- Vaclav POSELT

Contents

Introduction

Since 2019, I manage several websites dedicated to FORTH language development for ARDUINO and ESP32 boards, as well as the eForth web version:

- ARDUINO : https://arduino-forth.com/
- ESP32 : https://esp32.arduino-forth.com/
- eForth web : https://eforth.arduino-forth.com/

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

It will happen sooner or later – and as late as possible – that I will no longer be able to ensure the sustainability of these sites. The consequence will be that the information disseminated by these sites disappears.

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

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

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

Translation help

Google Translate allows you to translate texts easily, but with errors. So I'm asking for help to correct the translations.

In practice, I provide the chapters already translated in the LibreOffice format. If you want to help with these translations, your role will simply be to correct and return these translations.

Correcting a chapter takes little time, from one to a few hours.

To contact me : petremann@arduino-forth.com

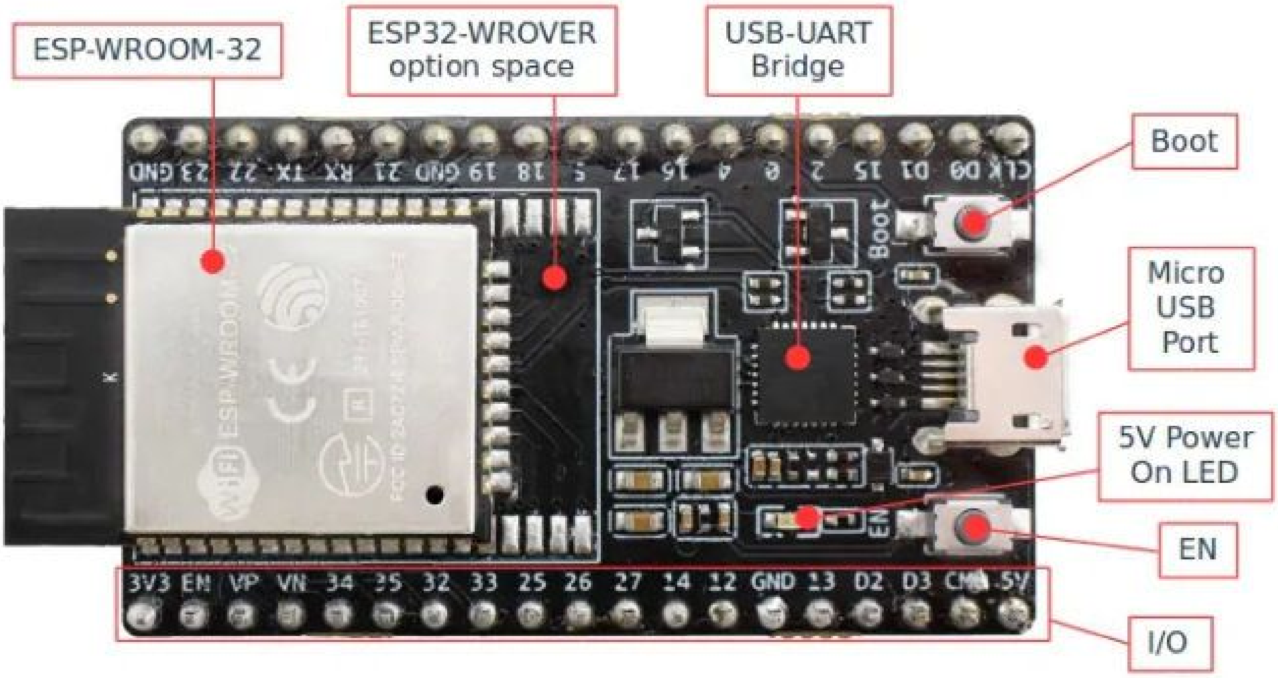
Discovery of the ESP32 card

Presentation

The ESP32 board is not an ARDUINO board. However, development tools leverage certain elements of the ARDUINO eco-system, such as the ARDUINO IDE.

The strong points

In terms of the number of ports available, the ESP32 card is located between an ARDUINO



NANO and ARDUINO UNO. The basic model has 38 connectors:

ESP32 devices include :

- 18 analog-to-digital converter (ADC) channels
- 3 SPI interfaces
- 3 UART interfaces
- 2 I2C interfaces
- 16 PWM output channels
- 2 digital-to-analog converters (DAC)
- 2 I2S interfaces

- 10 capacitive sensing GPIOs

The ADC (analog-to-digital converter) and DAC (digital-to-analog converter) functionality are assigned to specific static pins. However, you can decide which pins are UART, I2C, SPI, PWM, etc. You just need to assign them in the code. This is possible thanks to the multiplexing function of the ESP32 chip.

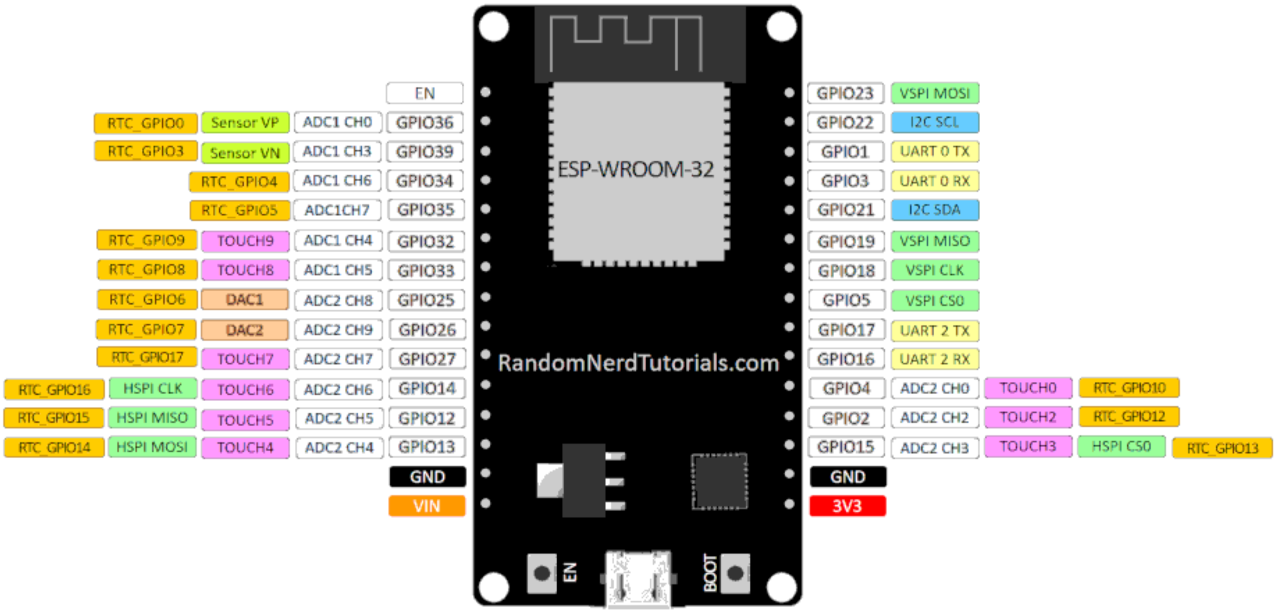
Most connectors have multiple uses.

But what sets the ESP32 board apart is that it is equipped as standard with WiFi and Bluetooth support, something that ARDUINO boards only offer in the form of extensions.

GPIO inputs/outputs on ESP32

Here, in photo, the ESP32 card from which we will explain the role of the different GPIO inputs/outputs :

The position and number of GPIO I/Os may change depending on the card brand. If this is the case, only the indications appearing on the physical map are authentic. Pictured, bottom row, left to right: CLK, SD0, SD1, G15, G2, G0, G4, G16.....G22, G23, GND.



In this diagram, we see that the bottom row begins with 3V3 while in the photo, this I/O is at the end of the top row. It is therefore very important not to rely on the diagram and instead to double check the correct connection of the peripherals and components on the physical ESP32 card.

Development boards based on an ESP32 generally have 33 pins apart from those for the power supply. Some GPIO pins have somewhat particular functions :

GPIO	Possibles usage
6	SCK/CLK
7	SCK/CLK
8	SDO/SD0
9	SDI/SD1
10	SHD/SD2
11	CSC/CMD

If your ESP32 card has I/O GPIO6, GPIO7, GPIO8, GPIO9, GPIO10, GPIO11, you should definitely not use them because they are connected to the flash memory of the ESP32. If you use them the ESP32 will not work.

GPIO1(TX0) and GPIO3(RX0) I/O are used to communicate with the computer in UART via USB port. If you use them, you will no longer be able to communicate with the card.

GPIO36(VP), GPIO39(VN), GPIO34, GPIO35 I/O can be used as input only. They also do not have built-in internal pullup and pulldown resistors.

The EN terminal allows you to control the status of the ESP32 via an external wire. It is connected to the EN button on the card. When the ESP32 is turned on, it is at 3.3V. If we connect this pin to ground, the ESP32 is turned off. You can use it when the ESP32 is in a box and you want to be able to turn it on/off with a switch.

ESP32 Peripherals

To interact with modules, sensors or electronic circuits, the ESP32, like any micro-controller, has a multitude of peripherals. There are more of them than on a classic Arduino board.

ESP32 has the following peripherals :

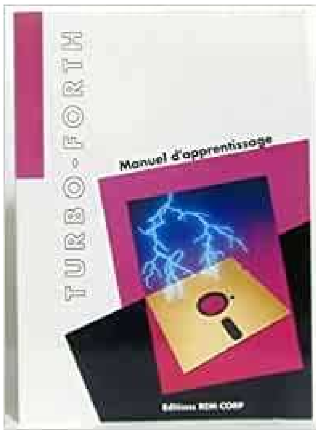
- 3 UART interface
- 2 I2C interfaces
- 3 SPI interfaces
- 16 PWM outputs
- 10 capacitive sensors
- 18 analog inputs (ADC)
- 2 DAC outputs

Some peripherals are already used by ESP32 during its basic operation. There are therefore fewer possible interfaces for each device.

Why program in FORTH language on ESP32?

Preamble

I have been programming in FORTH since 1983. I stopped programming in FORTH in 1996. But I have never stopped monitoring the evolution of this language. I resumed programming in 2019 on ARDUINO with FlashForth then ESP32forth.



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

- Introduction au ZX-FORTH (ed Eyrolles - 1984 - ASIN:B0014IGOXO)
- Tours de FORTH (ed Eyrolles - 1985 - ISBN-13: 978-2212082258)
- FORTH pour CP/M et MSDOS (ed Loisitech - 1986)
- TURBO-Forth, manuel d'apprentissage (ed Rem CORP - 1990)
- TURBO-Forth, guide de référence (ed Rem CORP - 1991)

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

The two software designers of this company programmed in C language: TURBO-C from Borland to be precise. And their code couldn't be compact and fast enough to fit into the 64 kilobytes of RAM memory. It was 1992 and flash memory type expansions did not exist. In these 64 KB of RAM, we had to fit MS-DOS 3.0 and the application!

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

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

Boundaries between language and application

All programming languages are shared like this :

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

The FORTH language is an exception. It integrates :

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

What is a FORTH word?

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

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

With the FORTH language, the essential principle is that we do not create an application. In FORTH, we extend the dictionary! Each new word you define will be as much a part of the FORTH dictionary as all the words pre-defined when FORTH starts. Example:

```
: typeToLoRa ( -- )
    0 echo !    \ disable display echo from terminal
    ['] serial2-type is type
  ;
: typeToTerm ( -- )
    ['] default-type is type
    -1 echo !   \ enable display echo from terminal
  ;
```

We create two new words: **`typeToLoRa`** and **`typeToTerm`** which will complete the dictionary of pre-defined words.

A word is a function?

Yes and no. In fact, a word can be a constant, a variable, a function... Here, in our example, the following sequence :

```
: typeToLoRa ...code... ;
```

would have its equivalent in C langage :

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

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

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

Yes, but then why FORTH rather than C?

