The great book for ESP32forth

version 1.5 - 24 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

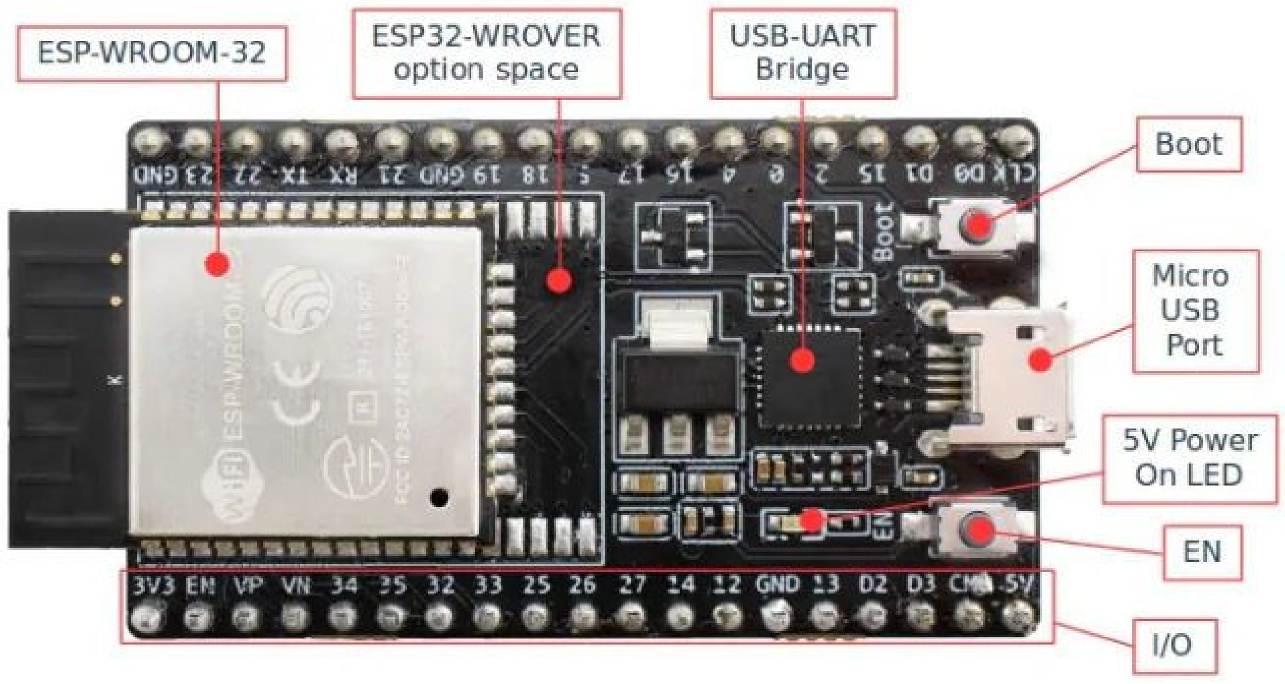
Discovery of the ESP32 card

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

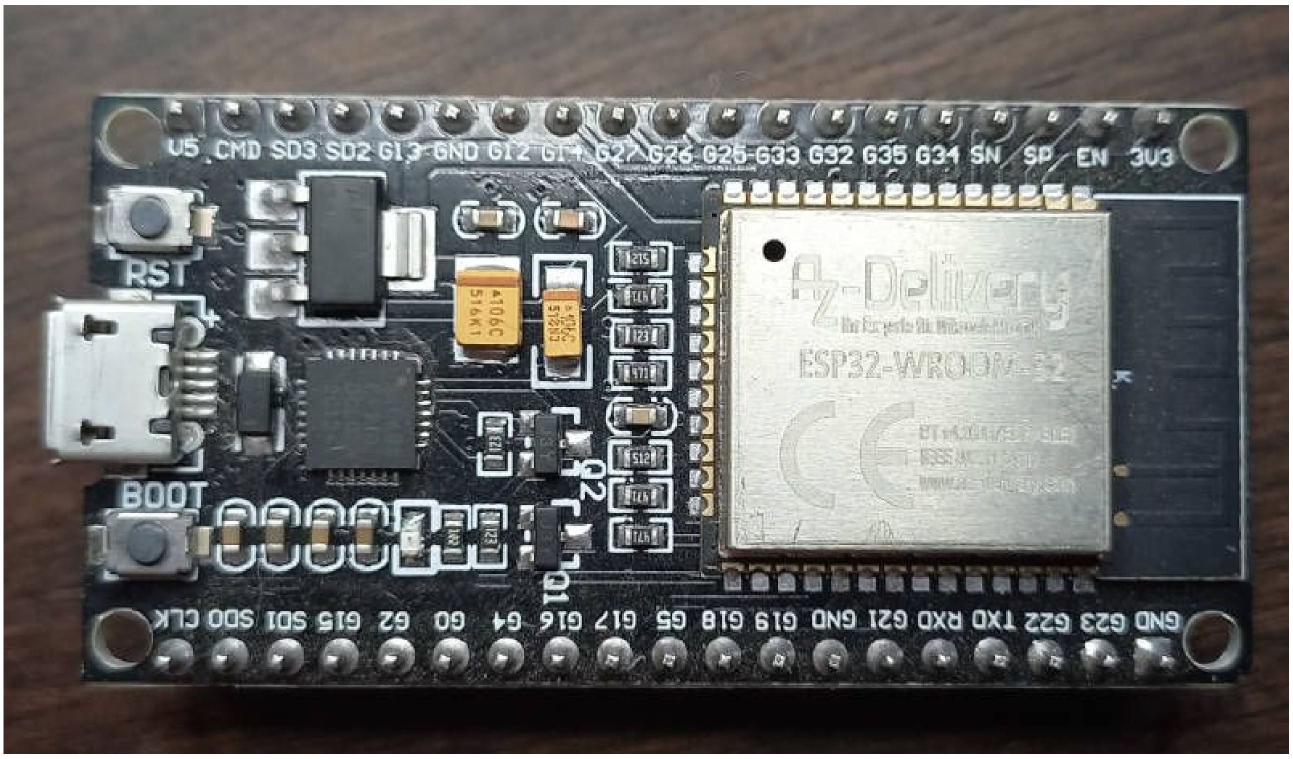
The ADC (analog-to-digital converter) and DAC (digital-to-analog converter) functionality are assigned to specific static pins. However, you can decide which pins are UART, I2C, SPI, PWM, etc. You just need to assign them in the code. This is possible thanks to the multiplexing function of the ESP32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

Most connectors have multiple uses.

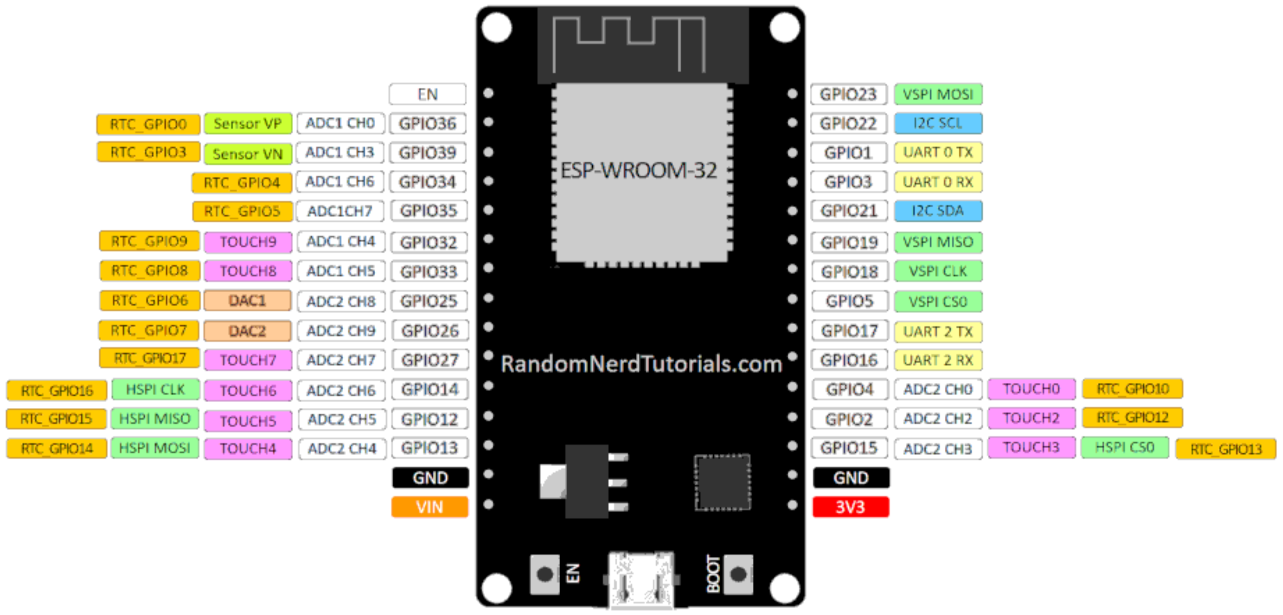
But what sets the ESP32 board apart is that it is equipped as standard with WiFi and Bluetooth support, something that ARDUINO boards only offer in the form of extensions.

GPIO inputs/outputs on ESP32

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


V5 CMD SD3 SD2 G13 GND G12 G14 G27 G26 G25 G33 G32 G35 G34 SN SP EN 3V3
RST
BOOT
ESP32-WROOM-32
GND G23 G22 TXD RXD G21 GND G19 G18 G5 G17 G16 G4 G0 G2 G15 SDI SDO CLK

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.


EN
RTC_GPIO0 Sensor VP ADC1 CH0 GPIO36
RTC_GPIO3 Sensor VN ADC1 CH3 GPIO39
RTC_GPIO4 ADC1 CH6 GPIO34
RTC_GPIO5 ADC1CH7 GPIO35
RTC_GPIO9 TOUCH9 ADC1 CH4 GPIO32
RTC_GPIO8 TOUCH8 ADC1 CH5 GPIO33
RTC_GPIO6 DAC1 ADC2 CH8 GPIO25
RTC_GPIO7 DAC2 ADC2 CH9 GPIO26
RTC_GPIO17 TOUCH7 ADC2 CH7 GPIO27
RTC_GPIO16 HSPI CLK TOUCH6 ADC2 CH6 GPIO14
RTC_GPIO15 HSPI MISO TOUCH5 ADC2 CH5 GPIO12
RTC_GPIO14 HSPI MOSI TOUCH4 ADC2 CH4 GPIO13
GND
VIN
ESP-WROOM-32
RandomNerdTutorials.com
EN
BOOT
GPIO23 VSPI MOSI
GPIO22 I2C SCL
GPIO1 UART 0 TX
GPIO3 UART 0 RX
GPIO21 I2C SDA
GPIO19 VSPI MISO
GPIO18 VSPI CLK
GPIO5 VSPI CS0
GPIO17 UART 2 TX
GPIO16 UART 2 RX
GPIO4 ADC2 CH0 TOUCH0 RTC_GPIO10
GPIO2 ADC2 CH2 TOUCH2 RTC_GPIO12
GPIO15 ADC2 CH3 TOUCH3 HSPI CS0 RTC_GPIO13
GND
3V3

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 :

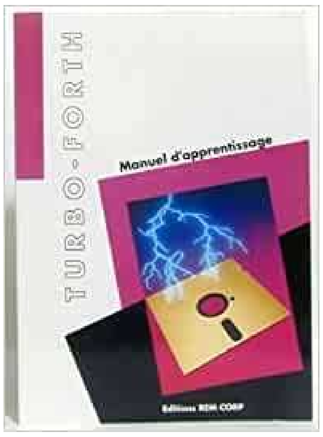
- 3 UART interface
- 2 I2C interfaces
- 3 SPI interfaces
- 16 PWM outputs
- 10 capacitive sensors
- 18 analog inputs (ADC)
- 2 DAC outputs

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 32-bit numbers.

The FORTH input line :

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

leaves the parameter stack as it is

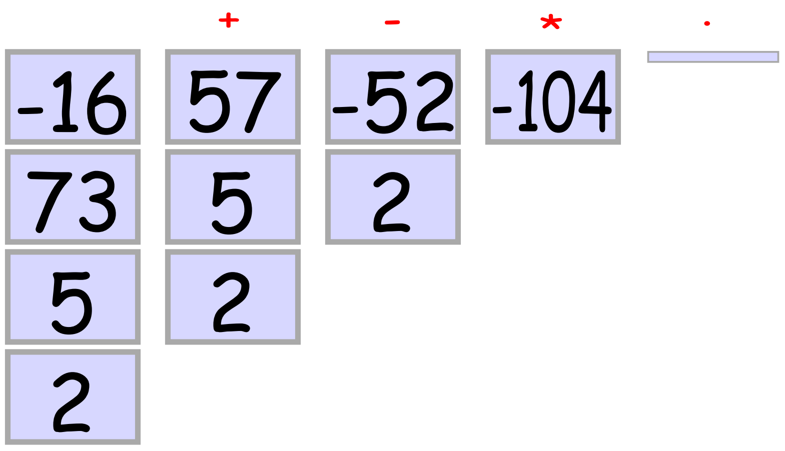
Cell	Content	comment
0	-16	(TOS) Top of stack
1	73	(NOS) Next in stack
2	5	
3	2	

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
```

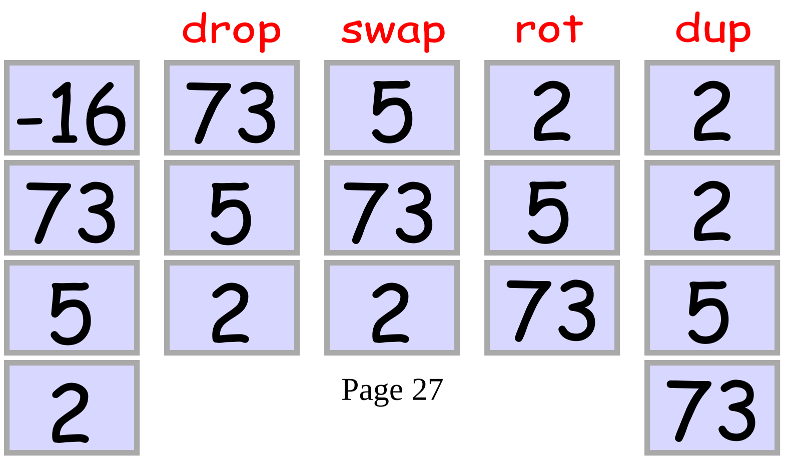
Note that ESP32forth conveniently displays the stack elements when interpreting each line and that the value of **-16** is displayed as a 32-bit unsigned integer. Furthermore, the word **.** consumes data value **-104**, leaving the stack empty. If we execute **.** on the now empty stack, the external 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 return stack. 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 stack 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 return stack as temporary storage can make your code less complex.

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

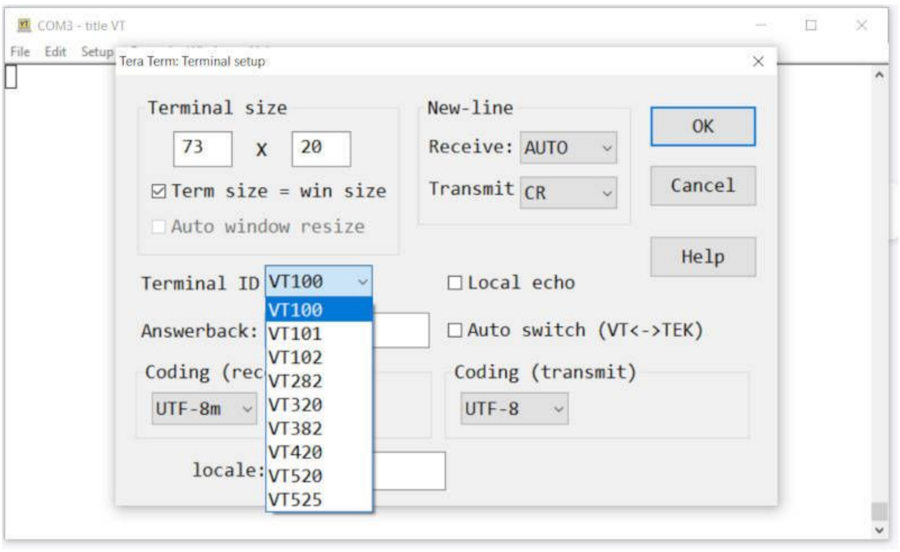
We can now access the memory allocated to `graphic-array` using the fetch and store words explained earlier. To calculate the address of the third byte assigned to `graphic-array` we can write `graphic-array 2 +`, remembering that the indices start at 0.