I was expecting this question.

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

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

The FORTH interpreter is immediately accessible on the ESP32 card via a terminal type program and a USB link between the ESP32 card and the PC.

The compilation of programs written in FORTH language is carried out in the ESP32 card and not on the PC. There is no upload. Example:

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

This definition is transmitted by copy/paste into the terminal. The FORTH interpreter/compiler will parse the stream and compile the new word **`>gray`**.

In the definition of **`>gray`**, we see the sequence **`dup 2/ xor`**. To test this sequence, simply type it in the terminal. To execute **`>gray`**, simply type this word in the terminal, preceded by the number to transform.

FORTH language compared to C language

This is my least favorite part. I don't like to compare the FORTH language to the C language. But as almost all developers use the C language, I'm going to try the exercise.

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

```
if(j > 13){                 // If all bits are received
    rc5_ok = 1;             // Decoding process is OK
    detachInterrupt(0);     // Disable external interrupt (INT0)
    return;
}
```

Test with if in FORTH language (code snippet) :

```
var-j @ 13 >        \ If all bits are received
    if
        1 rc5_ok !  \ Decoding process is OK
```

```
        di           \ Disable external interrupt (INT0)
        exit
    then
```

Here is the initialization of registers in C langage :

```
void setup() {
  // Timer1 module configuration
  TCCR1A = 0;
  TCCR1B = 0;          // Disable Timer1 module
  TCNT1  = 0;          // Set Timer1 preload value to 0 (reset)
  TIMSK1 = 1;          // enable Timer1 overflow interrupt
}
```

The same definition in FORTH langage :

```
: setup
  \ Timer1 module configuration
  0 TCCR1A !
  0 TCCR1B !     \ Disable Timer1 module
  0 TCNT1  !     \ Set Timer1 preload value to 0 (reset)
  1 TIMSK1 !     \ enable Timer1 overflow interrupt
;
```

What FORTH allows you to do compared to the C language

We understand that FORTH immediately gives access to all the words in the dictionary, but not only that. Via the interpreter, we also access the entire memory of the ESP32 card. Connect to the ESP32 board that has ESP32forth installed, then simply type :

```
hex here 100 dump
```

You should find this on the terminal screen :

```
3FFEE964                        DF DF 29 27 6F 59 2B 42 FA CF 9B 84
3FFEE970            39 4E 35 F7 78 FB D2 2C A0 AD 5A AF 7C 14 E3 52
3FFEE980            77 0C 67 CE 53 DE E9 9F 9A 49 AB F7 BC 64 AE E6
3FFEE990            3A DF 1C BB FE B7 C2 73 18 A6 A5 3F A4 68 B5 69
3FFEE9A0            F9 54 68 D9 4D 7C 96 4D 66 9A 02 BF 33 46 46 45
3FFEE9B0            45 39 33 33 2F 0D 08 18 BF 95 AF 87 AC D0 C7 5D
3FFEE9C0            F2 99 B6 43 DF 19 C9 74 10 BD 8C AE 5A 7F 13 F1
3FFEE9D0            9E 00 3D 6F 7F 74 2A 2B 52 2D F4 01 2D 7D B5 1C
3FFEE9E0            4A 88 88 B5 2D BE B1 38 57 79 B2 66 11 2D A1 76
3FFEE9F0            F6 68 1F 71 37 9E C1 82 43 A6 A4 9A 57 5D AC 9A
3FFEEA00            4C AD 03 F1 F8 AF 2E 1A B4 67 9C 71 25 98 E1 A0
3FFEEA10            E6 29 EE 2D EF 6F C7 06 10 E0 33 4A E1 57 58 60
3FFEEA20            08 74 C6 70 BD 70 FE 01 5D 9D 00 9E F7 B7 E0 CA
3FFEEA30            72 6E 49 16 0E 7C 3F 23 11 8D 66 55 EC F6 18 01
3FFEEA40            20 E7 48 63 D1 FB 56 77 3E 9A 53 7D B6 A7 A5 AB
```

```
3FFEEA50            EA 65 F8 21 3D BA 54 10 06 16 E6 9E 23 CA 87 25
3FFEEA60            E7 D7 C4 45
```

This corresponds to the contents of flash memory.

And the C language couldn't do that?

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

But why a stack rather than variables?

The stack is a mechanism implemented on almost all microcontrollers and microprocessors. Even the C language leverages a stack, but you don't have access to it.

Only the FORTH language gives full access to the data stack. For example, to make an addition, we stack two values, we execute the addition, we display the result: `2 5 + .` displays 7.

It's a little destabilizing, but when you understand the mechanism of the data stack, you greatly appreciate its formidable efficiency.

The data stack allows data to be passed between FORTH words much more quickly than by processing variables as in C language or any other language using variables.

Are you convinced?

Personally, I doubt that this single chapter will irremediably convert you to programming in the FORTH language. When trying to master ESP32 cards, you have two options :

- program in C language and use the numerous libraries available. But you will remain locked into the capabilities of these libraries. Adapting codes to C language requires real knowledge of programming in C language and mastering the architecture of ESP32 cards. Developing complex programs will always be a problem.
- try the FORTH adventure and explore a new and exciting world. Of course, it won't be easy. You will need to understand the architecture of ESP32 cards, the registers, the register flags in depth. In return, you will have access to programming perfectly suited to your projects.

Are there any professional applications written in FORTH?

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

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

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

The choice of the FORTH language for professional applications turns out to be interesting if we consider each word as a black box. Each word must be simple, therefore have a fairly short definition and depend on few parameters.

During the debugging phase, it becomes easy to test all the possible values processed by this word. Once made perfectly reliable, this word becomes a black box, that is to say a function in which we have absolute confidence in its proper functioning. From word to word, it is easier to make a complex program reliable in FORTH than in any other programming language.

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

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

```
\ Turn a port pin on, dont change the others.
: __/ ( pinmask portadr -- )
    mset
  ;
\ Turn a port pin off, dont change the others.
: \__ ( pinmask portadr -- )
    mclr
  ;
```

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

```
_O_  __/     \ turn LED on
```

Yes! The sequence `_o_ __/` is in FORTH language!

With ESP32forth, here are all the characters at your disposal that can compose a FORTH word :

```
~}|{zyxwvutsrqponmlkjihgfedcba`_
^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?
>=<;:9876543210/.-,+*)('&%$#"!
```

Good programming.

A real 32-bit FORTH with ESP32Forth

ESP32Forth is a real 32-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 ESP32Forth starts, the FORTH interpreter is available. If you enter any number, it will be dropped onto the stack as a 32-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 32-bit integer values are added together and the result is dropped onto the stack. To display this result, we will use the word **.**:

```
. \ display 80
```

In FORTH language, we can concentrate all these operations in a single line :

```
35 45 + .  \ display 80
```

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

With ESP32Forth, an ASCII character will be designated by a 32-bit integer, but whose value will be bounded [32..255]. Example :

```
67 emit  \ display C
```

Values in memory

ESP32Forth allows you to define constants and variables. Their content will always be in 32-bit format. But there are situations where that doesn't necessarily suit us. Let's take a simple example, defining a Morse code alphabet. We only need a few bytes :

- one to define number of marks in Morse code character
- one or more bytes for Morse code marks

```
create mA ( -- addr )
    2 c,
    char . c,   char - c,
```

```
create mB ( -- addr )
    4 c,
    char - c,   char . c,   char . c,   char . c,

create mC ( -- addr )
    4 c,
    char - c,   char . c,   char - c,   char . c,
```

Here we define only 3 words, **mA**, **mB** and **mC**. In each word, several bytes are stored. The question is: how will we retrieve the information in these words ?

The execution of one of these words deposits a 32-bit value, a value which corresponds to the memory address where we stored our Morse code information. It is the word **c@** that we will use to extract the Morse code from each letter :

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

The first byte placed on the stack will be used to manage a loop to display the code of a character in Morse code :

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

There are plenty of certainly more elegant examples. Here we show a way to manipulate 8-bit values, our bytes, while operating these bytes on a 32-bit stack.

Word processing depending on data size or type

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

```
$bread = "Baked bread";
$price = 2.30;
echo $bread . " : " . $price;
// display   Baked bread: 2.30
```

For all programmers, this way of doing things is THE STANDARD! So how would FORTH do this example in PHP?

```
: bread s" Baked bread" ;
: price s" 2.30" ;
bread type   s" : " type    price type
\ display   Baked bread: 2.30
```

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

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

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

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

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

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

Conclusion

FORTH has no data typing. All data passes through a data stack. Each position in the stack is ALWAYS a 32-bit integer!

That's all there is to know.

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

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

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

```
: morse: ( comp: c -- | exec -- )
    create
        c,
```

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

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

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

Dictionary / Stack / Variables / Constants

Expand Dictionary

Forth belongs to the class of woven interpretive languages. This means that it can interpret commands typed on the console, as well as compile new subroutines and programs.

The Forth compiler is part of the language and special words are used to create new dictionary entries (i.e. words). The most important are `:` (start a new definition) and `;` (finishes the definition). Let's try this by typing :

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

What happened? The action of `:` is to create a new dictionary entry named `*+` and switch from interpretation mode to compilation mode. In compile mode, the interpreter searches for words and, rather than executing them, installs pointers to their code. If the text is a number, instead of pushing it onto the stack, ESP32forth constructs the number in the dictionary space allocated for the new word, following special code that puts the stored number on the stack each time the word is executed. The execution action of `*+` is therefore to sequentially execute the previously defined words `*` and `+`.

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

Now let's try this new word :

```
decimal 5 6 7 *+ . \ display 47 ok<#,ram>
```

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

Dictionary management

The word **`forget`** followed by the word to delete will remove all dictionary entries you have made since that word :

```
: test1 ;
: test2 ;
: test3 ;
forget test2  \ delete test2 and test3 in dictionnary
```

Stacks and reverse Polish notation

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

In FORTH, The pile is analogous to a pile of cards with numbers written on them. Numbers are always added to the top of the stack and removed from the top of the stack. ESP32forth integrates two stacks: the parameter stack and the feedback stack, each consisting of a number of cells that can hold 16-bit numbers.

The FORTH input line :

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

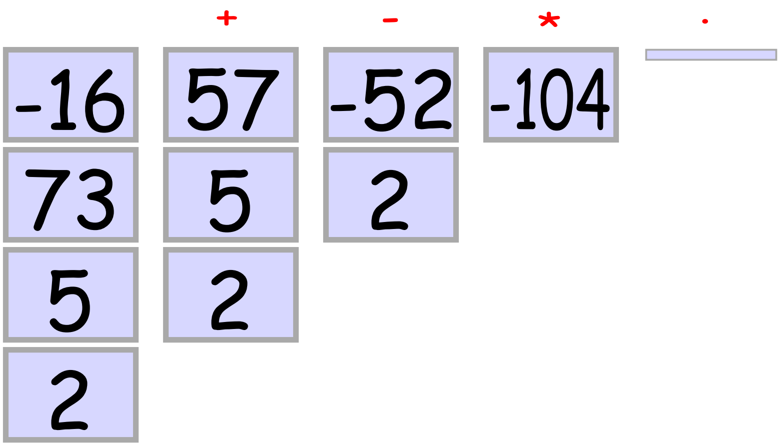
leaves the parameter stack as it is

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

We will typically use zero-based relative numbering in Forth data structures such as stacks, arrays, and tables. Note that when a sequence of numbers is entered like this, the rightmost number becomes TOS and the leftmost number is at the bottom of the stack.

Let's continue with this:

```
+ - * .
```



The operations would produce successive stack operations :

After the two lines, the console displays :

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

Note that ESP32forth conveniently displays the stack elements when interpreting each line and that the value of **-16** is displayed as a 32-bit unsigned integer. Furthermore, the word

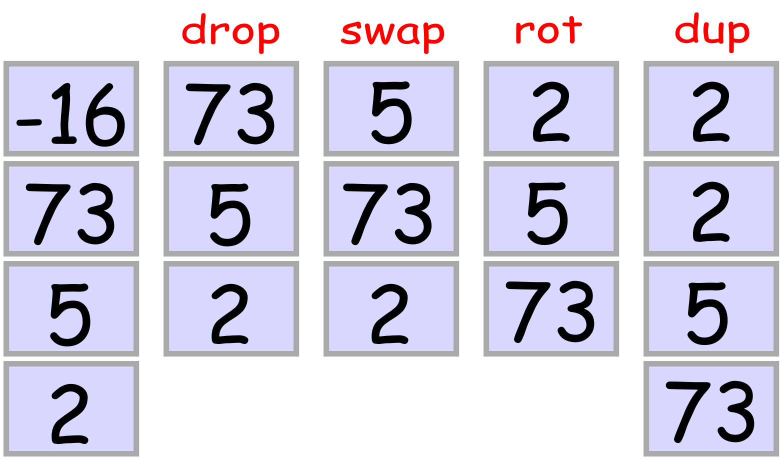
`.` consumes data value `-104`, leaving the stack empty. If we execute `.` on the now empty stack, the external interpreter aborts with a stack pointer error STACK UNDERFLOW ERROR.

The programming notation where the operands appear first, followed by the operator(s) is called Reverse Polish Notation (RPN).

Handling the parameter stack

Being a stack-based system, ESP32forth must provide ways to put numbers on the stack, remove them and rearrange their order. We have already seen that we can put numbers on the stack simply by typing them. We can also integrate numbers into the definition of a FORTH word.

The word `drop` removes a number from the top of the stack thus putting the next one on top. The word `swap` exchanges the first 2 numbers. `dup` copies the number at the top, pushing all other numbers down. `rot` rotates the first 3 numbers. These actions are



presented below.

The Return Stack and Its Uses

When compiling a new word, ESP32forth establishes links between the calling word and previously defined words that are to be invoked by the execution of the new word. This linking mechanism, at runtime, uses the rstack. The address of the next word to be invoked is placed on the back stack so that when the current word has finished executing, the system knows where to move to the next word. Since words can be nested, there must be a stack of these return addresses.

In addition to serving as a reservoir of return addresses, the user can also store and retrieve from the return stack, but this must be done carefully because the return stack is essential to program execution. If you use the return battery for temporary storage, you must return it to its original state, otherwise you will likely crash the ESP32forth system. Despite the danger, there are times when using backstack 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 from the stack back to the top of the parameter stack. To simply remove a value from the top of the stack, there is the word `rdrop`. The word `r@` copies the top of the stack back into the parameter stack.

Memory usage

In ESP32forth, 32-bit numbers are fetched from memory to the stack by the word `@` (fetch) and stored from the top to memory by the word `!` (store). `@` expects an address on the stack and replaces the address with its contents. `!` expects a number and an address to store it. It places the number in the memory location referenced by the address, consuming both parameters in the process.

Unsigned numbers that represent 8-bit (byte) values can be placed in character-sized characters. memory cells using `c@` and `c!`.

```
create testVar
    cell allot
$f7 testVar c!
testVar c@ .    \ display 247
```

Variables

A variable is a named location in memory that can store a number, such as the intermediate result of a calculation, off the stack. For example :

```
variable x
```

creates a storage location named, `x`, which executes leaving the address of its storage location at the top of the stack :

```
x .    \ display address
```

We can then retrieve or store at this address :

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

Constants

A constant is a number that you would not want to change while a program is running. The result of executing the word associated with a constant is the value of the data remaining on the stack.

```
\ define VSPI pins
19 constant VSPI_MISO
23 constant VSPI_MOSI
```

```
18 constant VSPI_SCLK
05 constant VSPI_CS

\ define SPI frequency port
4000000 constant SPI_FREQ

\ select SPI vocabulary
only FORTH  SPI also

\ initialize the SPI port
: init.VSPI ( -- )
    VSPI_CS OUTPUT pinMode
    VSPI_SCLK VSPI_MISO VSPI_MOSI VSPI_CS SPI.begin
    SPI_FREQ SPI.setFrequency
  ;
```

Pseudo-constant values

A value defined with **`value`** is a hybrid type of **`variable`** and **`constant`**. We set and initialize a value and it is invoked as we would a constant. We can also change a value like we can change a variable.

```
decimal
13 value thirteen
thirteen .      \ display: 13
47 to thirteen
thirteen .      \ display: 47
```

The word **`to`** also works in word definitions, replacing the value following it with whatever is currently at the top of the stack. You need to be careful that **`to`** is followed by a value defined by value and not something else.

Basic tools for memory allocation

The words **`create`** and **`allot`** are the basic tools for reserving memory space and attaching a label to it. For example, the following transcription shows a new dictionary entry **`graphic-array`** :

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

When executed, the word **`graphic-array`** stacks the address of the first entry.

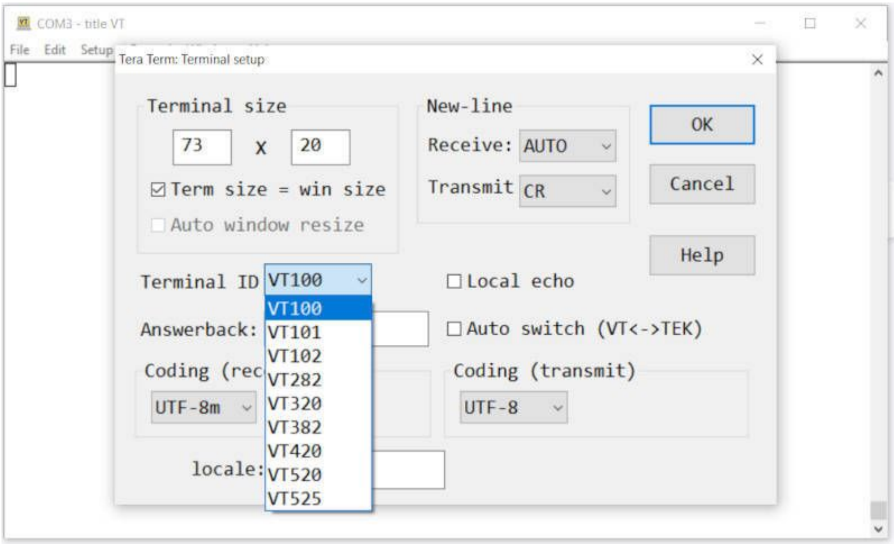
We can now access the memory allocated to **`graphic-array`** using the fetch and store words explained earlier. To calculate the address of the third byte assigned to **`graphic-array`** we can write **`graphic-array 2 +`**, remembering that the indices start at 0.

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

Text colors and display position on terminal

ANSI coding of terminals

If you are using terminal software to communicate with ESP32forth, there is a good chance that this terminal emulates a VT type terminal or equivalent. Here, TeraTerm configured to emulate a VT100 terminal:



These terminals have two interesting features :

- color the page background and the text to display
- position the display cursor

Both of these features are controlled by ESC (escape) sequences. This is how the words **bg** and **fg** are defined in ESP32forth :

```
forth definitions ansi
: fg ( n -- ) esc ." [38;5;" n. ." m" ;
: bg ( n -- ) esc ." [48;5;" n. ." m" ;
: normal   esc ." [0m" ;
: at-xy ( x y -- ) esc ." [" 1+ n. ." ;" 1+ n. ." H" ;
: page   esc ." [2J" esc ." [H" ;
```

The word **normal** overrides the coloring sequences defined by **bg** and **fg**.

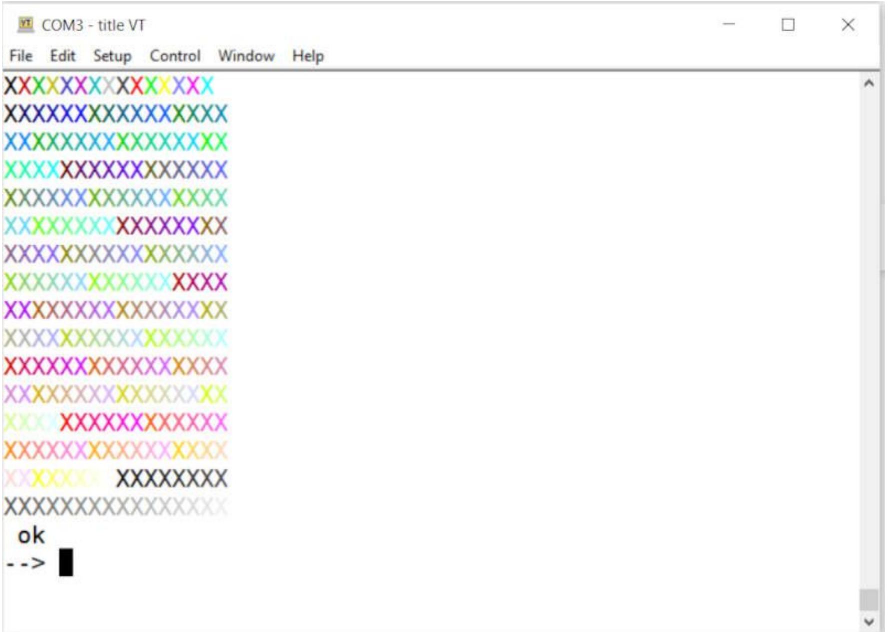
The word **page** clears the terminal screen and positions the cursor at the upper left corner of the screen.

Text coloring

Let's see how to color the text first :

```
: testFG ( -- )
    page
    16 0 do
        16 0 do
            j 16 * i + fg
            ." X"
        loop
        cr
    loop
    normal
  ;
```

Running **testFG** gives this on display :

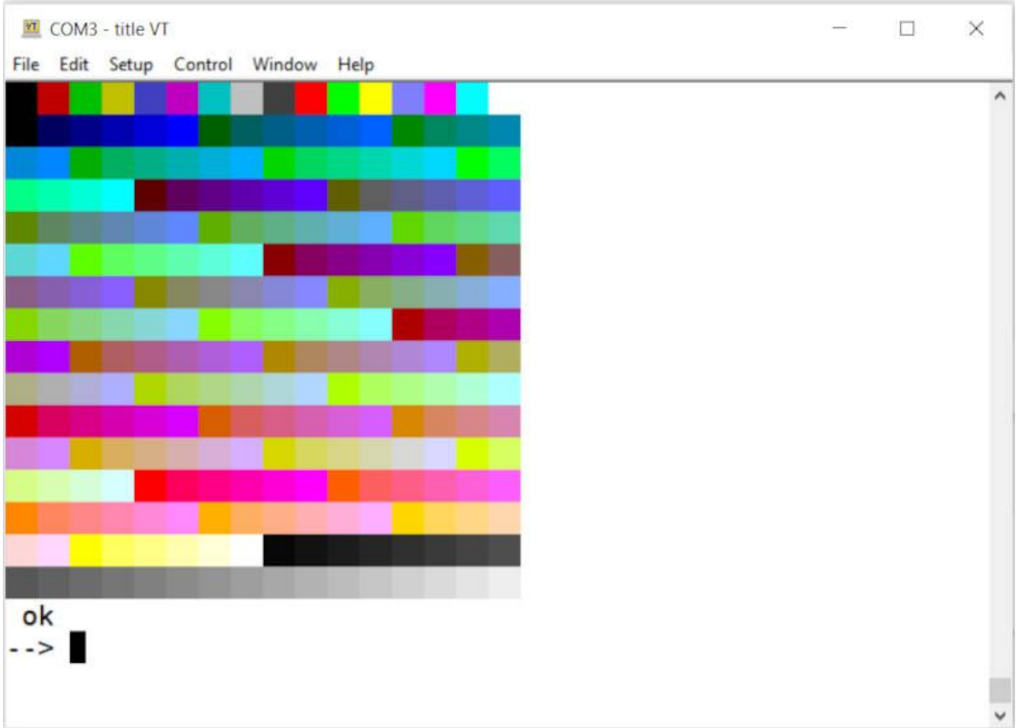


To test the background colors, we will proceed as follows :

```
: testBG ( -- )
    page
    16 0 do
        16 0 do
            j 16 * i + bg
            space space
        loop
        cr
    loop
    normal
```

```
  ;
```

Running testBG gives this on display :



Display position

The terminal is the simplest solution to communicate with ESP32forth. With ANSI escape sequences it is easy to improve the presentation of data.

```
09 constant red
11 constant yellow
14 constant cyan
15 constant whyte
: box { x0 y0 xn yn color -- }
    color bg
    yn y0 - 1+    \ determine height
    0 do
        x0  y0 i + at-xy
        xn x0 - spaces
    loop
    normal
  ;

: 3boxes ( -- )
    page
    2 4 20 6 cyan box
```

```
    8 6 28 8 red  box
    14 8  36 10 yellow box
    0 0 at-xy
  ;
```

Running **3boxes** shows this :

You are now equipped to create simple and effective interfaces allowing interaction with FORTH definitions compiled by ESP32forth.