```
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:



Tera Term: Terminal setup

Terminal size
73 x 20
☑ Term size = win size
☐ Auto window resize

New-line
Receive: AUTO
Transmit CR

Terminal ID VT100
VT100
VT101
VT102
VT282
VT320
VT382
VT420
VT520
VT525

☐ Local echo
Answerback:
☐ Auto switch (VT<->TEK)
Coding (rec
UTF-8m
Coding (transmit)
UTF-8
locale:

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**.

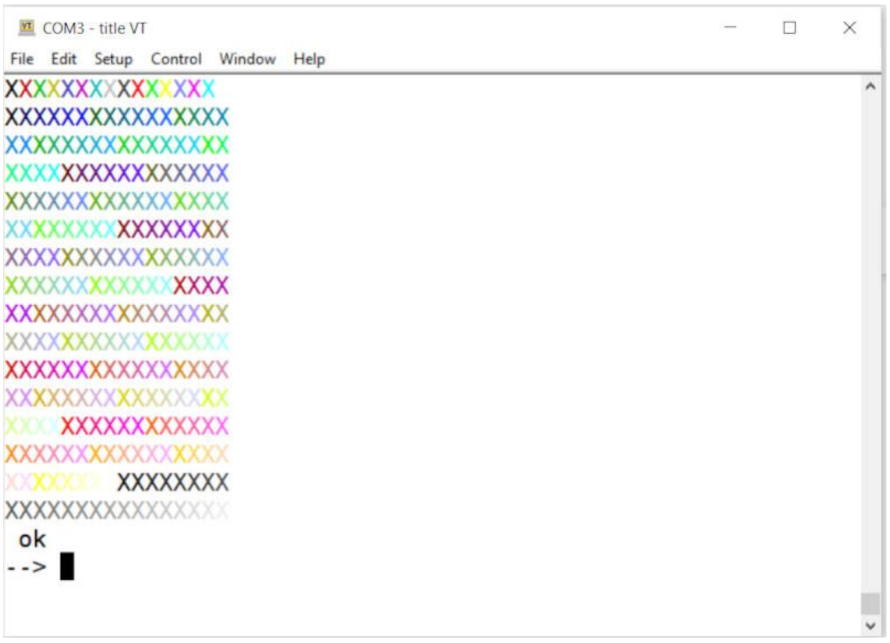
The word **page** clears the terminal screen and positions the cursor at the upper left corner of the screen.

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 :

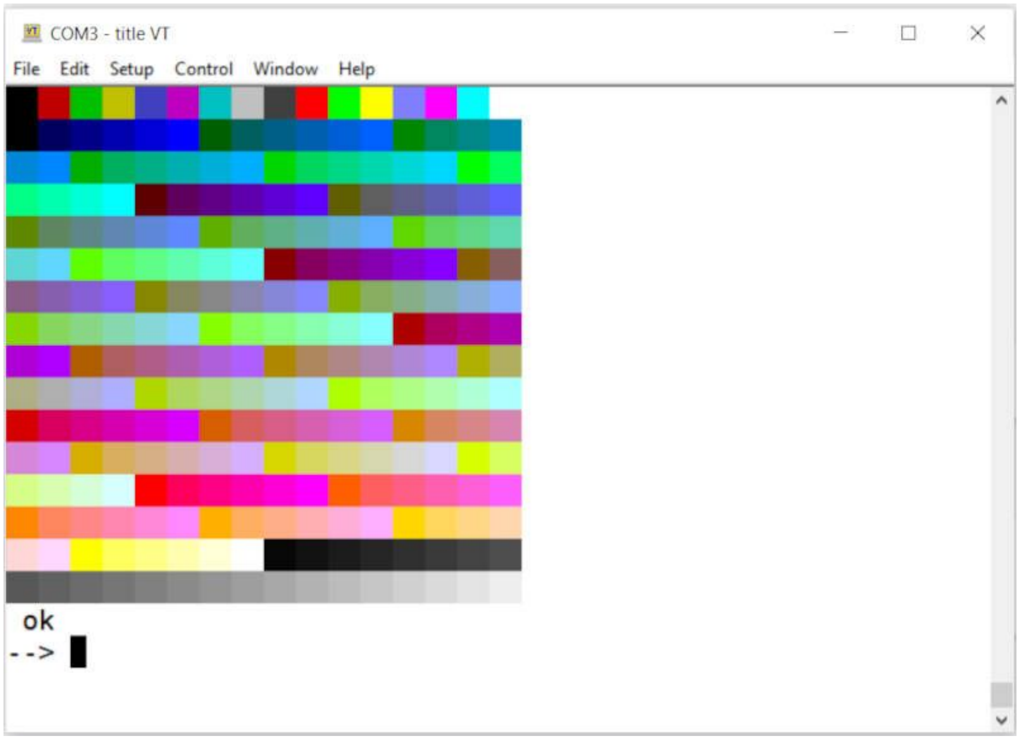


```
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
 ok
-->
```

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 :



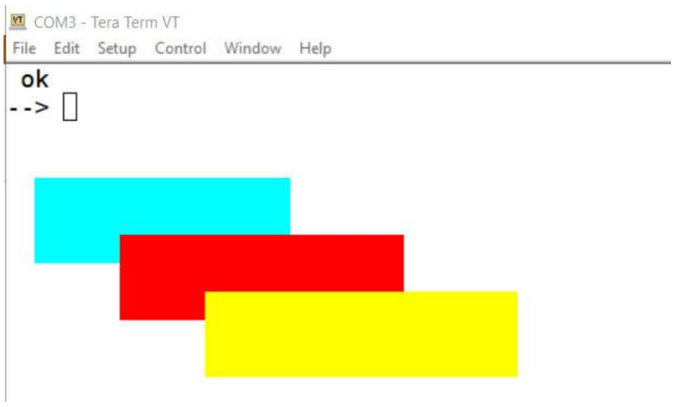
```
ok
-->
```

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 :



```
 ok
-->
```

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 ;
```

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 :

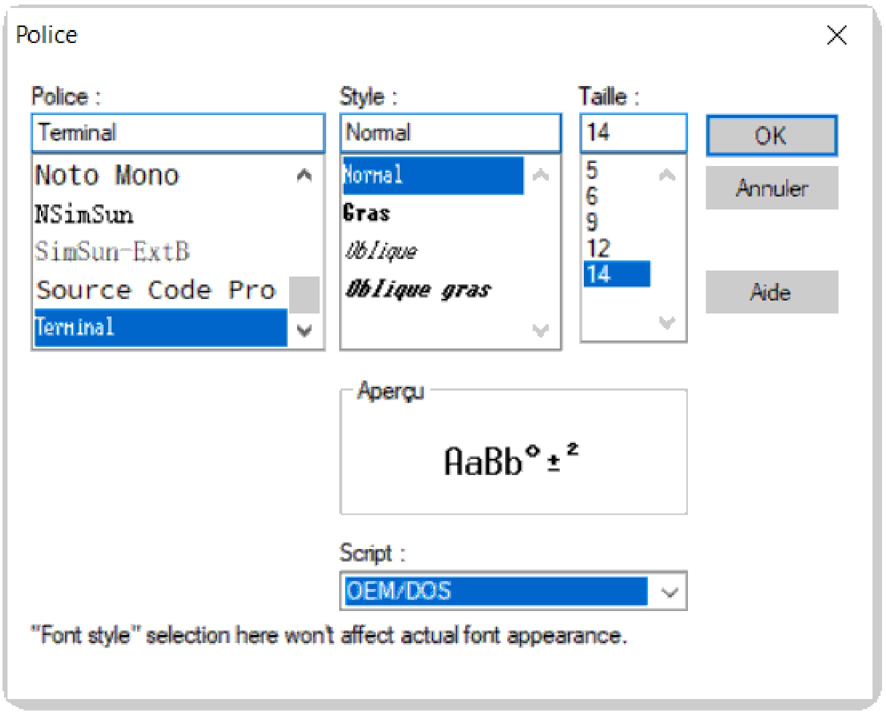
```
: xySCRaddr { x y -- addr }
    SCR_WIDTH y *
    x + mySCREEN +
  ;
: SCR@ ( x y -- c )
    xySCRaddr c@
  ;
: SCR! ( c x y -- )
    xySCRaddr c!
  ;
char X 15 5 SCR!     \ store char X at col 15 line 5
15 5 SCR@ emit       \ display X
```

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 :


Police

Police : Terminal

Style : Normal

Taille : 14

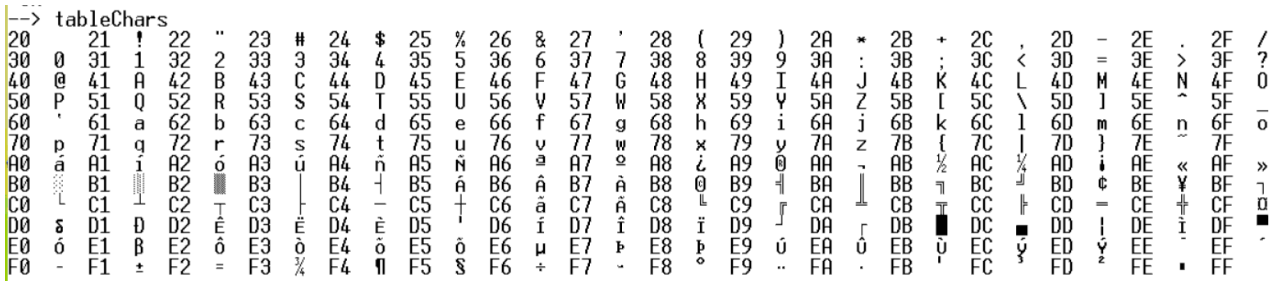
Script : OEM/DOS

"Font style" selection here won't affect actual font appearance.

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 :

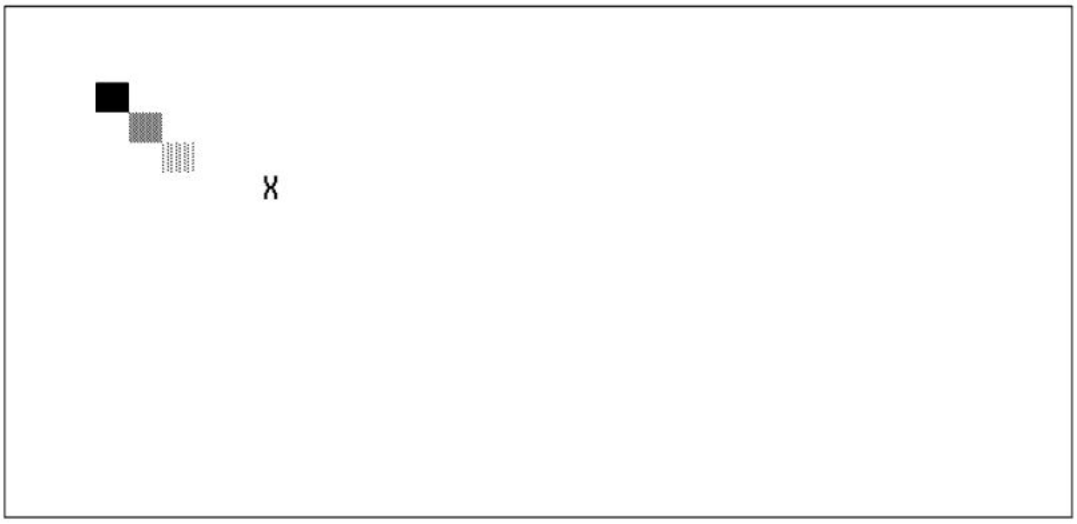
```
: nEmit ( c n -- )
    for
        aft dup emit then
    next
    drop
  ;
```

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
```