Local variables with ESP32Forth

Introduction

The FORTH language processes data primarily through the data stack. This very simple mechanism offers unrivaled performance. Conversely, following the flow of data can quickly become complex. Local variables offer an interesting alternative.

The fake stack comment

If you follow the different FORTH examples, you will have noticed the stack comments framed by (and `)` . Example:

```
\ addition two unsigned values, leaves sum and carry on the stack
: um+ ( u1 u2 -- sum carry )
    \ here the definition
  ;
```

Here, the comment `( u1 u2 -- sum carry )` has absolutely no action on the rest of the FORTH code. This is pure commentary.

When preparing a complex definition, the solution is to use local variables framed by `{` and `}` . Example :

```
: 2OVER { a b c d }
    a b c d a b
  ;
```

We define four local variables `a` `b` `c` and `d`.

The words `{` and `}` are similar to the words `(` and `)` but do not have the same effect at all. Codes placed between `{` and `}` are local variables. The only constraint: do not use variable names that could be FORTH words from the FORTH dictionary. We might as well have written our example like this :

```
: 2OVER { varA varB varC varD }
    varA varB varC varD varA varB
  ;
```

Each variable will take the value of the data stack in the order of their deposit on the data stack. here, 1 goes into `varA`, 2 into `varB`, etc.:

```
--> 1 2 3 4
 ok
1 2 3 4 --> 2over
 ok
1 2 3 4 1 2 -->
```

Our fake stack comment can be completed like this :

```
: 2OVER { varA varB varC varD -- varA varB }
  ......
```

The characters following `--` have no effect. The only point is to make our fake comment look like a real stack comment.

Action on local variables

Local variables act exactly like pseudo-variables defined by `value`. Example :

```
: 3x+1 { var -- sum }
    var 3 * 1 +
  ;
```

A le même effet que ceci:

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

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

We assign a value to a local variable with the word `to` or `+to` to increment the content of a local variable. In this example, we add a local variable `result` initialized to zero in the code of our word:

```
: a+bEXP2 { varA varB -- (a+b)EXP2 }
    0 { result }
    varA varA *      to result
    varB varB *     +to result
    varA varB * 2 * +to result
    result
  ;
```

Isn't it more readable than this?

```
: a+bEXP2 ( varA varB -- result )
    2dup
    * 2 * >r
    dup *
    swap dup * +
    r> +
  ;
```

Here is a final example, the definition of the word `um+` which adds two unsigned integers and leaves the sum and the overflow value of this sum on the data stack:

```
\ add two unsigned integers, leaves sum and carry on the stack
: um+ { u1 u2 -- sum carry }
    0 { sum }
    cell for
        aft
            u1 $100 /mod to u1
            u2 $100 /mod to u2
            +
            cell 1- i - 8 * lshift  +to sum
        then
    next
    sum
    u1 u2 + abs
  ;
```

Here is a more complex example, rewriting **DUMP** using local variables:

```
\ local variables in DUMP:
\ START_ADDR      \ first address for dump
\ END_ADDR        \ last address for dump
\ 0START_ADDR     \ first address for loop in dump
\ LINES           \ number of lines for dump loop
\ myBASE          \ current numerical base
internals
: dump ( start len -- )
    cr cr ." --addr---  "
    ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------
chars-----"
    2dup + { END_ADDR }             \ store latest address to dump
    swap { START_ADDR }             \ store START address to dump
    START_ADDR 16 / 16 * { 0START_ADDR } \ calc. addr for loop start
    16 / 1+ { LINES }
    base @ { myBASE }               \ save current base
    hex
    \ outer loop
    LINES 0 do
        0START_ADDR i 16 * +        \ calc start address for current
line
        cr <# # # # #  [char] - hold # # # # #> type
        space space     \ and display address
        \ first inner loop, display bytes
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
            ca@ <# # # #> type space \ display byte in format: NN
        loop
```

```
        space
        \ second inner loop, display chars
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
            \ display char if code in interval 32-127
            ca@     dup 32 < over 127 > or
            if      drop [char] . emit
            else    emit
            then
        loop
    loop
    myBASE base !               \ restore current base
    cr cr
  ;
forth
```

The use of local variables greatly simplifies data manipulation on stacks. The code is more readable. Note that it is not necessary to pre-declare these local variables, it is enough to designate them when using them, for example: **`base @ { myBASE }`**.

WARNING: if you use local variables in a definition, no longer use the words **`>r`** and **`r>`**, otherwise you risk disrupting the management of local variables. Just look at the decompilation of this version of **`DUMP`** to understand the reason for this warning:

```
: dump  cr cr s" --addr---  " type
    s" 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----" type
    2dup + >R SWAP >R -4 local@ 16 / 16 * >R 16 / 1+ >R base @ >R
    hex -8 local@ 0 (do) -20 local@ R@ 16 * + cr
    <# # # # # 45 hold # # # # #> type space space
    16 0 (do) -28 local@ j 16 * R@ + + CA@ <# # # #> type space 1 (+loop)
    0BRANCH rdrop rdrop space 16 0 (do) -28 local@ j 16 * R@ + + CA@ DUP 32 < OVER 127 > OR
    0BRANCH DROP 46 emit BRANCH emit 1 (+loop) 0BRANCH rdrop rdrop 1 (+loop)
    0BRANCH rdrop rdrop -4 local@ base ! cr cr rdrop rdrop rdrop rdrop rdrop ;
```

Data structures for ESP32forth

Preamble

ESP32forth is a 32-bit version of the FORTH language. Those who have practiced FORTH since its beginnings have programmed with 16-bit versions. This data size is determined by the size of the elements deposited on the data stack. To find out the size in bytes of the elements, you must execute the word cell. Running this word for ESP32forth :

```
cell .  \ display 4
```

The value 4 means that the size of the elements placed on the data stack is 4 bytes, or 4x8 bits = 32 bits.

With a 16-bit FORTH version, cell will stack the value 2. Likewise, if you use a 64-bit version, cell will stack the value 8.

Tables in FORTH

Let's start with fairly simple structures : tables. We will only discuss one- or two-dimensional arrays.

One-dimensional 32-bit data array

This is the simplest type of table. To create a table of this type, we use the word **`create`** followed by the name of the table to create :

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

Dans ce tableau, on stocke 6 valeurs: 34, 37....12. Pour récupérer une valeur, il suffit d'utiliser le mot `@` en incrémentant l'adresse empilée par **`temperatures`** avec le décalage souhaité:

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        \ push addr on stack
    0 cell *        \ calculate offset 0
    +               \ add offset to addr
    @ .             \ display 34

temperatures        \ push addr on stack
    1 cell *        \ calculate offset 0
    +               \ add offset to addr
    @ .             \ display 37
```

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

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

Mots de définition de tableaux

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

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

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

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

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

Read and write in a table

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

```
arrayC myCTemps
    6 allot              \ allocate 6 bytes
    0 myCTemps 6 0 fill \ fill this 6 bytes with value 0
```

```
32 0 myCTemps c!          \ store 32 in myCTemps[0]
25 5 myCTemps c!          \ store 25 in myCTemps[5]
0 myCTemps c@ .           \ display 32
```

In our example, the array contains 6 elements. With ESP32forth, there is enough memory space to process much larger arrays, with 1,000 or 10,000 elements for example. It's easy to create multi-dimensional tables. Example of a two-dimensional array :

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

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

Simply reserve a memory space which is the product of the dimensions X and Y of the table to use. Now let's see how to manage this two-dimensional array :

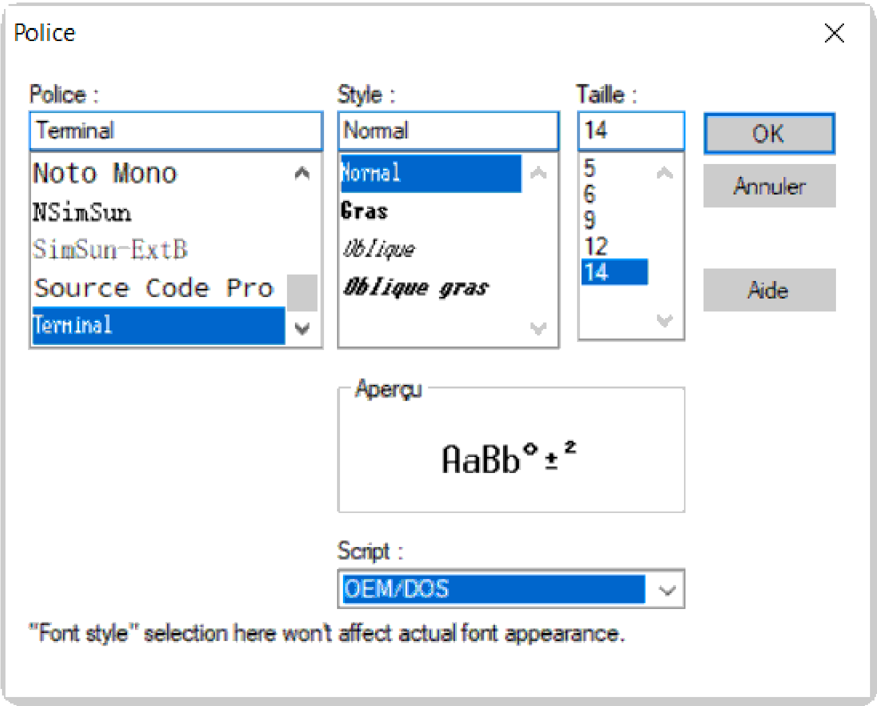
```
: xySCRaddr { x y -- addr }
    SCR_WIDTH y *
    x + mySCREEN +
  ;
: SCR@ ( x y -- c )
    xySCRaddr c@
  ;
: SCR! ( c x y -- )
    xySCRaddr c!
  ;
char X 15 5 SCR!      \ store char X at col 15 line 5
15 5 SCR@ emit        \ display X
```

Practical example of managing a virtual screen

Before going further in our example of managing a virtual screen, let's see how to modify the character set of the TERA TERM terminal and display it.

Launch TERA TERM terminal :

- in the menu bar, click on *Setup*
- select *Font* and *Font*...
- configure the font below :



Here's how to display the table of available characters :

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

Here is the result of running `tableChars` :