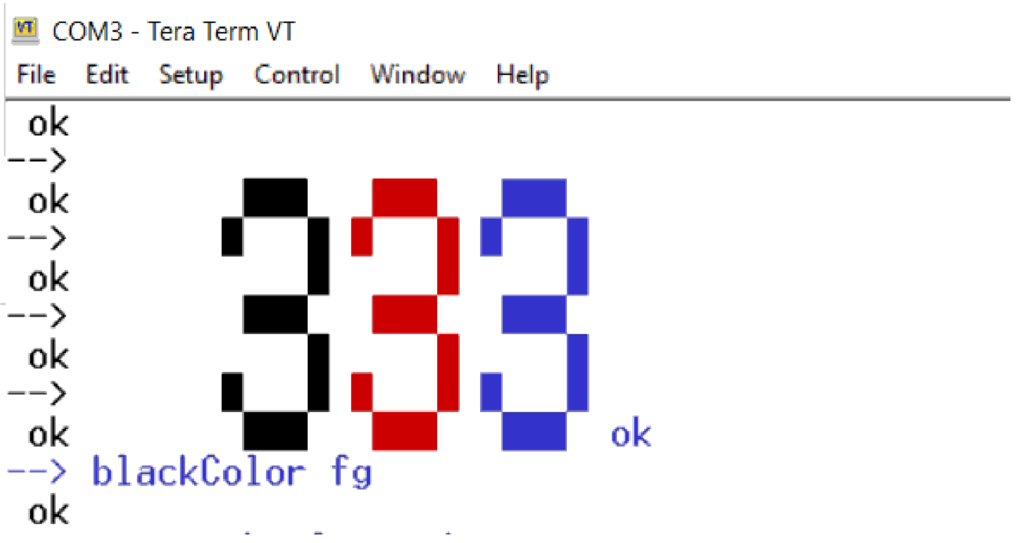
Here's how to define a 5 x 7 byte sprite :

```
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 :



```
 ok
-->
 ok
-->
 ok
-->
 ok
-->
 ok                                    ok
--> blackColor fg
 ok
```

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

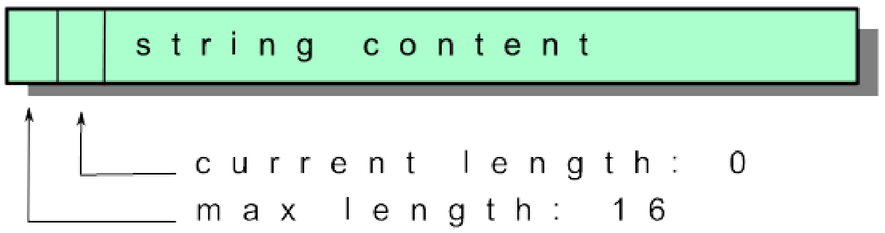
Alpha-numeric text variables do not exist natively in ESP32forth. Here is the first attempt to define the word **string** :

```
\ 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
```

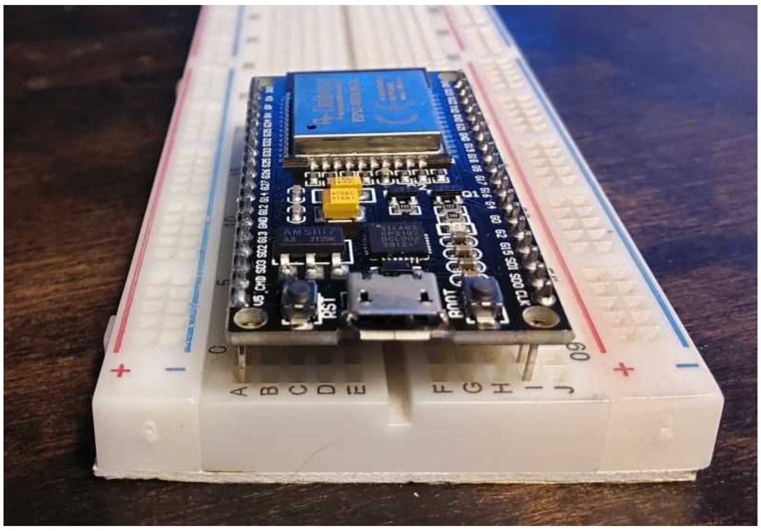
The search order, here, will start with the `SPIFFS vocabulary` , then `Serial` , then `ledc` and finally the `FORTH vocabulary` :

- 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 :

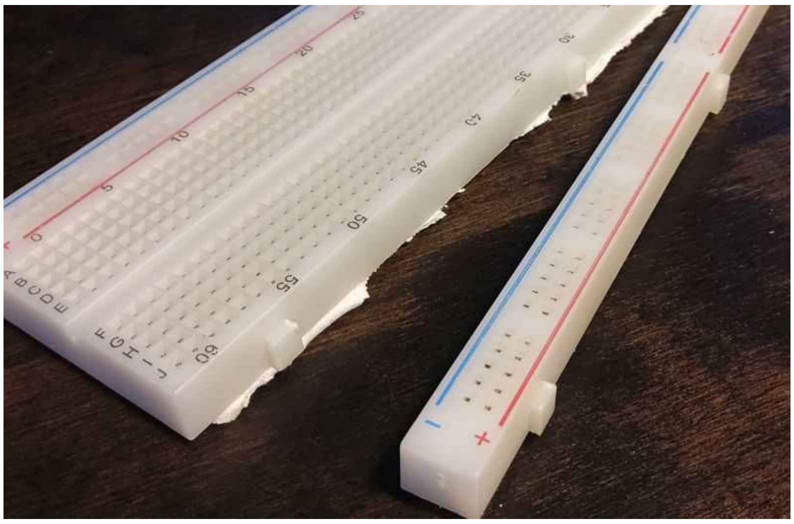


There is no breadboard specifically suited to ESP32 boards.

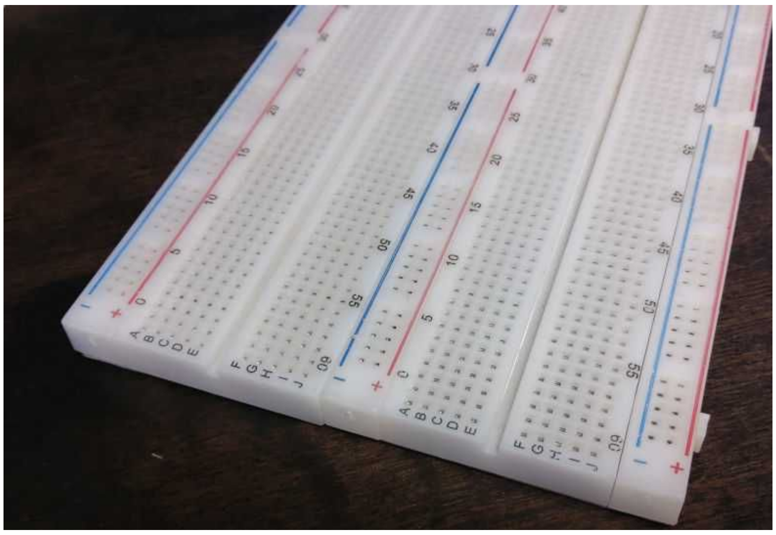
Build a breadboard suitable for the ESP32 board

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

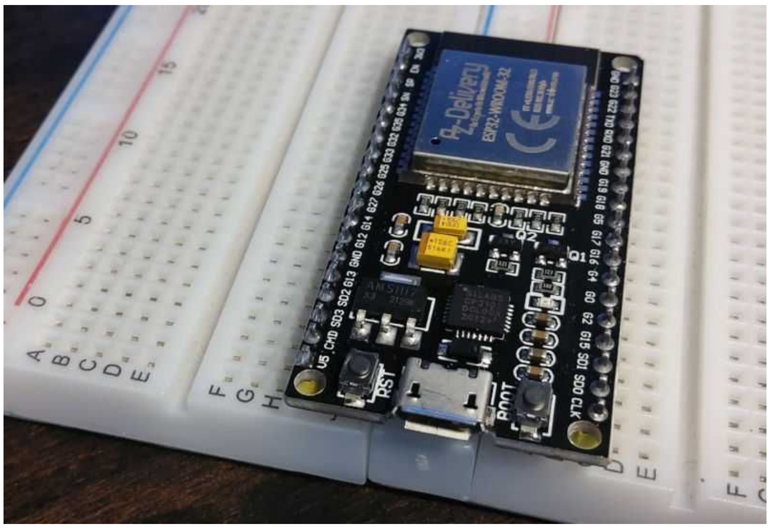
On one of breadboard, we will remove a power line. To do this, use a cutter and cut from below. You should be able to separate this power line like this :



We can then reassemble the entire breadboard 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.

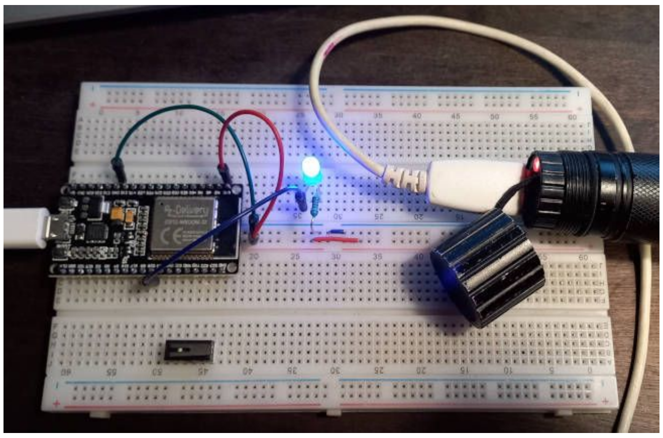
Alimenter la carte ESP32

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).

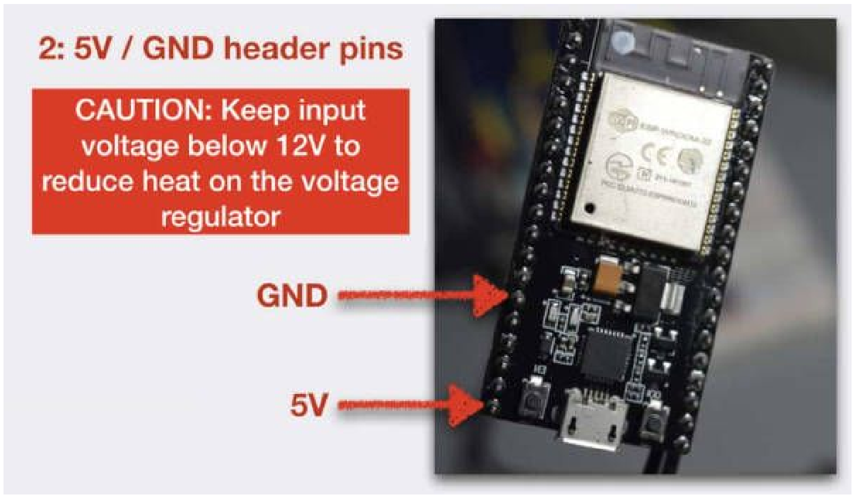
Ici, on alimente notre carte ESP32 avec une batterie de secours pour appareils mobiles.

Alimentation par le pin 5V

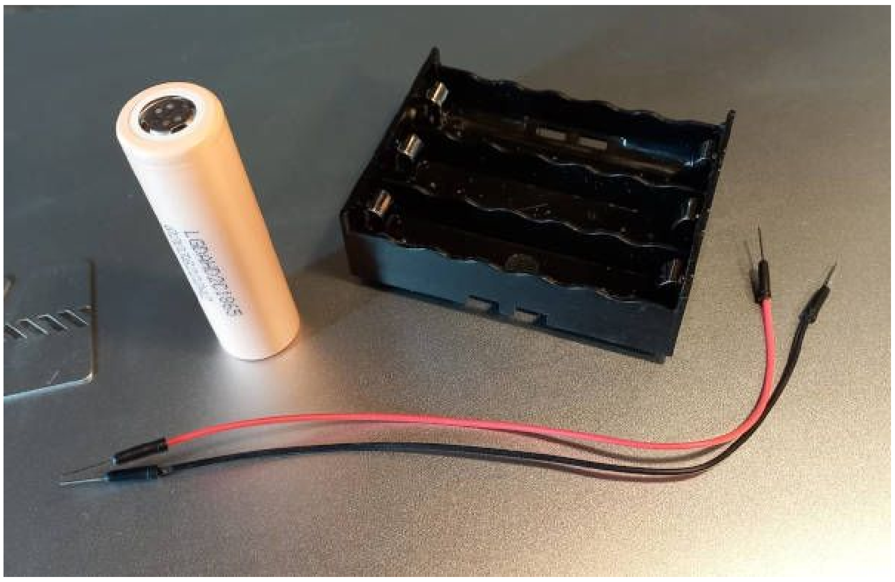
La deuxième option consiste à connecter une alimentation externe non régulée à la broche 5 V et à la masse. Tout ce qui se situe entre 5 et 12 volts devrait fonctionner.

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:


2: 5V / GND header pins
CAUTION: Keep input voltage below 12V to reduce heat on the voltage regulator
GND
5V

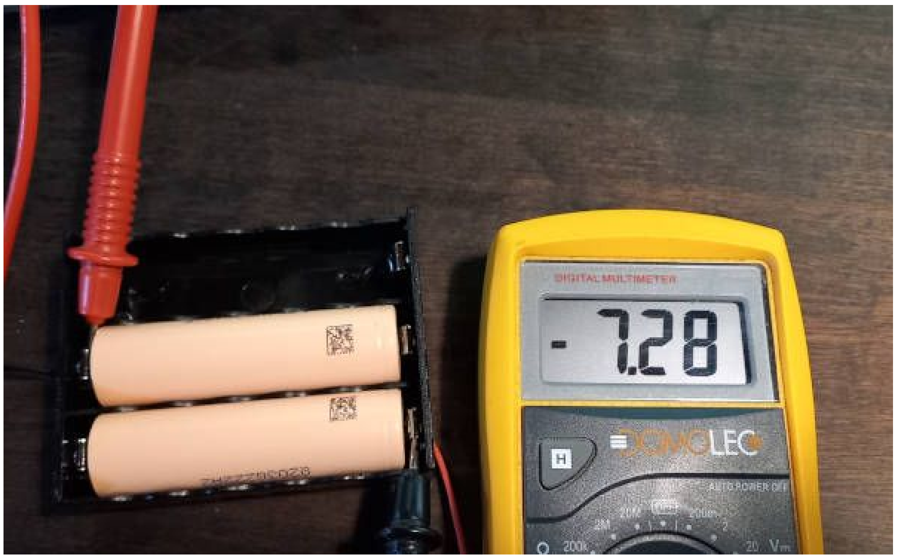
Pour exploiter l'alimentation 5V, il faut ce matériel:



- 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.

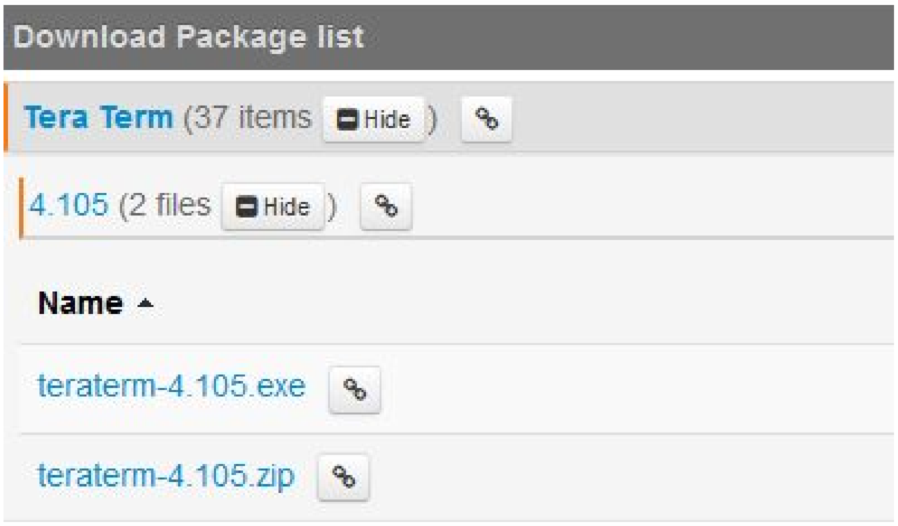
Install and use the Tera Term terminal on Windows

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:



Download Package list

Tera Term (37 items Hide)

4.105 (2 files Hide)

Name ▲

teraterm-4.105.exe

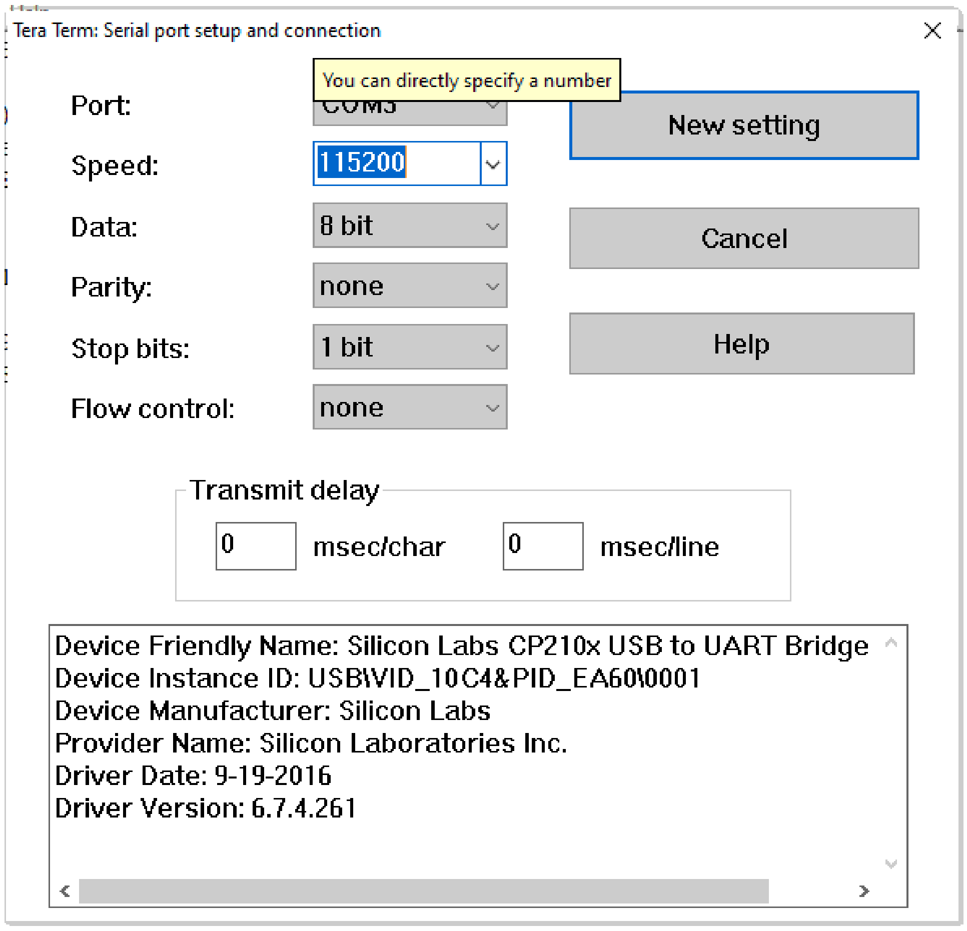
teraterm-4.105.zip

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


Tera Term: Serial port setup and connection
You can directly specify a number
New setting
Cancel
Help
Transmit delay

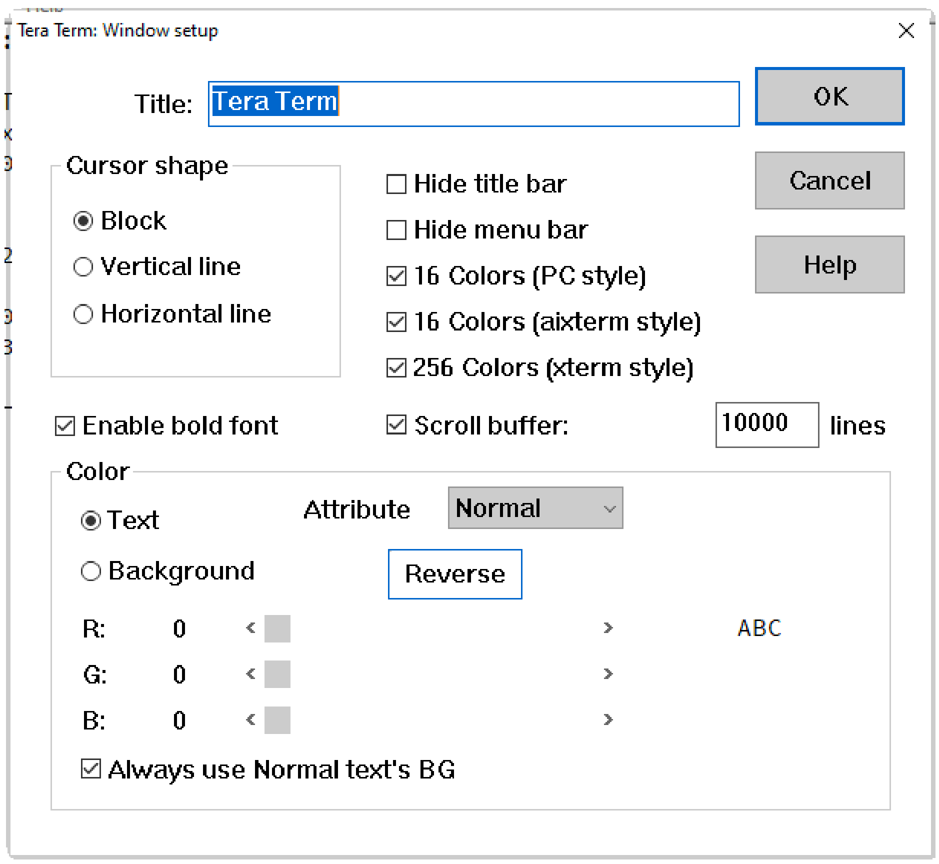
Port: COM3
Speed: 115200
Data: 8 bit
Parity: none
Stop bits: 1 bit
Flow control: none

0 msec/char 0 msec/line

```
Device Friendly Name: Silicon Labs CP210x USB to UART Bridge
Device Instance ID: USB\VID_10C4&PID_EA60\0001
Device Manufacturer: Silicon Labs
Provider Name: Silicon Laboratories Inc.
Driver Date: 9-19-2016
Driver Version: 6.7.4.261
```

For comfortable viewing:

- click on Configuration -> window


Tera Term: Window setup

Title: Tera Term

Cursor shape: [✓ Block] Vertical line Horizontal line

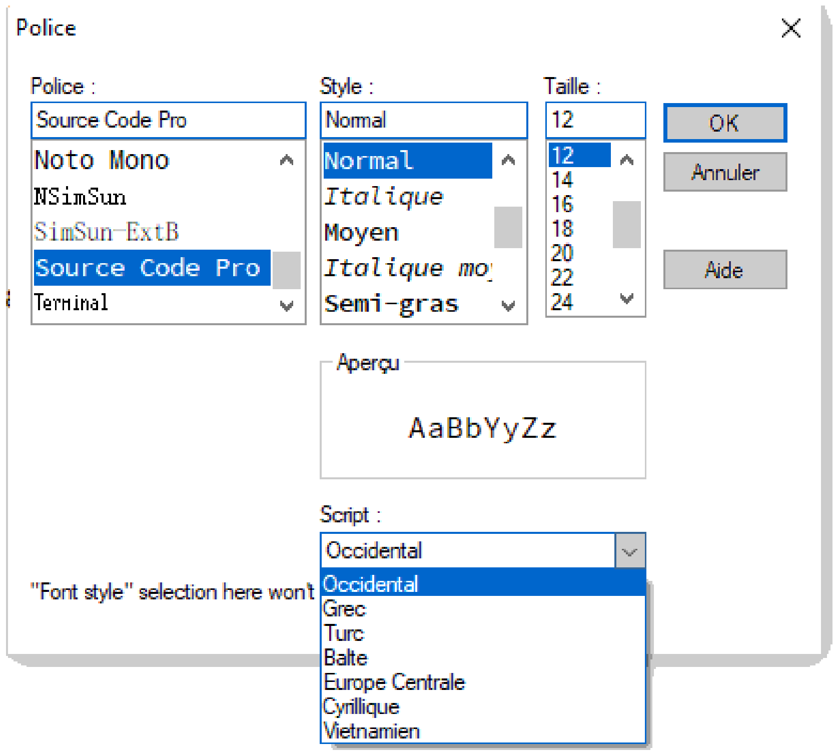
☐ Hide title bar
☐ Hide menu bar
☑ 16 Colors (PC style)
☑ 16 Colors (aixterm style)
☑ 256 Colors (xterm style)

☑ Enable bold font
☑ Scroll buffer: 10000 lines

Color: [✓ Text] Background
Attribute: Normal
Reverse
R: 0
G: 0
B: 0
ABC
☑ Always use Normal text's BG

For readable characters:

- click on Configuration -> font


Police

Police : Source Code Pro
Noto Mono
NSimSun
SimSun-ExtB
Source Code Pro
Terminal

Style : Normal
Normal
Italique
Moyen
Italique mo
Semi-gras

Taille : 12
12
14
16
18
20
22
24

Aperçu
AaBbYyZz

Script : Occidental
Occidental
Grec
Turc
Balte
Europe Centrale
Cyrillique
Vietnamien

"Font style" selection here wont

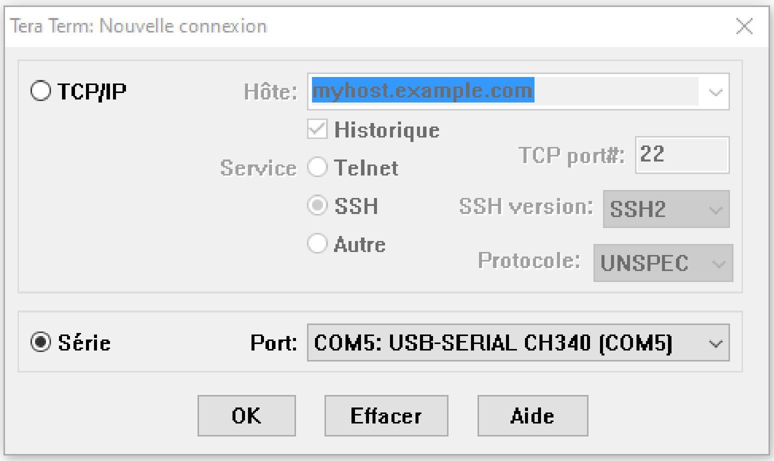
Using Tera Term

Once configured, close Tera Term.

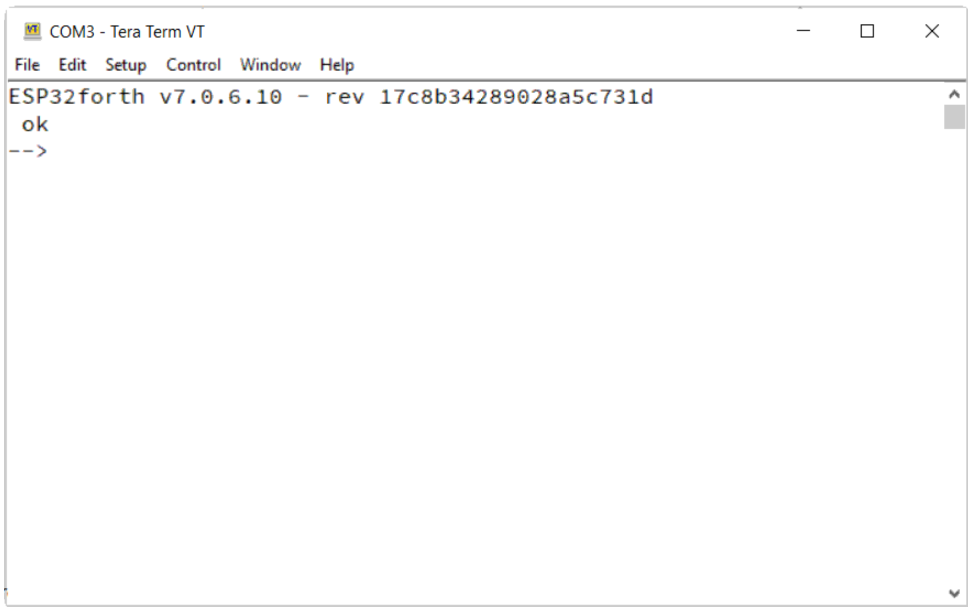
Connect your ESP32 board to an available USB port on your PC.

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

Select the serial port :



If everything went well, you should see this:



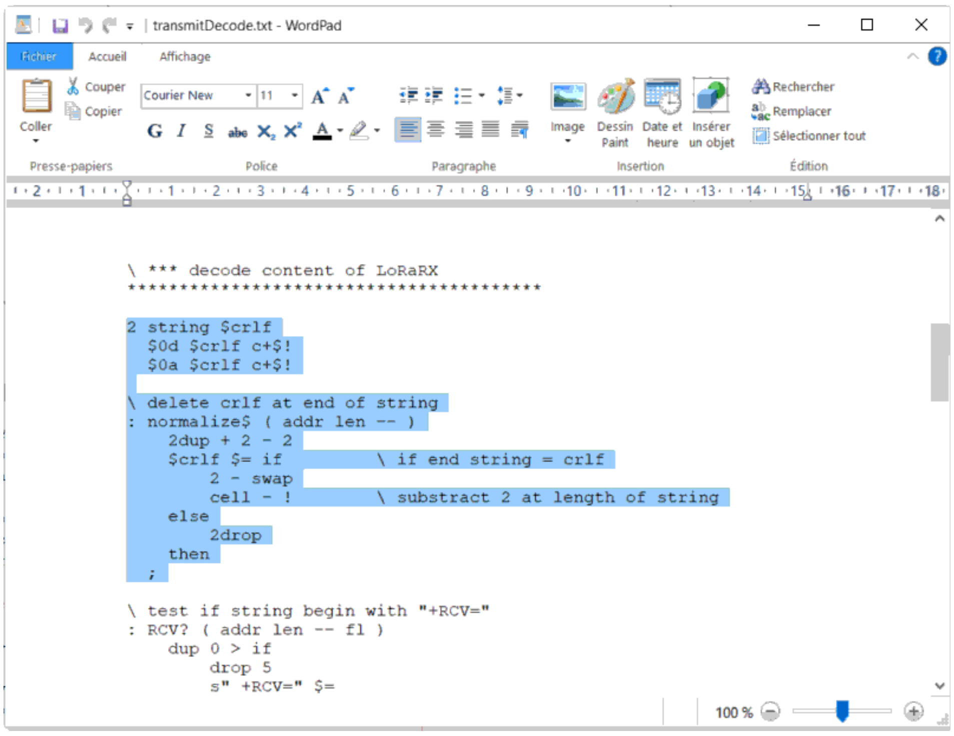
```
ESP32forth v7.0.6.10 - rev 17c8b34289028a5c731d
 ok
-->
```

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:



```
\ *** decode content of LoRaRX
****************************************

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

\ delete crlf at end of string
: normalize$ ( addr len -- )
    2dup + 2 - 2
    $crlf $= if             \ if end string = crlf
        2 - swap
        cell - !            \ substract 2 at length of string
    else
        2drop
    then
  ;

\ test if string begin with "+RCV="
: RCV? ( addr len -- fl )
    dup 0 > if
        drop 5
        s" +RCV=" $=
```

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:

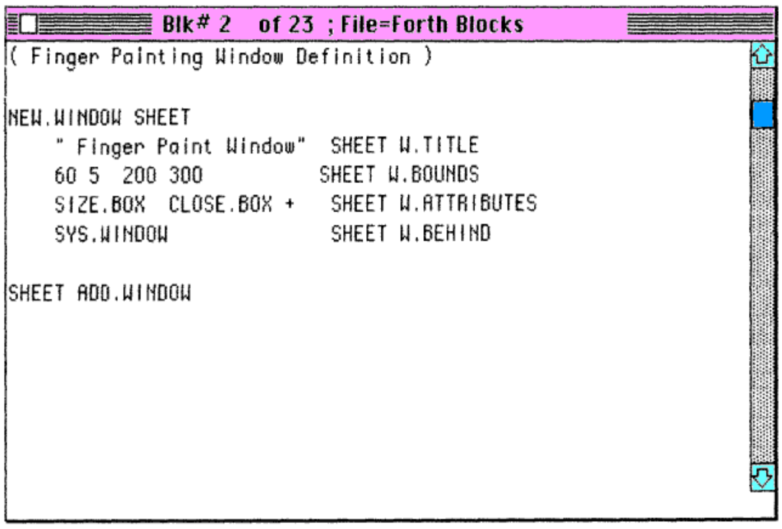
Simply validate by clicking OK and the code will be interpreted and/or compiled.

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

Management of source files by blocks

The blocks

Here a block on an old computer:



```
( Finger Painting Window Definition )

NEW.WINDOW SHEET
    " Finger Paint Window"  SHEET W.TITLE
    60 5  200 300          SHEET W.BOUNDS
    SIZE.BOX  CLOSE.BOX +   SHEET W.ATTRIBUTES
    SYS.WINDOW              SHEET W.BEHIND

SHEET ADD.WINDOW
```

A block is a storage space whose unit has 16 lines of 64 characters. The size of a block is therefore 16x64=1024 bytes. It's exactly the size of a kilobyte!

Open a block file

A file is already open by default when ESP32forth starts.

blocks.fb file .

If in doubt, run `default-use` .

To find out what's in this file, use the editor commands by first typing `editor` .

Here are our first commands to know to manage the content of blocks:

- `l` lists the contents of the current block
- `n` selects the next block
- `p` selects the previous block

ATTENTION: a block always has a number between 0 and n. If you end up with a negative block number, it throws an error.

Edit the contents of a block

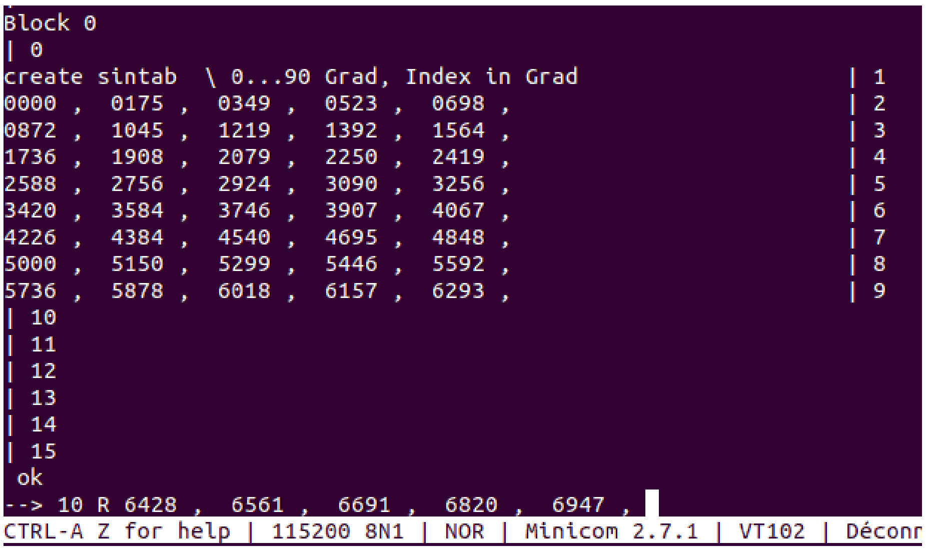
Now that we know how to select a particular block, let's see how to insert source code in FORTH language...

One strategy is to create a source file on your computer using a text editor. You will then just need to copy/paste your source code by line into the block files.

Here are the essential commands for managing the contents of a block:

- `wipe` empties the contents of the current block
- `d` deletes line n. The line number must be in the range 0..14. The following lines move upwards. Example: 3 D erases the contents of line 3 and brings up the contents of lines 4 to 15.
- `e` erases the contents of line n. The line number must be in the range 0..15. The other lines do not go up.
- `a` inserts a line n. The line number must be in the range 0..14. The lines located after the inserted line move back down. Example: 3 A test inserts test in line 3 and moves down the contents of lines 4 to 15.
- `r` replaces the contents of line n. Example: 3 R test replaces the contents of line 3 with test

Here is our block 0 currently being edited:



```
Block 0
| 0
create sintab  \ 0...90 Grad, Index in Grad                   | 1
0000 ,  0175 ,  0349 ,  0523 ,  0698 ,                        | 2
0872 ,  1045 ,  1219 ,  1392 ,  1564 ,                        | 3
1736 ,  1908 ,  2079 ,  2250 ,  2419 ,                        | 4
2588 ,  2756 ,  2924 ,  3090 ,  3256 ,                        | 5
3420 ,  3584 ,  3746 ,  3907 ,  4067 ,                        | 6
4226 ,  4384 ,  4540 ,  4695 ,  4848 ,                        | 7
5000 ,  5150 ,  5299 ,  5446 ,  5592 ,                        | 8
5736 ,  5878 ,  6018 ,  6157 ,  6293 ,                        | 9
| 10
| 11
| 12
| 13
| 14
| 15
 ok
--> 10 R 6428 ,  6561 ,  6691 ,  6820 ,  6947 ,
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Déconr
```

At the bottom of the screen, line `10 R 6428, 6561, .....` is being integrated into our block at line 10.

You notice that line 0 has no content. This generates an error when compiling the FORTH code. To fix this, simply type `0 R` followed by two spaces.

With a little practice, in a few minutes, you will have inserted your FORTH code into this block.

Do the same for the following blocks if necessary. When moving to the next block, you force the contents of the blocks to be saved by typing `flush` .

Compiling block contents

Before compiling the contents of a block file, we will check that their contents are well saved. For that:

- type `flush` , then unplug the ESP32 board;
- wait a few seconds and reconnect the ESP32 board;
- type `editor` and `l` . You must find your block 0 with the content that you edited.

To compile the content of your blocks, you have two words:

- `load` preceded by the number of the block whose content we want to execute and/or compile. To compile the contents of our block 0, we will execute `0 load` ;
- `thru` preceded by two block numbers will execute and/or compile the contents of the blocks as if we were executing a succession of `load words` . Example: `0 2 thru` executes and/or compiles the contents of blocks 0 to 2.

The speed of execution and/or compilation of block content is almost instantaneous.

Practical step-by-step example

We will see, with a practical example, how to insert source code in block 1. We take a code ready to be integrated into our block:

```
1 list
editor
 0 r \ tools for REGISTERS definitions and manipulations
 1 r : mclr { mask addr -- }    addr @  mask invert and addr !  ;
 2 r : mset { mask addr -- }    addr @  mask or addr !  ;
 3 r : mtst { mask addr -- x }  addr @  mask and ;
 4 r : defREG: \ define a register, similar as constant
 5 r     create ( addr1 -- <name> )  ,
 6 r     does>  ( -- regAddr )       @  ;
 7 r : .reg ( reg -- ) \ display reg content
 8 r     base @ >r binary @ <#
 9 r     4 for aft 8 for aft # then next
10 r     bl hold  then next  #>
11 r     cr space ." 33222222 22221111 11111100 00000000"
12 r     cr space ." 10987654 32109876 54321098 76543210"
13 r     cr type  r> base !  ;
14 r : defMASK:  create ( mask0 position -- )    lshift ,
15 r     does> ( -- mask1 )              @  ;
save-buffers
```
Simply copy/paste parts of the code above and run this code through ESP32 Forth:

- `1 list` to select and see what block 1 contains
- `editor` to select vocabulary `editor`
- copy the lines `n r....` in packs of three and run them
- `save-buffers` hard-saves code in block file

Turn off the ESP32 board. Restart it. If you type `1 list` you should see the code edited and saved.

To compile this code, simply type `1 load` .

Conclusion

The available file space for ESP32forth is close to 1.8MB. You can therefore worry-free manage hundreds of blocks for source files in FORTH language. It is recommended to install source codes of stable code parts. Thus, during the program development phase, it will be much easier to integrate into your code in the development phase:

```
2 5 thru \ integrate pwm commands for motors
```
instead of systematically reloading this code via serial line or WiFi.

The other advantage of blocks is to allow the on-site embedding of parameters, data tables, etc. which can then be used by your programs.

Editing source files with VISUAL Editor

Edit a FORTH source file

To edit a FORTH source file with ESP32forth, we will use the visual editor.

To edit a `dump.fs file` , proceed like this from the terminal connected to an ESP32 card containing ESP32forth:

```
visual edit /spiffs/dump.fs
```

The full `DUMP code` is available here:

https://github.com/MPETREMANN11/ESP32forth/blob/main/tools/dumpTool.txt

The word `edit` is followed by the directory where the source files are stored:

- if the file does not exist, it is created;
- if the file exists, it is retrieved in the editor.

Note the name of the file you created.

fs as the file extension , for **F** orth **S** ource.

Editing the FORTH code

In the editor, move the cursor with the left-right-up-down arrows available on the keyboard.

The terminal refreshes the display each time the cursor is moved or the source code is modified.

To exit the editor :

- CTRL-S : saves the contents of the file currently being edited
- CTRL-X : exits editing:
    - N: without saving file changes
    - Y: with saving of changes

Compiling file contents

Compiling the contents of our **dump.fs file** is done like this:

```
include /spiffs/dump.fs
```

Compiling is much faster than via the terminal.

The source files embedded in the ESP32 card with ESP32forth are persistent. After turning off the power and reconnecting the ESP32 card, the saved file remains available immediately.

You can define as many files as necessary.

It is therefore easy to integrate into the ESP32 card a collection of tools and routines from which you can draw as needed.

The SPIFFS file system

ESP32Forth contains a rudimentary file system on internal Flash memory. The files are accessible via a serial interface called SPIFFS for Serial Peripheral Interface Flash File System.

Even though the SPIFFS file system is simple, it considerably increases the flexibility of your developments with ESP32Forth:

- manage configuration files
- integrate software extensions accessible on request
- modularize developments into reusable functional modules

And many other uses that we will let you discover...

Access to the SPIFFS file system

To compile the contents of a source file edited by visual edit, type:

```
include /spiffs/dumpTool.fs
```

The word `include` must always be used from the terminal.

To see the list of SPIFFS files, use the word `ls` :

```
ls /spiffs/
\ displays:
\ dumpTool.fs
```

Here, the **dumpTool.fs file** has been saved. For SPIFFS, file extensions are irrelevant. File names must not contain space characters or the / character.

Let's edit and save a new **myApp.fs file** with `visual editor` . Let's run `ls again` :

```
ls /spiffs/
\ displays:
\ dumpTool.fs
\ myApp.fs
```

The SPIFFS file system does not manage subfolders like on a Linux computer. To create a pseudo directory, simply indicate it when creating a new file. For example, let's edit the **other/myTest.fs file** . Once edited and saved, let's run `ls` :

```
ls /spiffs/
```

```
\ displays:
\ dumpTool.fs
\ myApp.fs
\ other/myTest.fs
```

If you want to view only the files in this **other** pseudo directory , you must follow **/spiffs/** with the name of this pseudo directory :

```
ls /spiffs/other
\ displays:
\ myTest.fs
```

There is no option to filter file names or pseudo directories.

Handling files

To completely delete a file, use the word `rm` followed by the name of the file to be deleted :

```
rm /spiffs/other/myTest.fs
ls /spiffs/
\ poster:
\dumpTool.fs
\myApp.fs
```

To rename a file, use the word `mv` :

```
mv /spiffs/myApp.fs /spiffs/main.fs
ls /spiffs/
\ displays:
\ dumpTool.fs
\ main.fs
```

To copy a file, use the word `cp` :

```
cp /spiffs/main.fs /spiffs/mainTest.fs
ls /spiffs/
\ displays:
\ dumpTool.fs
\ main.fs
\ mainTest.fs
```

To see the contents of a file, use the word `cat` :

```
cat /spiffs/dumpTool.fs
\ displays contents of dumpTool.fs
```

To save the contents of a string to a file, act in two phases :

- create a new file with `touch`
- save string contents with `dump-file`