```
--> tableChars
20    21 !  22 "  23 #  24 $  25 %  26 &  27 '  28 (  29 )  2A *  2B +  2C ,  2D -  2E .  2F /
30 0  31 1  32 2  33 3  34 4  35 5  36 6  37 7  38 8  39 9  3A :  3B ;  3C <  3D =  3E >  3F ?
40 @  41 A  42 B  43 C  44 D  45 E  46 F  47 G  48 H  49 I  4A J  4B K  4C L  4D M  4E N  4F O
50 P  51 Q  52 R  53 S  54 T  55 U  56 V  57 W  58 X  59 Y  5A Z  5B [  5C \  5D ]  5E ^  5F _
60 `  61 a  62 b  63 c  64 d  65 e  66 f  67 g  68 h  69 i  6A j  6B k  6C l  6D m  6E n  6F o
70 p  71 q  72 r  73 s  74 t  75 u  76 v  77 w  78 x  79 y  7A z  7B {  7C |  7D }  7E ~  7F
A0 á  A1 í  A2 ó  A3 ú  A4 ñ  A5 Ñ  A6 ª  A7 º  A8 ¿  A9 ®  AA ¬  AB ½  AC ¼  AD ¡  AE «  AF »
B0 ░  B1 ▒  B2 ▓  B3 │  B4 ┤  B5 Á  B6 Â  B7 À  B8 ©  B9 ╣  BA ║  BB ╗  BC ╝  BD ¢  BE ¥  BF ┐
C0 └  C1 ┴  C2 ┬  C3 ├  C4 ─  C5 ┼  C6 ã  C7 Ã  C8 ╚  C9 ╔  CA ╩  CB ╦  CC ╠  CD ═  CE ╬  CF ¤
D0 ð  D1 Ð  D2 Ê  D3 Ë  D4 È  D5 ı  D6 Í  D7 Î  D8 Ï  D9 ┘  DA ┌  DB █  DC ▄  DD ¦  DE Ì  DF ▀
E0 Ó  E1 ß  E2 Ô  E3 Ò  E4 õ  E5 Õ  E6 µ  E7 þ  E8 Þ  E9 Ú  EA Û  EB Ù  EC ý  ED Ý  EE ¯  EF ´
F0 -  F1 ±  F2 ‗  F3 ¾  F4 ¶  F5 §  F6 ÷  F7 ¸  F8 °  F9 ¨  FA ·  FB ¹  FC ³  FD ²  FE ■  FF
```

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

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

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

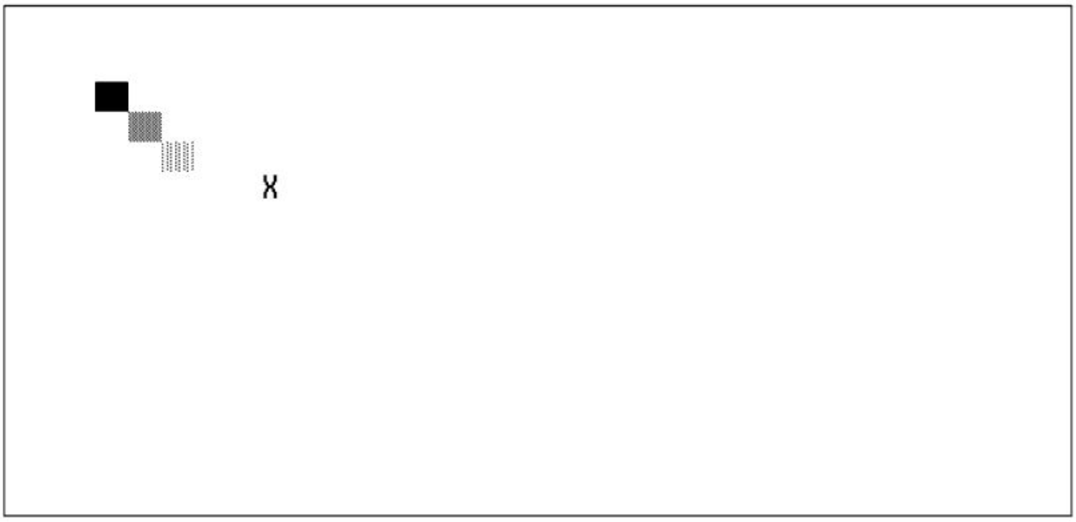
```
: dispLine { numLine -- }
    SCR_WIDTH numLine *
    mySCREEN + SCR_WIDTH type
  ;
```

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

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

And now, we define the word allowing us to display the content of our virtual screen. To clearly see the content of this virtual screen, we frame it with special characters :

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



Running our **dispScreen** word displays this :

In our virtual screen example, we show that managing a two-dimensional array has a concrete application. Our virtual screen is accessible for writing and reading. Here we display our virtual screen in the terminal window. This display is far from efficient.

Management of complex structures

ESP32forth has the **structures** vocabulary. The content of this vocabulary makes it possible to define complex data structures.

Voici un exemple trivial de structure:

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

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

The sole purpose of the **YMDHMS** word is to initialize and group the pointers in the complex structure. Here is how these pointers are used :

```
create DateTime
    YMDHMS allot

2022 DateTime >year  !
  03 DateTime >month !
  21 DateTime >day   !
```

```
  22 DateTime >hour  !
  36 DateTime >min   !
  15 DateTime >sec   !

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

DateTime .date
```

On a défini le mot `DateTime` qui est un tableau simple de 6 cellules 32 bits consécutives. L'accès à chacune des cellules est réalisée par l'intermédiaire du pointeur correspondant. On peut redéfinir l'espace alloué de notre structure `YMDHMS` en utilisant le mot `i8` pour pointer des octets:

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

create cDateTime
    cYMDHMS allot

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

: .cDate ( 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
  ;
cDateTime .cDate    \ affiche:
\  YEAR: 2022
\ MONTH: 3
\   DAY: 21
\    HH: 22
\    MM: 36
\    SS: 15
```

In this **cYMDHMS** structure, we kept the year in 32-bit format and reduced all other values to 8-bit integers. We see, in the **.cDate** code, that the use of pointers allows easy access to each element of our complex structure....

Definition of sprites

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

```
63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
    SCR_WIDTH SCR_HEIGHT * allot
    mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill
```

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

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

create myVscreen    \ define a screen 8x32 bytes
    32 c,           \ compile width
    08 c,           \ compile height
    myVscreen >width  c@
    myVscreen >height c@ * allot
```

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

```
: sprite: ( width height -- )
    create
        swap c, c,  \ compile width et height
    does>
  ;
2 1 sprite: blackChars
    $db c, $db c,
2 1 sprite: greyChars
    $b2 c, $b2 c,
blackChars >content 2 type   \ display content of sprite blackChars
```

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

```
5 7 sprite: char3
    $20 c,  $db c,  $db c,  $db c,  $20 c,
    $db c,  $20 c,  $20 c,  $20 c,  $db c,
    $20 c,  $20 c,  $20 c,  $20 c,  $db c,
    $20 c,  $db c,  $db c,  $db c,  $20 c,
    $20 c,  $20 c,  $20 c,  $20 c,  $db c,
    $db c,  $20 c,  $20 c,  $20 c,  $db c,
    $20 c,  $db c,  $db c,  $db c,  $20 c,
```

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

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

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

Result of displaying our sprite :

I hope the content of this chapter has given you some interesting ideas that you would like to share...

Displaying numbers and character strings

Change of numerical base

FORTH does not process just any numbers. The ones you used when trying the previous examples are single-precision signed integers. The definition domain for 32-bit integers is -2147483648 to 2147483647. Example :

```
2147483647 .     \ displays    2147483647
2147483647 1+ . \ displays    -2147483648
-1 u.            \ displays    4294967295
```

These numbers can be processed in any number base, with all number bases between 2 and 36 being valid :

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

You can choose an even larger numerical base, but the available symbols will fall outside the alpha-numeric set [0..9,A..Z] and risk becoming inconsistent.

The current numerical base is controlled by a variable named **BASE** and whose content can be modified. So, to switch to binary, simply store the value **2** in **BASE** . Example:

```
2 BASE !
```

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

ESP32forth has two pre-defined words allowing you to select different numerical bases:

- **DECIMAL** to select the decimal numeric base. This is the numerical base taken by default when starting ESP32forth;
- **HEX** to select the hexadecimal numeric base.

Upon selection of one of these numerical bases, the literal numbers will be interpreted, displayed or processed in this base. Any number previously entered in a number base other than the current number base is automatically converted to the current number base. Example :

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

One can define one's own numerical base by defining the appropriate word or by storing this base in **BASE** . Example :

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

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

```
VARIABLE RANGE_BASE       \ RANGE-BASE variable definition
BASE @ RANGE_BASE !       \ storage BASE contents in RANGE-BASE
HEX FF 10 + .             \ displays 10F
RANGE_BASE @ BASE !       \ restores BASE with contents of RANGE-BASE
```

In a definition **:** , the contents of **BASE** can pass through the return stack :

```
: OPERATION ( ---)
    BASE @ >R         \ stores BASE on back stack
    HEX FF 10 + .     \ operation of the previous example
    R> BASE ! ;       \ restores initial BASE value
```

WARNING : the words **>R** and **R>** cannot be used in interpreted mode. You can only use these words in a definition that will be compiled.

Definition of new display formats

Forth has primitives allowing you to adapt the display of a number to any format. These primitives only deal with double precision numbers:

- **<#** begins a format definition sequence;
- **#** inserts a digit into a format definition sequence;
- **#S** is equivalent to a succession of **#** ;
- **HOLD** inserts a character into a format definition;
- **#>** completes a format definition and leaves on the stack the address and length of the string containing the number to display.

These words can only be used within a definition. Example, either to display a number expressing an amount denominated in euros with the comma as a decimal separator :

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

Execution examples:

```
35   .EUROS               \ displays     0,35 EUR
35.75 .EUROS              \ displays     35,75 EUR
1015 3575 + .EUROS        \ displays     45,90 EUR
```

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

At runtime, a display format sequence deals exclusively with signed or unsigned 32-bit integers. The concatenation of the different elements of the string is done from right to left, i.e. starting with the least significant digits.

The processing of a number by a display format sequence is executed based on the current numeric base. The numerical base can be modified between two digits.

Here is a more complex example demonstrating the compactness of FORTH. This involves writing a program converting any number of seconds into HH:MM:SS format:

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

Execution examples :

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

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

Another example, to define a program converting a single precision decimal integer into binary and displaying it in the format bbbb bbbb bbbb bbbb:

```
: FOUR-DIGITS ( ---)
    # # # # 32 HOLD ;
: AFB ( d ---)              \ format 4 digits and a space
    BASE @ >R               \ Current database backup
```

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

Execution example :

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

Another example is to create a telephone diary where one or more telephone numbers are associated with a surname. We define a word by surname :

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

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

Displaying characters and character strings

A character is displayed using the word **EMIT** :

```
65 EMIT          \ displays A
```

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

```
variable #out
: #out+! ( n -- )
    #out +!                 \ increment #out
  ;
: (.)  ( n -- a l )
  DUP ABS <# #S ROT SIGN #>
;
: .R  ( n l -- )
  >R (.) R> OVER - SPACES TYPE
```

```
;
: ASCII-SET ( ---)
    cr 0 #out !
    128 32
    DO
        I 3 .R SPACE        \ displays character code
        4 #out+!
        I EMIT 2 SPACES     \ displays character
        3 #out+!
        #out @ 77 =
        IF
            CR   0 #out !
        THEN
    LOOP ;
```

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

```
hex  ASCII-SET
 20      21 !    22 "    23 #    24 $    25 %    26 &    27 '    28 (    29 )    2A *
 2B +    2C ,    2D -    2E .    2F /    30 0    31 1    32 2    33 3    34 4    35 5
 36 6    37 7    38 8    39 9    3A :    3B ;    3C <    3D =    3E >    3F ?    40 @
 41 A    42 B    43 C    44 D    45 E    46 F    47 G    48 H    49 I    4A J    4B K
 4C L    4D M    4E N    4F O    50 P    51 Q    52 R    53 S    54 T    55 U    56 V
 57 W    58 X    59 Y    5A Z    5B [    5C \    5D ]    5E ^    5F _    60 `    61 a
 62 b    63 c    64 d    65 e    66 f    67 g    68 h    69 i    6A j    6B k    6C l
 6D m    6E n    6F o    70 p    71 q    72 r    73 s    74 t    75 u    76 v    77 w
 78 x    79 y    7A z    7B {    7C |    7D }    7E ~    7F     ok
```

Character strings are displayed in various ways. The first, usable in compilation only, displays a character string delimited by the character " (quote mark):

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

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

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

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

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

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

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