```
touch /spiffs/mTest,fs  \ creates new mtest,fs file
ls /spiffs/             \ displays:
\ dumpTool.fs
\ main.fs
\ mainTest.fs
\ mTests

\ save string "Insert my text into mTest" in mTest
r| ." Insert my text into mTest" |   s" /spiffs/mTest" dump-file

include /spiffs/mTest   \ displays: Insert my text in mTest
```

Organize and compile your files on the ESP32 card

We will see how to manage files for an application being developed on an ESP32 board with ESP32forth installed on it.

It is agreed that all files used are in ASCII text format.

The following explanations are given as advice only. They come from a certain experience and aim to facilitate the development of large applications with ESP32forth.

Editing and transmitting source files

All the source files for your project are on your computer. It is advisable to have a subfolder dedicated to this project. For example, you are working on an SSD1306 OLED display. So you create a directory named SSD1306.

Regarding file name extensions, we recommend using the **fs** extension .

Editing files on a computer is carried out with any text file editor.

In these source files, do not use any characters not included in the ASCII code characters. Some extended codes can disrupt program compilation.

These source files will then be copied or transferred to the ESP32 card via the serial link and a terminal type program :

- by copy/pasted using visual on ESP32forth, to be reserved for small files ;
- with a specific procedure which will be detailed later for important files.

Organize your files

In the following, all our files will have the extension **fs** .

Let's start from our SSD1306 directory on our computer.

The first file we will create in this directory will be the **main.fs** file . This file will contain the calls to load all the other files of our application under development.

Example of content of our **main.fs** file :

```
\ OLED SSD1306 128x32 dev and display tests
s" /SPIFFS/config.fs" included
```

In development phase, the contents of this **main.fs file** will be loaded manually by running `include` like this :

```
include /spiffs/main.fs
```

This causes the contents of our **main.fs file to be executed** . Loading of other files will be executed from this **main.fs file** . Here we load the **config.fs file** of which here is an extract :

```
\ **************************************************
*******************
\ Configuration for SSD1306 128x32 OLED display
\ **************************************************
*******************

\ for SSD1306_128_32
    128 constant SSD1306_LCDWIDTH
     32 constant SSD1306_LCDHEIGHT
```

config.fs file we will put all the constant values and various parameters used by the other files.

Our next file will be **SSD10306commands.fs** . Here's how to load its contents from **main.fs** :

```
\ OLED SSD1306 128x32 dev and display tests
s" /spiffs/config.fs" included
s" /spiffs/SSD10306commands.fs" included
```

The contents of the **SSD10306commands.fs** file are almost 230 lines of code. It is not possible to copy the contents of this file line by line into the ESP32forth visual editor. here is a method to copy and save the contents of this large file on ESP32forth in one go.

Transfer a large file to ESP32forth

To enable this large file transmission, compile this code in ESP32forth:

```
: noType
    2drop ;

visual
```

```
: xEdit
    ['] noType is type
    edit
    ['] default-type is type
  ;
```
This very short Forth code allows you to transfer a very long FORTH program from the editor on PC to a file on the ESP32 card :

- compile with ESP32forth, the FORTH code, here the words noType and xEdit
- open the program on your PC to transfer to a file on the ESP32 card
- add the line at the top of the program allowing this fast transfer **`xEdit /spiffs/SSD10306commands.fs`**
- copy all the code of your edited program to your PC,
- pass into the terminal connected to ESP32forth
- copy your code

If all goes well, nothing should appear on the terminal screen. Wait a few seconds.

Next, type: CTRL-X, then press "Y"

You should regain control.

Check the presence of your new file: **`ls /spiffs/`**

You can now compile the contents of your new file.

Conclusion

Files saved in the ESP32forth SPIFFS file system are permanently available.

If you take the ESP32 board out of service and then plug it back in, the files will be available immediately.

The content of the files can be modified in situ with **`visual edit`** .

This convenience will make developments much faster and easier.

Managing a traffic light with ESP32

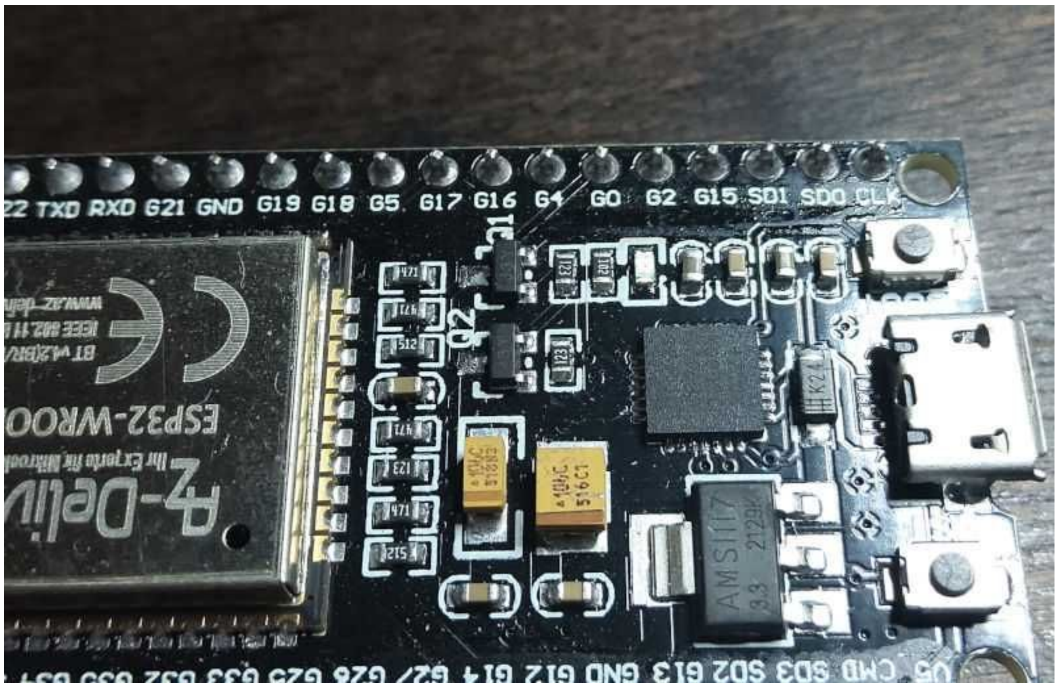
GPIO ports on the ESP32 board

GPIO ports (General Purpose Input/Output) are input-output ports widely used in the world of microcontrollers.

The ESP32 board comes with 48 pins having multiple functions. Not all pins are used on ESP32 development boards, and some pins cannot be used.

There are many questions about how to use ESP32 GPIOs. Which connectors should you use? Which connectors should you avoid using in your projects?

If we look under a magnifying glass at an ESP32 card, we see this:


22 TXD RXD G21 GND G19 G18 G5 G17 G16 G4 G0 G2 G15 SD1 SD0 CLK
ESP32-WROOM
AMS1117

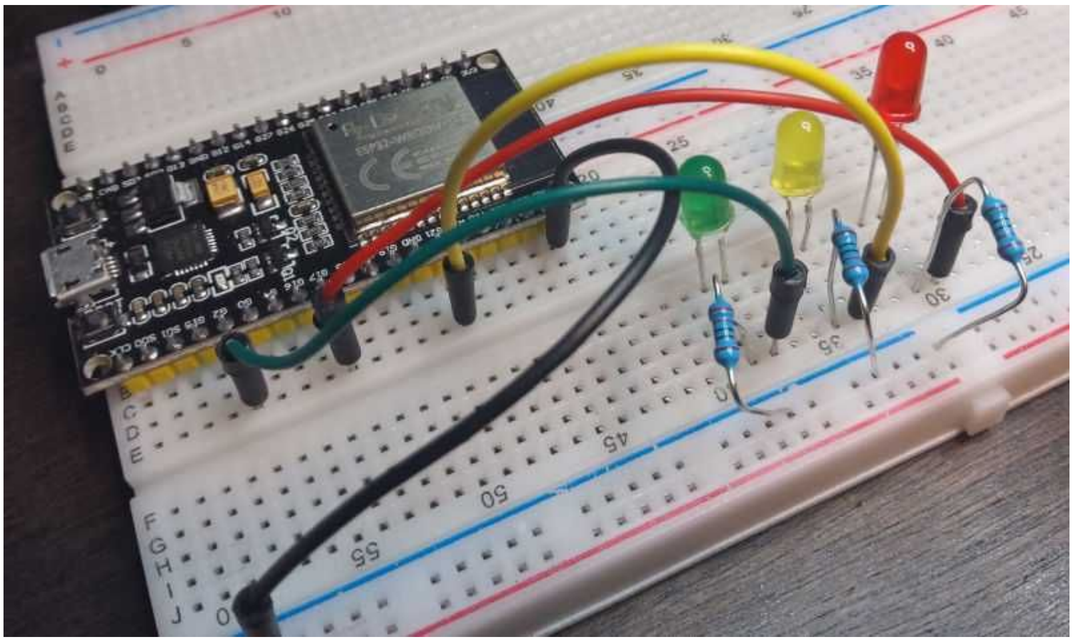
Each connector is identified by a series of letters and numbers, here from left to right in our photo: G22 TXD RXD G21 GND G19 G18, etc...

The connectors that interest us for this handling are prefixed by the letter G followed by one or two numbers. For example, G2 corresponds to GPIO 2.

Defining and operating a GPIO connector in output mode is quite simple.

Mounting the LEDs

The assembly is quite simple and only one photo is enough:



- Green LED connected to G2 - green wire
- Yellow LED connected to G21 - yellow wire
- Red LED connected to G17 - red wire
- black wire connected to GND

Our code uses the word **include** followed by the file to load.

We define our LEDs with **defPin:**

```
DEFINED? defPIN: invert
    [if] include /spiffs/defpin.txt [then]
\ Use:
\ numGPIO defPIN: PD7  ( define portD pin #7)

 2 defPIN: ledGREEN
21 defPIN: ledYELLOW
17 defPIN: ledRED

: LEDinit
    ledGREEN    output pinMode
    ledYELLOW   output pinMode
    ledRED      output pinMode
    ;
```

Many programmers have the bad habit of naming connectors by their number. Example :

```
17 defPin: pin17
```

Or

```
17 defPin: GPIO17.
```

To be effective, you must name the connectors by their function. Here we define the `ledRED` or `ledGREEN connectors` .

For what? Because the day you need to add accessories and release for example the G21 connector, simply redefine `21 defPIN: ledYELLOW` with the new connector number. The rest of the code will be unchanged and usable.

Management of traffic lights

Here is the part of code that controls our LEDs in our traffic light simulation:

```
\ traficLights execute one light cycle
: trafficLights ( ---)
    high ledGREEN   pin     3000 ms    low ledGREEN   pin
    high ledYELLOW  pin      800 ms    low ledYELLOW  pin
    high ledRED     pin     3000 ms    low ledRED     pin
    ;

\ classic traffic lights loop
: lightsLoop ( ---)
    LEDinit
    begin
        trafficLights
    key? until
    ;

\ german trafic light style
: Dtraffic ( ---)
    high ledGREEN   pin    3000 ms    low ledGREEN   pin
    high ledYELLOW  pin     800 ms    low ledYELLOW  pin
    high ledRED     pin    3000 ms
    ledYELLOW  high    800 ms
    \ simultaneous red and yellow ON
    high ledRED     pin  \ simultaneous red and yellow OFF
    high ledYELLOW  pin
    ;

\ german traffic lights loop
: DlightsLoop ( ---)
    LEDinit
    begin
        Dtraffic
    key? until
    ;
```

Conclusion

This traffic light management program could perfectly have been written in C language. But the advantage of the FORTH language is that it gives control, via the terminal, to analyze, debug and modify functions very quickly (in FORTH we say words).

Managing traffic lights is an easy exercise in C language. But when the programs become a little more complex, the compilation and upload process quickly becomes tedious.

Simply act via the terminal and simply copy/paste any fragment of FORTH language code for it to be compiled and/or executed.

If you are using a terminal program to communicate with the ESP32 board, simply type `DlightsLoop` or `lightsLoop` to test how the program works. These words use a conditional loop. Simply press a key on the keyboard and the word stops playing at the end of the loop.

Hardware interrupts with ESP32forth

Interruptions

When we want to manage external events, a push button for example, we have two solutions :

- test the state of the button as regularly as possible, through a loop. We will act according to the state of this button.
- use an interrupt. We assign the execution code to an interrupt attached to a pin. The button is connected to this pin and the state change will execute this word.

The interrupt solution is the most elegant. It allows you to relieve the main program by avoiding monitoring the button in a loop.

In his ESP32forth documentation, Brad NELSON gives a simple example of interrupt handling :

```
17 input pinMode
: test ." pinvalue: " 17 digitalRead . cr ;
' test 17 pinchange
```

Except that this example, as written, has a good chance of not working. We will see why and provide the elements to make it work.

Mounting a push button

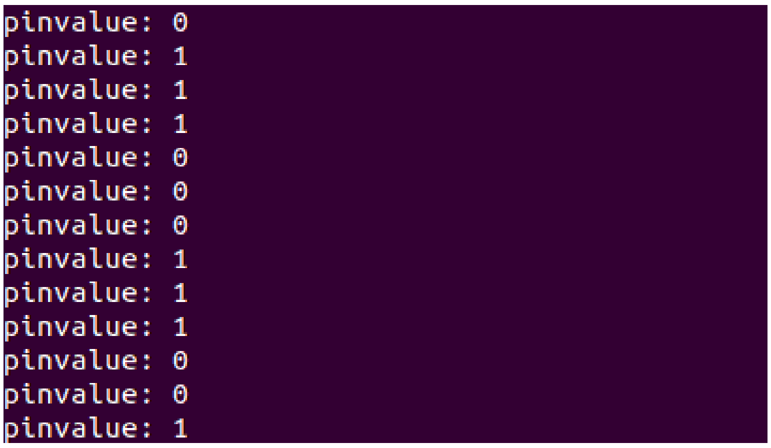
The button is connected as an input to the 3.3V power supply of the ESP32 board.

The push button output is connected to the GPIO17 pin. That's all.

For Brad NELSON's example to be functional, you must select the `interrupts vocabulary` before configuring the interrupt using `pinchange` . Along the way, we will define the `button constant` :

```
17 constant button
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
  ;
interrupts
' test button pinchange
forth
```

It works, but there is an unexpected effect which causes the interrupt to be triggered unexpectedly :