```
CR TITLE CR CR LINE1 CR TYPE
\ displays:
\ GENERAL MENU
\
\ E..Data logging
```

One or more spaces can be added at the start or end of the display of an alphanumeric string :

```
SPACE           \ displays a space character
10 SPACES       \ displays 10 space characters
```

String variables

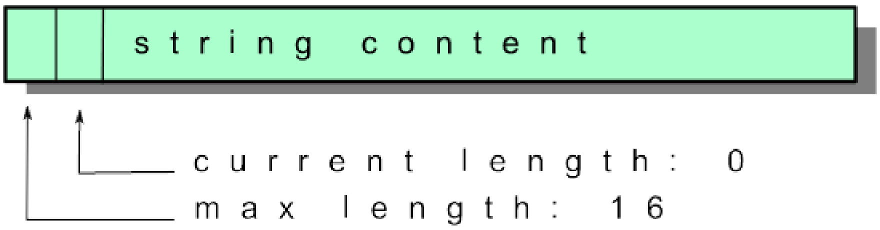
Alpha-numeric text variables do not exist natively in ESP32forth. Here is the first attempt to define the word `string` :

```
\ define a strvar
: string  ( comp: n --- names_strvar | exec: --- addr len )
    create
        dup
        c,      \ n is maxlength
        0 c,    \ 0 is real length
        allot
    does>
        2 +
        dup 1 - c@
  ;
```

A character string variable is defined like this:

```
16 string strState
```

Here is how the memory space reserved for this text variable is organized:



Text variable management word code

Here is the complete source code for managing text variables:

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

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

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

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

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

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

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

\ extract n chars right from string
: right$  ( str1 n --- str2 )
```

```
    0 max over min >r + r@ - r>
    ;

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

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

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

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

Creating an alphanumeric character string is very simple :

```
64 string myNewString
```

Here we create an alphanumeric variable **myNewString** which can contain up to 64 characters.

To display the contents of an alphanumeric variable, simply use **type** . Example :

```
s" This is my first example.." myNewString $!
myNewString type   \ display: This is my first example..
```

If we try to save a character string longer than the maximum size of our alphanumeric variable, the string will be truncated:

```
s" This is a very long string, with more than 64 characters. It
can't store complete"
myNewString $!
myNewString type
  \ displays: This is a very long string, with more than 64
characters. It can
```

Adding character to an alphanumeric variable

Some devices, the LoRa transmitter for example, require processing command lines containing the non-alphanumeric characters The word **c+$!** allows this code insertion:

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

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

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

Here is a clever way to create an alphanumeric variable allowing you to transmit a carriage return, a **CR+LF** compatible with the end of commands for the LoRa transmitter:

```
2 string $crlf
$0d $crlf c+$!
$0a $crlf c+$!

: crlf ( -- )       \ same action as cr, but adapted for LoRa
    $crlf type
  ;
```

Vocabularies with ESP32forth

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

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

List of vocabularies

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

To see the list of different vocabularies available in ESP32forth, we will use the word **voclist** :

```
--> internals voclist \ displays
registers
ansi
editor
streams
tasks
rtos
sockets
Serial
ledc
SPIFFS
SD_MMC
SD
Wireless
Wire
ESP
structures
internalized
internals
FORTH
```

This list is not limited. Additional vocabularies may appear if we compile certain extensions.

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

List of vocabulary contents

To see the content of a vocabulary, we use the word **vlist** having previously selected the appropriate vocabulary :

```
vlist sockets
```

Select **sockets** vocabulary and displays its contents :

```
--> sockets vlist\displays:
ip. ip# ->h_addr ->addr! ->addr@ ->port! ->port@ sockaddr l, s, bs, SO_REUSEADDR
SOL_SOCKET sizeof(sockaddr_in) AF_INET SOCK_RAW SOCK_DGRAM SOCK_STREAM
socket setsockopt bind listen connect sockaccept select poll send sendto
sendmsg recv recvfrom recvmsg gethostbyname errno sockets-builtins
```

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

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

The same word can be defined in two different vocabularies and have two different actions : the word **l** is defined in both **asm** and **editor vocabularies.**

This is even more obvious with the word **server , defined in the httpd** , **telnetd** and **web-interface** vocabularies.

Using vocabulary words

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

Here, we define a word **serial2-type** which uses the word **Serial2.write defined in the serial** vocabulary :

```
serial \ Selection vocabulary Serial
: serial2-type (an --)
Serial2.write drop
;
```

The second solution allows you to integrate a single word from a specific vocabulary :

```
: serial2-type (an --)
[ serial ] Serial2.write [ FORTH ] \ compile word from vocabulary
serial
drop
;
```

The selection of a vocabulary can be carried out implicitly from another word in the FORTH vocabulary.

Chaining of vocabularies

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

Before creating a chain of vocabularies, we restrict the search order with the word **only** :

```
asm xtensa
order\display: xtensa >> asm >> FORTH
only
order\display: FORTH
```

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

```
only
order\display: FORTH
asm also
order\display: asm >> FORTH
xtensa
order\display: xtensa >> asm >> FORTH
```

Here is a compact chaining sequence :

```
only asm also xtensa
```

The last vocabulary thus chained will be the first explored when we execute or compile a new word.

```
only
order\display: FORTH
also ledc also serial also SPIFFS
order \ displays: SPIFFS >> FORTH
\ Serial >> FORTH
\ ledc >> FORTH
\ FORTH
```

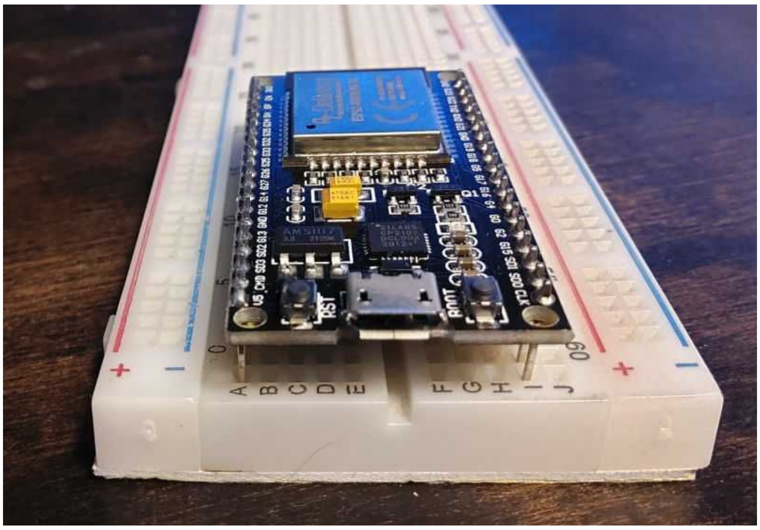
The search order, here, will start with the **SPIFFS vocabulary** , then **Serial** , then **ledc** and finally the **FORTH vocabulary** :

- if the searched word is not found, there is a compilation error ;
- if the word is found in a vocabulary, it is this word that will be compiled, even if it is defined in the following vocabulary.

Adapt breadboards to ESP32 board

Breadboards for ESP32

You have just received your ESP32 cards. And first bad surprise, this card fits very poorly on the test board :

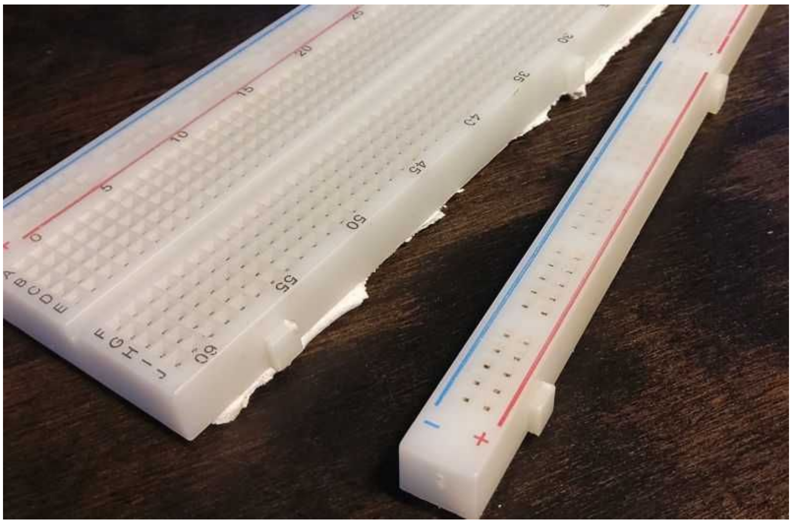


There is no breadboard specifically suited to ESP32 boards.

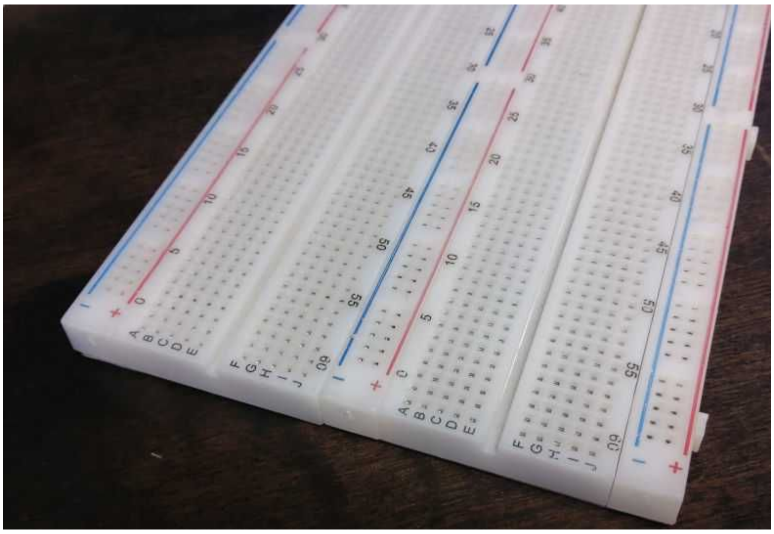
Build a breadboard suitable for the ESP32 board

We're going to build our own test plate. For this, two identical test plates must be available.

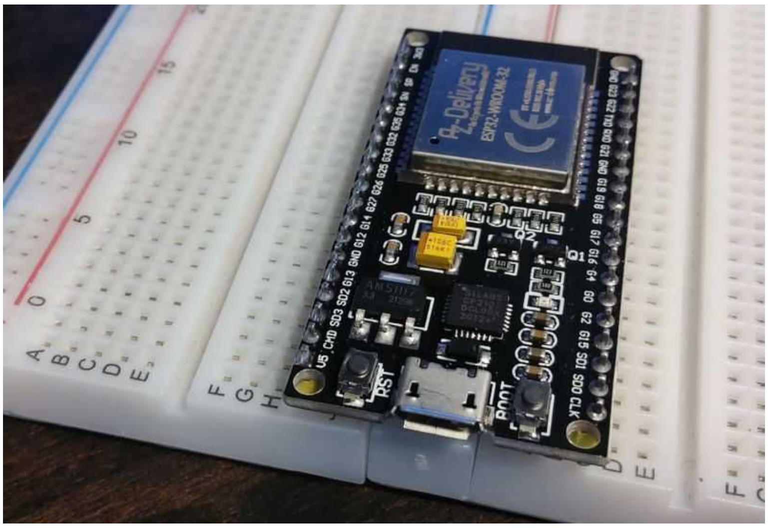
On one of breadboard, we will remove a power line. To do this, use a cutter and cut from below. You should be able to separate this power line like this :



We can then reassemble the entire tbreadboard with this board. You have rafters on the sides of the test plates to connect them together :



And there you go! We can now install our ESP32 card :



The I/O ports can now be used without difficulty.

Alimenter la carte ESP32

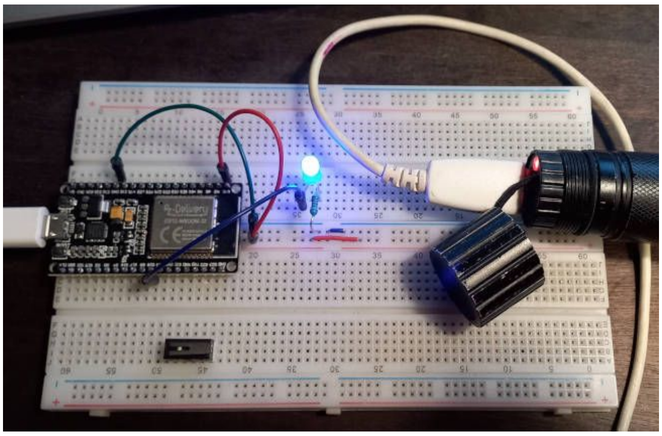
Choix de la source d'alimentation

Nous allons voir ici comment alimenter une carte ESP32. Le but est de donner des solutions pour exécuter les programmes FORTH compilés par ESP32forth.

Alimentation par le connecteur mini-USB

C'est la solution la plus simple. On remplace l'alimentation provenant du PC par une source différente:

- un bloc d'alimentation secteur comme ceux utilisés pour recharger un téléphone mobile;
- une batterie de secours pour téléphone mobile (power bank).



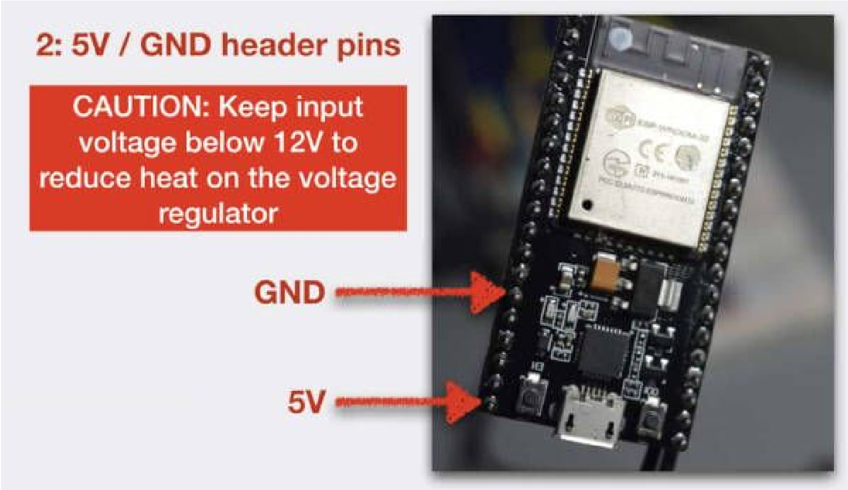
Ici, on alimente notre carte ESP32 avec une batterie de secours pour appareils mobiles.

Alimentation par le pin 5V

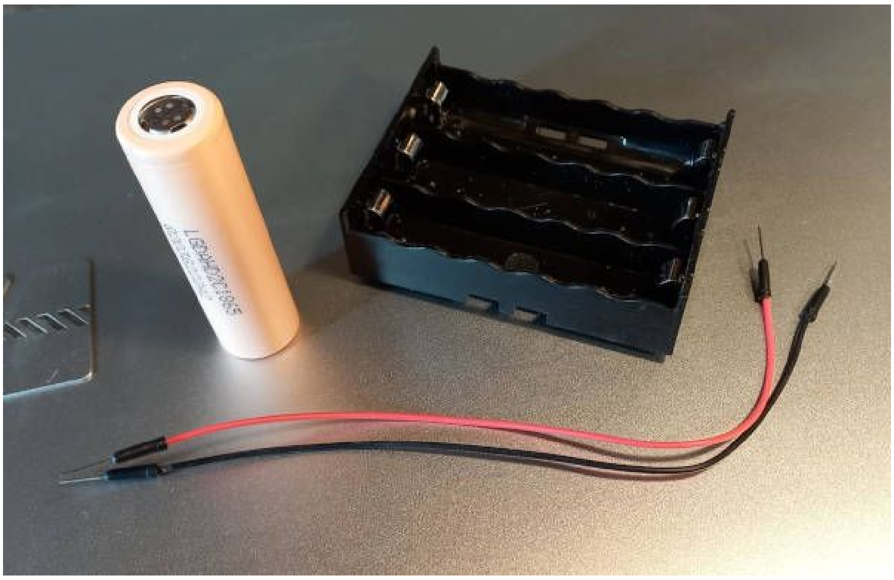
La deuxième option consiste à connecter une alimentation externe non régulée à la broche 5 V et à la masse. Tout ce qui se situe entre 5 et 12 volts devrait fonctionner.

Mais il est préférable de maintenir la tension d'entrée à environ 6 ou 7 Volts pour éviter de perdre trop de puissance sous forme de chaleur sur le régulateur de tension.

Voici les bornes permettant une alimentation externe 5-12V:



Pour exploiter l'alimentation 5V, il faut ce matériel:



- deux batteries lithium 3,7V
- un support batterie
- deux fils dupont

On soude une extrémité de chaque fil dupont aux bornes du support batteries. Ici, notre support accepte trois batteries. Nous n'exploiterons qu'un seul logement à batterie. Les batteries sont montées en série.

Une fois les fils dupont soudés, on installe la batterie et on vérifie que la polarité de sortie est bien respectée:

Maintenant, on peut alimenter notre carte ESP32 par le pin 5V.

ATTENTION: la tension batterie doit être entre 5 à 12 Volts.

Démarrage automatique d'un programme

Comment être certain que la carte ESP32 fonctionne bien une fois alimentée par nos batteries?

La solution la plus simple est d'installer un programme et de paramétrer ce programme pour qu'il démarre automatiquement à la mise sous tension de la carte ESP32. Compilez ce programme:

```
18 constant myLED

0 value LED_STATE

: led.on ( -- )
    HIGH  dup myLED pin
    to LED_STATE
  ;

: led.off ( -- )
    LOW  dup myLED pin
    to LED_STATE
  ;
timers also \ select timers vocabulary

: led.toggle ( -- )
    LED_STATE if
        led.off
    else
        led.on
    then
    0 rerun
  ;

: led.blink ( -- )
    myLED output pinMode
    ['] led.toggle 500000 0 interval
    led.toggle
  ;

startup: led.blink
bye
```

Installez une LED sur le pin G18.

Coupez l'alimentation et rebranchez la carte ESP32. Si tout s'est bien passé, la LED doit clignoter au bout de quelques secondes. C'est le signe que le programme s'exécute au démarrage de la carte ESP32.

Débranchez le port USB et branchez la batterie. La carte ESP32 doit démarrer et la LED clignoter.

Tout le secret tient dans la séquence `startup: led.blink`. Cette séquence fige le code FORTH compilé par ESP32forth et désigne le mot `led.blink` comme mot à exécuter au démarrage de ESP32forth une fois la carte ESP32 sous tension.

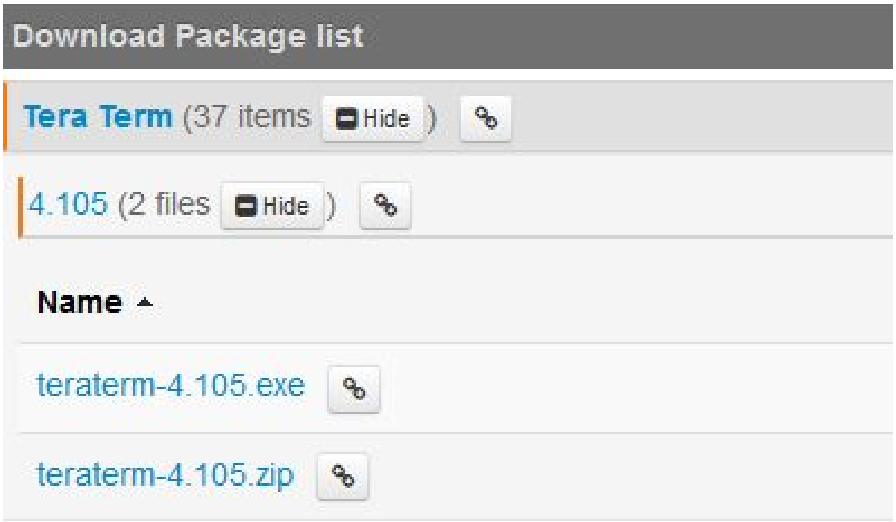
Install and use the Tera Term terminal on Windows

Install Tera Term

The English page for Tera Term is here:

https://ttssh2.osdn.jp/index.html.en

Go to the download page, get the exe or zip file:

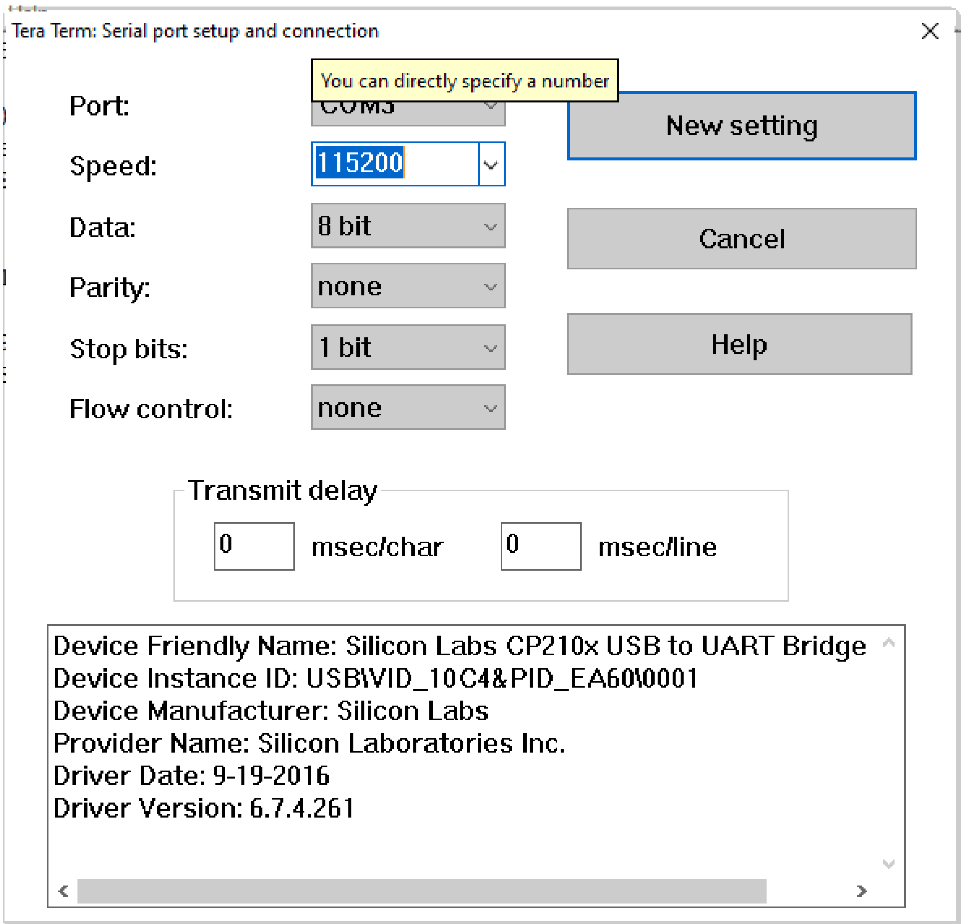


Install Tera Term. Installation is quick and easy.

Setting up Tera Term

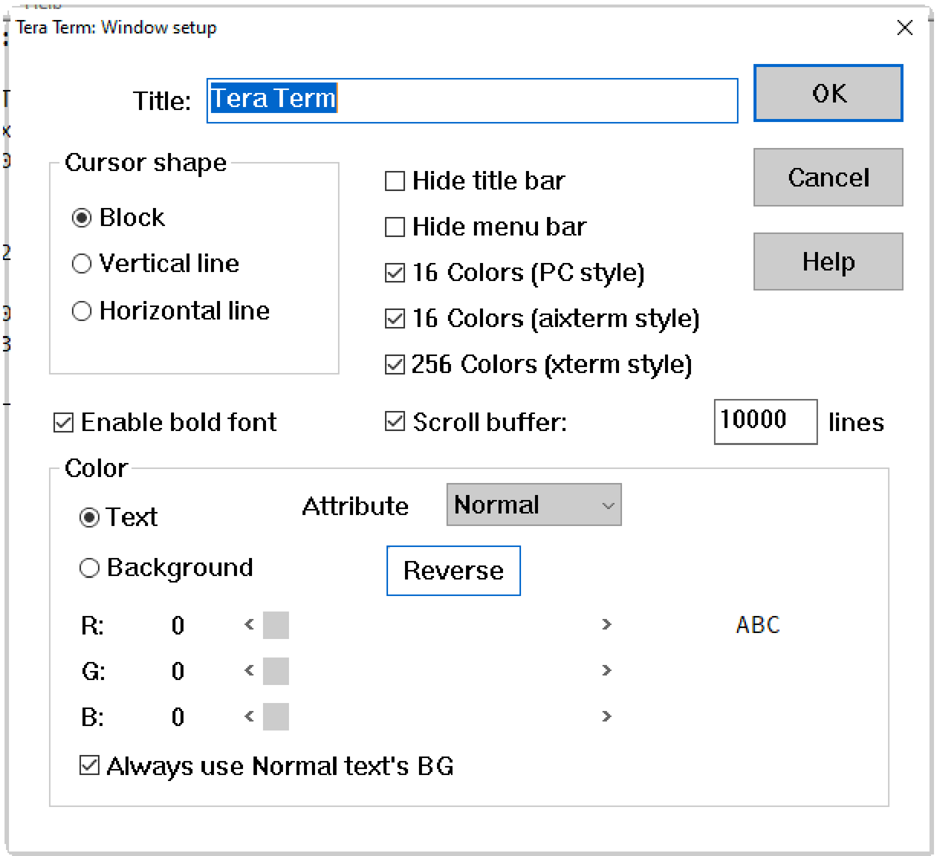
To communicate with the ESP32 card, you must adjust certain parameters:

- click on Configuration -> serial port



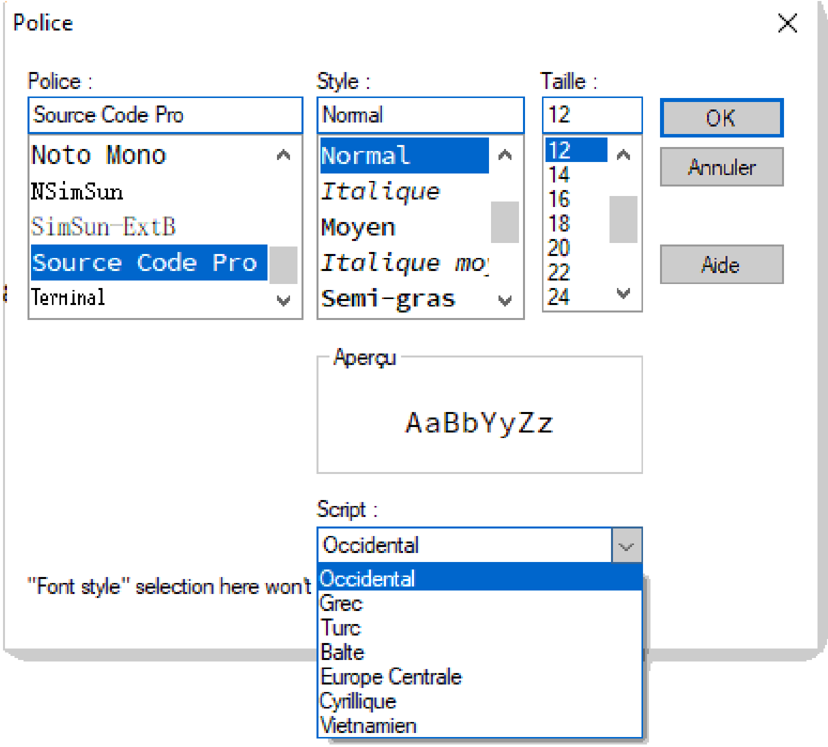
For comfortable viewing:

- click on Configuration -> window



For readable characters:

- click on Configuration -> font



To find all these settings the next time you launch the Tera Term terminal, save the configuration:

- click on *Setup* -> *Save setup*
- accept the name **TERATERM.INI** .

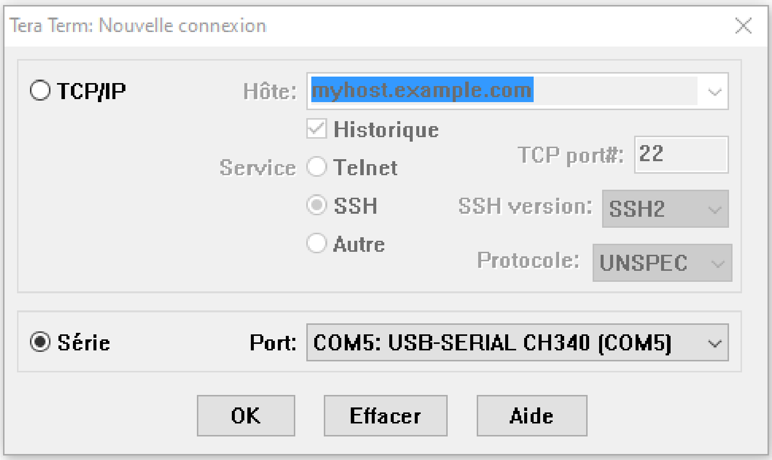
Using Tera Term

Once configured, close Tera Term.

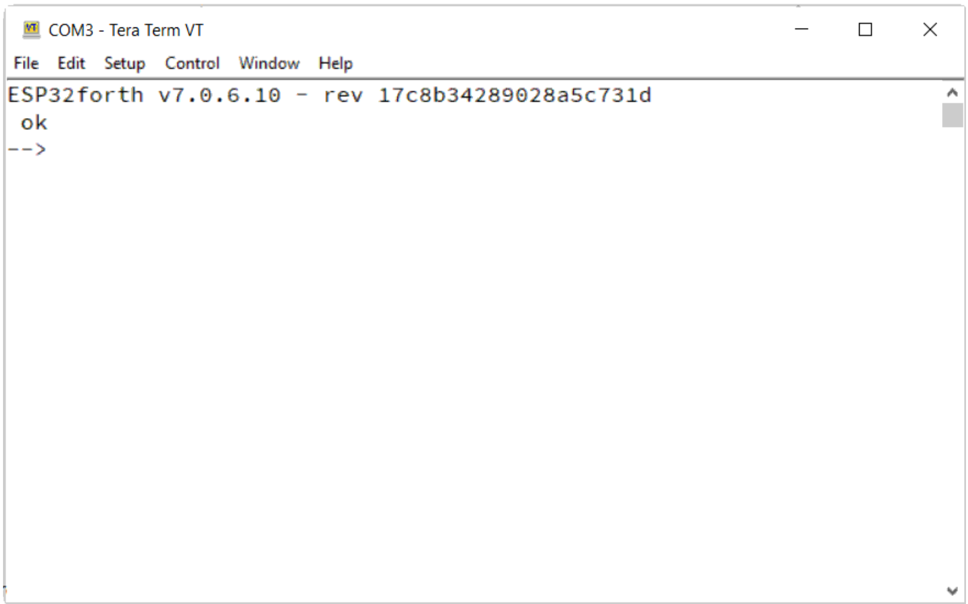
Connect your ESP32 board to an available USB port on your PC.

Relaunch Tera Term, then click *file* -> *new connection*

Select the serial port :



If everything went well, you should see this:

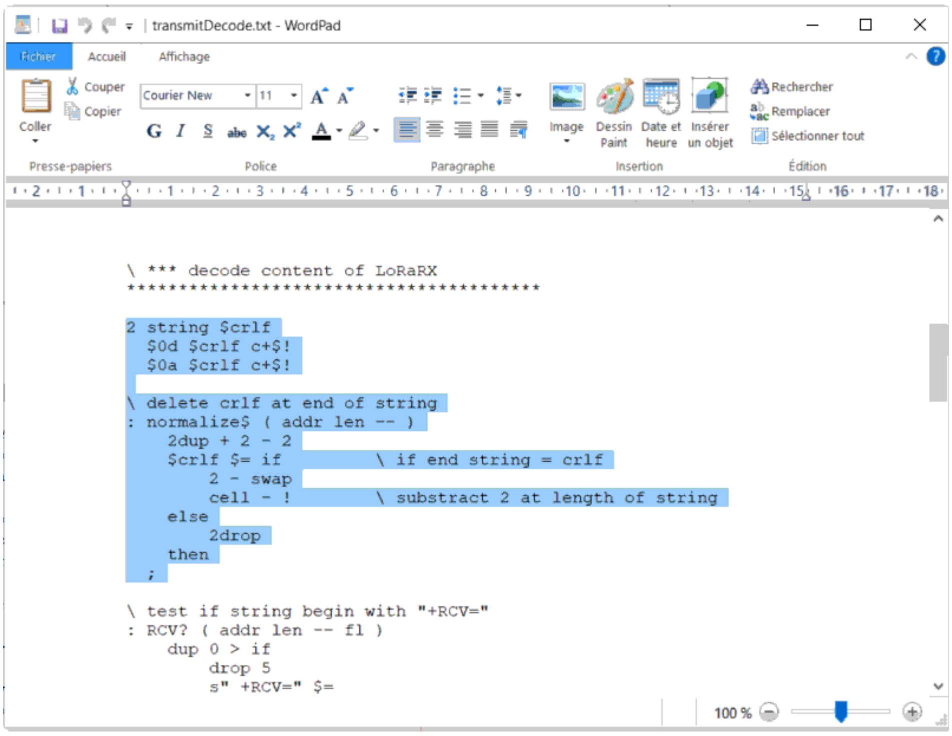


Compile source code in Forth language

First of all, let's remember that the FORTH language is on the ESP32 board! FORTH is not on your PC. Therefore, you cannot 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.

Then, we copy the source code to compile. Here, open source code with Wordpad:



The source code in FORTH language can be composed and edited with any text editor: notepad, PSpad, Wordpad..

Personally I use the Netbeans IDE. This IDE allows you to edit and manage source codes in many programming languages.

Select the source code or portion of code that interests you. Then click copy. The selected code is in the PC edit buffer.

Click on the Tera Term terminal window. Make Paste:

Simply validate by clicking OK and the code will be interpreted and/or compiled.

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

Ressources

in English

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

In french

- **ESP32 Forth** site in two languages (French, English) with lots of examples
  https://esp32.arduino-forth.com/

GitHub

- **Ueforth** resources maintained by Brad NELSON. Contains all Forth and C language source files for ESP32forth
  https://github.com/flagxor/ueforth
- **ESP32forth** source codes and documentation for ESP32forth. Resources maintained by Marc PETREMANN
  https://github.com/MPETREMANN11/ESP32forth
- **ESP32forthStation** resources maintained by Ulrich HOFFMAN. Stand alone Forth computer with LillyGo TTGO VGA32 single board computer and ESP32forth.
  https://github.com/uho/ESP32forthStation
- **ESP32Forth** resources maintained by F. J. RUSSO
  https://github.com/FJRusso53/ESP32Forth
- **esp32forth-addons** resources maintained by Peter FORTH
  https://github.com/PeterForth/esp32forth-addons
- **Esp32forth-org** Code repository for members of the Forth2020 and ESp32forth groups
  https://github.com/Esp32forth-org
-

Index