```
pinvalue: 0
pinvalue: 1
pinvalue: 1
pinvalue: 1
pinvalue: 0
pinvalue: 0
pinvalue: 0
pinvalue: 1
pinvalue: 1
pinvalue: 1
pinvalue: 0
pinvalue: 0
pinvalue: 1
```

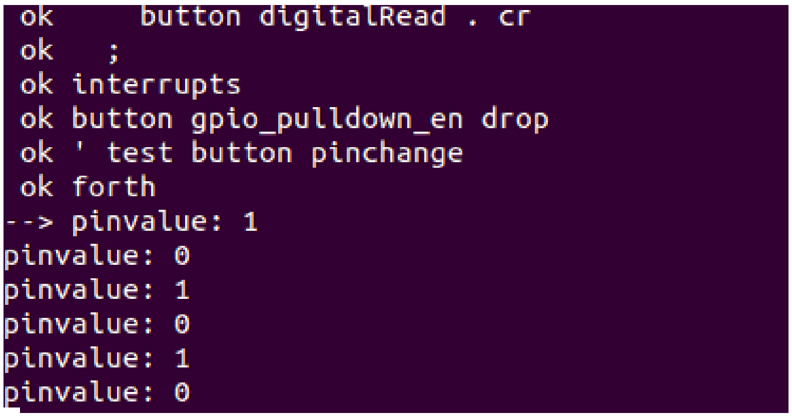
The hardware solution would consist of putting a high value resistor at the button output and connected to GND.

Software consolidation of the interrupt

In the ESP32 card, you can activate a resistor on any GPIO pin. This activation is carried out by the word `gpio_pulldown_en` . This word accepts as a parameter the GPIO pin number whose resistance must be activated. In return this word returns 0 if the action was successful, an error code otherwise :

```
17 constant button
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
  ;
interrupts
button gpio_pulldown_en drop
' test button pinchange
forth
```

The result of executing the interrupt is significantly better :



```
 ok     button digitalRead . cr
 ok   ;
 ok interrupts
 ok button gpio_pulldown_en drop
 ok ' test button pinchange
 ok forth
--> pinvalue: 1
pinvalue: 0
pinvalue: 1
pinvalue: 0
pinvalue: 1
pinvalue: 0
```

At each change of state, we have an interruption. On the screenshot above, each state change displays `pinvalue: 1` then `pinvalue: 0` .

It is possible to take into account an interruption on the rising edge alone. This is possible by indicating :

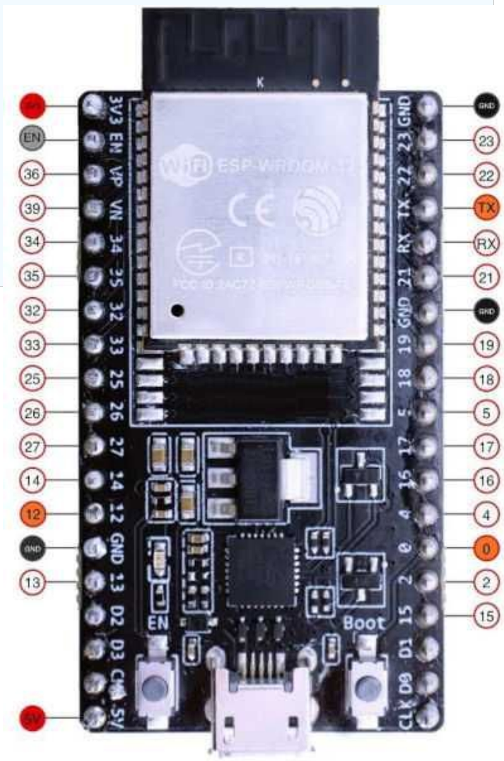
```
button GPIO_INTR_POSEDGE gpio_set_intr_type drop
```

The word `gpio_set_intr_type` accepts these parameters:

- `GPIO_INTR_ANYEDGE` to manage rising or falling edge interrupts
- `GPIO_INTR_NEGEDGE` to handle interrupts on falling edge only
- `GPIO_INTR_POSEDGE` to manage interrupts on rising edge only
- `GPIO_INTR_DISABLE` to disable interrupts

Complete FORTH code with rising edge detection:

```
17 constant button
0 constant GPIO_PULLUP_ONLY
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
  ;
interrupts
button gpio_pulldown_en drop
button GPIO_INTR_POSEDGE
gpio_set_intr_type drop
' test button pinchange
forth
```



Further information

For ESP32, all GPIO pins can be used as interrupt except GPIO6 to GPIO11.

Do not use pins colored orange or red. Your program might behave unexpectedly when using these.

Using the KY-040 rotary encoder

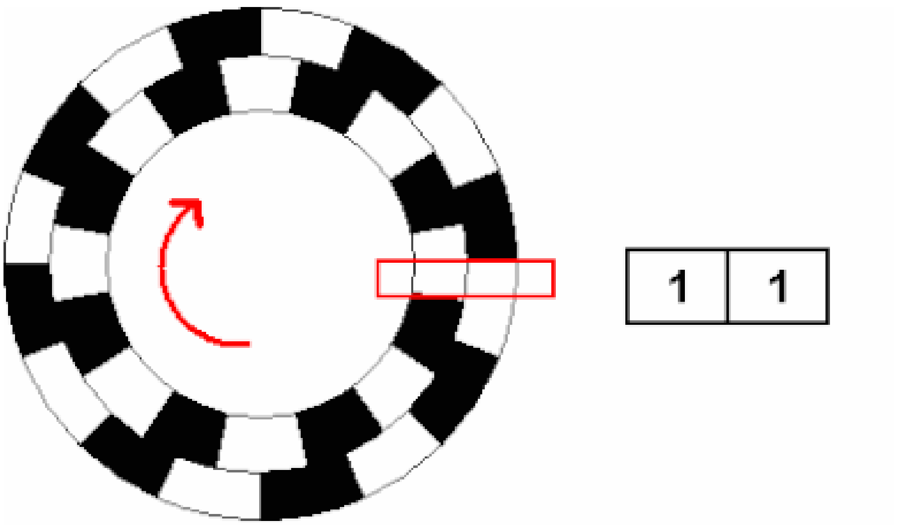
Encoder Overview

To vary a signal, we have several solutions:

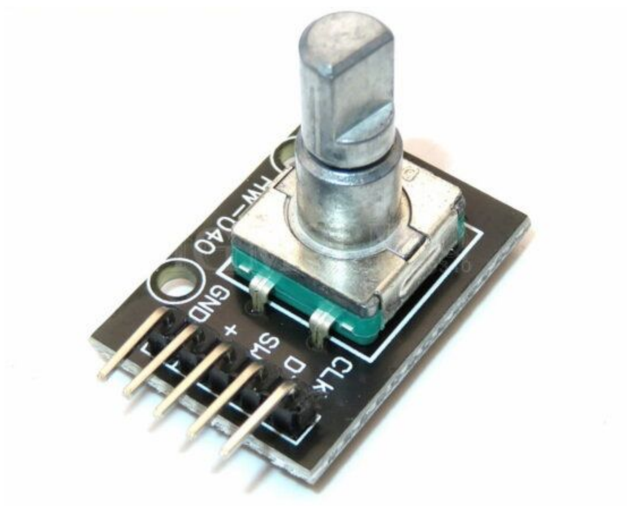
- a variable resistor in a potentiometer
- two buttons managing the variation by software
- a rotary encoder

The rotary encoder is an interesting solution. It can be made to act as a potentiometer, with the advantage of not having a start and end stop.

Its principle is very simple. Here are the signals emitted by our rotary encoder :



Here is our encoder :



Internal functioning diagram :

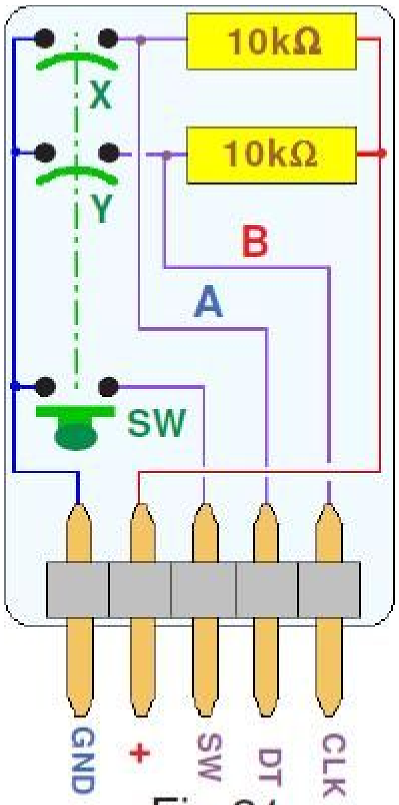


Fig.21

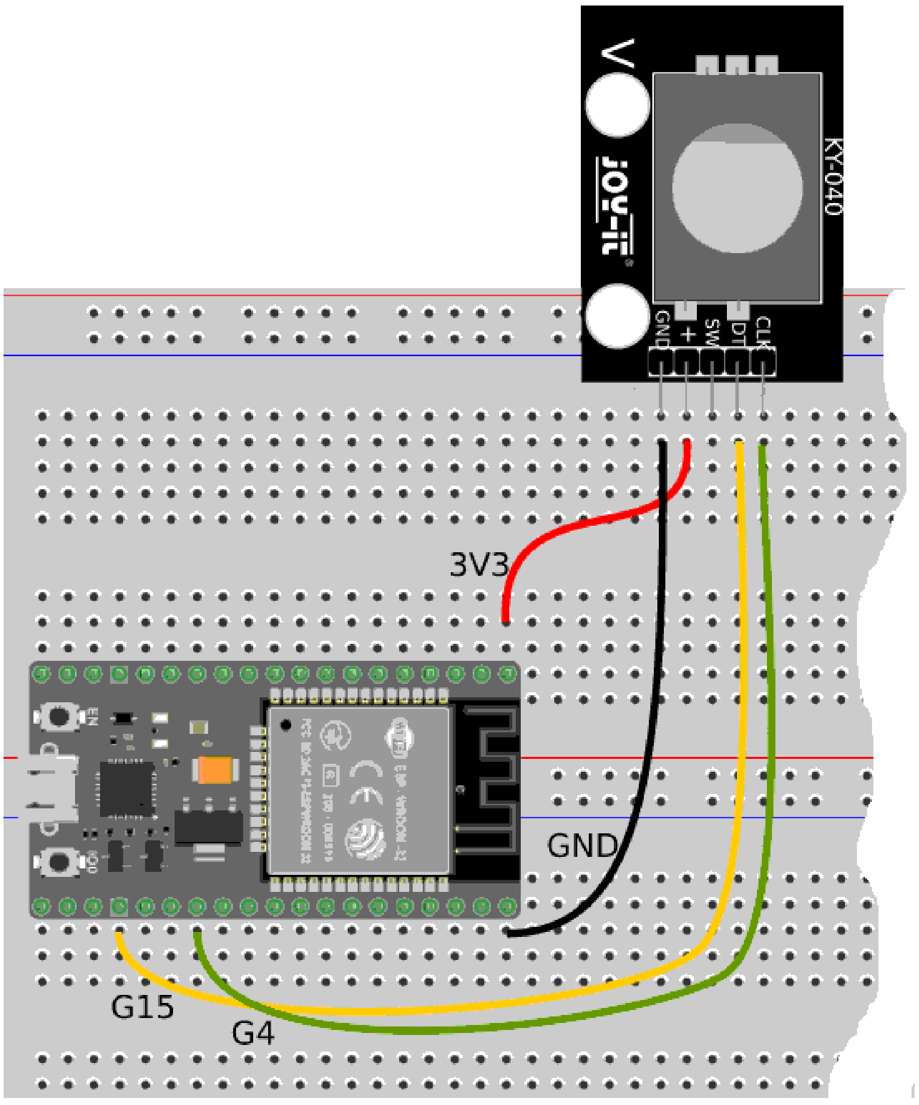
According to this diagram, two terminals interest us :

- A (DT) -> switch
- B (CLK) -> switch Y

This encoder can be powered with 5V or 3.3V. This suits us, because the ESP32 card has a 3.3V output.

Mounting the encoder on the breadboard

Wiring our encoder to the ESP32 board only requires 4 wires :

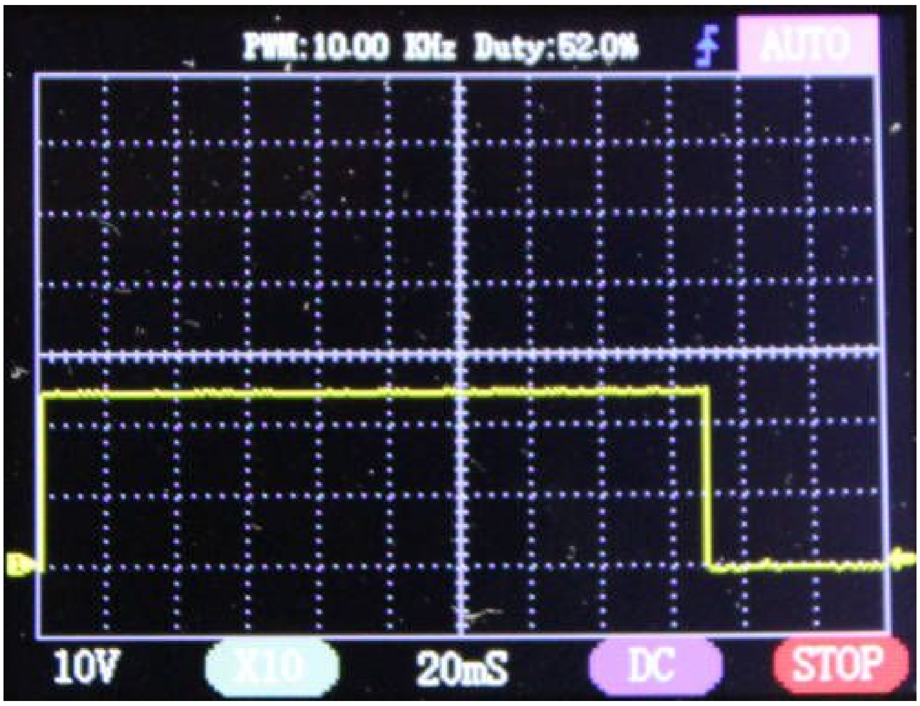


PLEASE NOTE : the position of pins G4 and G15 may vary depending on the version of your ESP32 card.

Analysis of encoder signals

As our encoder is connected, each terminal A or B receives a voltage, here 3.3V, the intensity of which is limited by a resistor of 10Kohms.

Analysis of the signal on terminal G15 clearly shows the presence of the 3.3V voltage :



In this signal capture, the low level on terminal G15 appears when operating the encoder control rod. When idle, the signal on terminal G15 is at high level.

This changes everything, because, at the programming level, we must process the G15 interrupt as a falling edge.

Encoder programming

The encoder will be managed by interrupt. Interrupts trigger the program only if a particular signal reaches a well-defined level.

We will manage a single interrupt on the GPIO G15 terminal:

```
interrupts

\ enable interrupt on GPIO G15
: intG15enable ( -- )
    15 GPIO_INTR_POSEDGE gpio_set_intr_type drop
  ;

\ disable interrupt on GPIO G15
: intG15disable ( -- )
    15 GPIO_INTR_DISABLE gpio_set_intr_type drop
  ;

: pinsInit ( -- )
    04 input pinmode            \ GO4 as an input
    04 gpio_pulldown_en drop    \ Enable pull-down on GPIO 04
    15 input pinmode             \ G15 as an input
    15 gpio_pulldown_en drop     \ Enable pull-down on GPIO 15
    intG15enable
  ;
```

In the word **pinsInit** , we initialize the GPIO pins G4 and G15 as input. Then we determine the interrupt mode of G15 on falling edge with **15 GPIO_INTR_POSEDGE gpio_set_intr_type drop** .

Testing the encoding

This part of code is not to be used in a final assembly. It is only used to check that the encoder is correctly connected and working properly:

```
: test ( -- )
    cr ." PIN: "
    cr ."  - G15: " 15 digitalRead .
    cr ."  - G04: " 04 digitalRead .
  ;

pinsInit    \ initialise G4 and G15
' test 15 pinchange
```

It is the **'test 15 pinchange' sequence** which tells ESP32Forth to execute the test code if an interrupt is triggered by action of terminal G15.

Result of the action on our encoder. We only kept the results of actions arriving at the stop, once counterclockwise, then clockwise:

```
PIN:
 - G15: 1  \ reverse clockwise turn
 - G04: 1
PIN:
 - G15: 0  \ clockwise turn
 - G04: 1
```

Increment and decrement a variable with the encoder

Now that we have tested the encoder by hardware interrupt, we will be able to manage the content of a variable. To do this, we define our variable **KYvar** and the words allowing us to modify its content:

```
0 value KYvar   \ content is incremented or decremented

\ increment content of KYvar
: incKYvar ( n -- )
    1 +to KYvar
  ;

\ decrement content of KYvar
: decKYvar ( n -- )
```

```
    -1 +to KYvar
  ;
```

The word **`incKYvar`** increments the content of **`Kyvar`** . The word **`decKYvar`** decrements the content of **`KYvar`** .

We test the modification of the content of the variable **`KYvar`** via this word **`testIncDec`** defined as follows:

```
\ used by interruption when G15 activated
: testIncDec ( -- )
    intG15disable
    15 digitalRead  if
        04 digitalRead  if
            decKYvar
        else
            incKYvar
        then
        cr ." KYvar: " KYvar .
    then
    1000 0 do loop  \ small wait loop
    intG15enable
  ;

pinsInit
' testIncDec 15 pinchange
```

Turning the encoder control to the right (clockwise) will increment the contents of the KYvar variable. A rotation to the left decrements the content of the KYvar variable:

```
pinsInit
' testIncDec 15 pinchange
-->
KYvar: 1\rotate Clockwise
KYvar: 2
KYvar: 3
KYvar: 4
KYvar: 3 \ rotate Contra Clockwise
KYvar: 2
KYvar: 1
KYvar: 0
KYvar: -1
KYvar: -2
```

Flashing of an LED per timer

Getting started with FORTH programming

Any beginner in programming knows this more than classic example very well: the flashing of an LED. Here is the source code, in C language for ESP32 :

```
/*
 * This ESP32 code is created by esp32io.com
 * This ESP32 code is released in the public domain
 * For more detail (instruction and wiring diagram),
 *  visit https://esp32io.com/tutorials/esp32-led-blink
 */

// the code in setup function runs only one time when ESP32 starts
void setup() {
  // initialize digital pin GIOP18 as an output.
  pinMode(18, OUTPUT);
}

// the code in loop function is executed repeatedly infinitely
void loop() {
  digitalWrite(18, HIGH); // turn the LED on
  delay(500);             // wait for 500 milliseconds
  digitalWrite(18, LOW);  // turn the LED off
  delay(500);             // wait for 500 milliseconds
}
```

In FORTH language, it's not much different :

```
18 constant myLED

: led.blink ( -- )
    myLED output pinMode
    begin
        HIGH myLED pin
        500 ms
        LOW myLED  pin
        500 ms
    key? until
  ;
```

If you compile this FORTH code with ESP32forth installed on your ESP32 board and type `led.blink` from the terminal, the LED connected to the GPIO18 port will blink.

To inject code written in C language, it will be necessary to compile it on the PC, then upload it to the ESP32 card, operations which take some time. Whereas with the FORTH language, the compiler is already operational on our ESP32 board. The compiler will compile the program written in FORTH language in two to three seconds and allow its immediate execution by simply typing the word containing this code, here `led.blink` for our example.

In FORTH language, we can compile hundreds of words and test them immediately, all individually, which is not possible at all in the C language.

We factor our FORTH code like this :

```
18 constant myLED

: led.on ( -- )
    HIGH myLED pin
  ;

: led.off ( -- )
    LOW myLED pin
  ;

: waiting ( -- )
    500 ms
  ;

: led.blink ( -- )
    myLED output pinMode
    begin
        led.on      waiting
        led.off     waiting
    key? until
  ;
```

From the terminal, we can simply turn the LED on by typing `led.on` and turn it off by typing `led.off` . Execution of `led.blink` remains possible.

The aim of factorization is to divide a complex and difficult to read function into a set of simpler and more readable functions. With FORTH, factorization is recommended, on the one hand to allow easier debugging, and on the other hand to allow the reuse of factored words.

These explanations may seem trivial to those who know and master the FORTH language. This is far from obvious for people programming in C, who are forced to group function calls into the general `loop() function` .

Now that this is explained, we will forget everything ! Because...

Flashing by TIMER

We will forget everything that was explained previously. Because this LED blinking example has a huge downside. Our program does just that and nothing else. In short, it's a real waste of hardware and software to flash an LED on our ESP32 card. We will see a very different way of producing this flashing, in FORTH language exclusively.

ESP32forth has two words that will be very useful to manage this LED flashing: `interval` and `rerun` .

But before discussing how these two words work, let's take a look at the notion of interruption...

Hardware and software interrupts

If you plan to manage microcontrollers without worrying about hardware or software interrupts, then abandon computer development for ESP32 boards!

You have the right to start and not experience interruptions. And we're going to explain interrupts to you and how to use timer interrupts.

Here is a non-computer example of what an interrupt is:

- you are expecting an important package;
- you go down to the gate of your home every minute to see if the postman has arrived.

In this scenario, you actually spend your time going down, looking, back up. In fact, you hardly have time to do anything else...

In reality, this is what should happen :

- you stay in your home ;
- the postman arrives and rings the doorbell ;
- you go down and collect your package...

A microcontroller, which includes the ESP32 card, has two types of interrupts :

- **hardware interrupts** : they are triggered by a physical action on one of the GPIO inputs of the ESP32 card ;
- **software interrupts** : they are triggered if certain registers reach pre-defined values.

This is the case for timer interrupts, which we will define as software interrupts.

Use the words interval and rerun

The word `interval` is defined in the `timers vocabulary` . It accepts three parameters :

- **xt** which executes the code for the word to be thrown when the interrupt is triggered ;
- **usec** is the wait time, in microseconds, before triggering the interrupt ;
- **t** is the number of the timer to trigger. This parameter must be in the range [0..3]

Let's partially take the factored code of our LED flashing :

```
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  \ select timers vocabulary
: led.toggle ( -- )
    LED_STATE if
        led.off
    else
        led.on
    then
    0 rerun
  ;

' led.toggle 500000 0 interval

: led.blink
    myLED output pinMode
    led.toggle
  ;
```

The word `rerun` is preceded by the number of timer activated before the definition of `interval` . The word `rerun` must be used in the definition of the word executed by the timer.

The word `led.blink` initializes the GPIO output used by the LED, then executes `led.toggle.`

In this sequence FORTH `' led.toggle 500000 0 interval` , we initialize timer 0 by recovering the execution code of the word using `rerun` , followed by the time interval, here 500 milliseconds, then the number of the timer to trigger.

The LED flashing starts immediately after execution of the word `led.blink` .

The FORTH interpreter of ESP32forth remains accessible while the LED flashes, something impossible in C language!

Housekeeper timer

Preamble

It's 1990. He's a computer programmer who works a lot. So he sometimes leaves his office a little late.

And it was during one of his late exits from the office that he entered the corridor, one of those corridors with a timer button at each end. The light is already on. But out of reflex, our programmer friend presses the switch and pricks his finger. A wooden point is inserted into the switch to block the timer.

It's the cleaning lady who is cleaning the floor who explains to him: "yes. The timer only lasts one minute. And I often find myself in the dark. As I'm tired of pressing again without stop on the timer switch, I block the button with this little wooden point"…

A solution

This anecdote sparked an idea in our programmer's head. As he had some knowledge about microcontrollers, he set out to find a solution for the cleaning lady.

History does not say in what language he programmed his solution. Certainly in assembler.

He derived the control of the lights to his circuit :

- an ordinary press starts the timer for one minute ;
- if the light is on, any brief press of a button reduces the ignition delay to one minute ;
- our programmer's secret is to have planned a long press of 3 seconds or more. This long press starts the timer for 10 minutes of lighting ;

- if the timer is in long circuit, another long press reduces the timer delay to one minute ;
- a short beep acknowledges the activation or deactivation of a long timer cycle.

The cleaning lady really appreciated this improvement in the timer. She no longer needed to block the button in any way.

What about the other workers? Since no one was aware of this feature, they continued to use the timer by briefly pressing the activation switch.

A FORTH timer for ESP32Forth

You understand, we are going to use `timers` to manage a timer by integrating the scenario described previously.

```
\ myLIGHTS connecté à GPIO18
18 constant myLIGHTS

\ définit temps max pour cycle normal ou étendu, en secondes
 60 constant MAX_LIGHT_TIME_NORMAL_CYCLE
600 constant MAX_LIGHT_TIME_EXXTENDED_CYCLE

\ temps max pour cycle normal ou étendu, en secondes
0 value MAX_LIGHT_TIME

timers
\ coupe éclairage si MAX_LIGHT_TIME égal 0
: cycle.stop ( -- )
    -1 +to MAX_LIGHT_TIME       \ décrémente temps max de 1 seconde
    MAX_LIGHT_TIME 0 = if
        LOW myLIGHTS pin        \ coupe éclairage
    else
        0 rerun
    then
  ;

\ initialise timer 0
' cycle.stop 1000000 0 interval

\ démarre un cycle d’éclairage, n est délai en secondes
: cycle.start ( n -- )
    1+ to MAX_LIGHT_TIME        \ sélect. Temps max
    myLIGHTS output pinMode
    HIGH myLIGHTS pin           \ active éclairage
    0 rerun
  ;
```

We can already test our timer :

```
 3 cycle.start    \ turns on lights for 3 seconds
```

```
10 cycle.start    \ turns on lights for 10 seconds
```

If we restart **cycle.start** while the light is on, we start again for a new lighting cycle of n seconds.

We therefore still have to manage the activation of these cycles from a switch.

Management of the light on button

This is not rocket science. We will manage a push button. As we have an ESP32 card on hand, programmable with ESP32Forth, we will take advantage of it to manage this button by interrupts. The interrupts managing the GPIO terminals on the ESP32 board are hardware interrupts.

Our button is mounted on the GPIO17 (G17) terminal.

We define two words, **intPosEdge** and **intNegEdge** , which determine the type of triggering of the interrupt:

- **intPosEdge** to trigger the interrupt on rising edge;
- **intNegEdge** to trigger the interrupt on falling edge.

```
17 constant button  \ mount button on GPIO17

interrupts                              \ select interrupts vocabulary

\ interrupt activated for upraising signal
: intPosEdge ( -- )
    button #GPIO_INTR_POSEDGE gpio_set_intr_type drop
  ;

\ interrupt activated for falldown signal
: intNegEdge ( -- )
    button #GPIO_INTR_NEGEDGE gpio_set_intr_type drop
  ;
```

We then need to define some variables and constants:

- two constants, **CYCLE_SHORT** and **CYCLE_LONG** which will be used to define the duration of lighting of the lights. Here we chose 3 and 10 seconds to do our tests ;
- the **msTicksPositiveEdge variable** which stores the position of the wait counter delivered by ms-ticks ;
- **DELAY_LIMIT** constant which determines the threshold for determining a short or long press on the push button. Here, it's 3000 milliseconds, or 3 seconds. A normal

user will NEVER press the light on button for 3 seconds. Only the cleaning lady knows the maneuver to have a long continuous lighting...

```
03 constant CYCLE_SHORT       \ lighting duration for short press,
in seconds
10 constant CYCLE_LONG        \ lighting duration for long press

\ stores value of ms-ticks on rising edge
variable msTicksPositiveEdge

\ deadline limit: if delay < DELAY_LIMIT, short cycle
3000 constant DELAY_LIMIT
```

The word **getButton** is launched at each interrupt triggered by pressing the push button connected to GPIO17 (G17) on our ESP32 board.

getButton execution , interrupts on G17 are disabled. This interruption will be reactivated at the end of the definition. This disabling is necessary to prevent interrupt stacking.

Disabling is followed by the **70000 0 do loop** . This loop is used to manage contact bounces. Here we manage the debounce by software.

```
\ word executed by interrupt
: getButton ( -- )
    button gpio_intr_disable drop
    70000 0 do loop   \ anti rebond
    button digitalRead 1 =
    if
        ms-ticks msTicksPositiveEdge !
        intNegEdge
    else
        intPosEdge
        ms-ticks msTicksPositiveEdge @ -
        DELAY_LIMIT >
        if      CYCLE_LONG  cr ." BEEP"
        else    CYCLE_SHORT cr ." ----"
    then
    cycle.start
    button gpio_intr_enable drop
  ;
```

On the rising edge, the word **getButton** records the state of the delay counter and positions the interrupts on the falling edge. Then we leave this word by reactivating the interruptions.

At the falling edge, the word **getButton** calculates the time elapsed since the rising edge. If this delay is greater than **DELAY_LIMIT** , a long ignition cycle is initiated. Otherwise, a short ignition cycle is initiated.

The engagement of a long ignition cycle is indicated by the display of “BEEP” on the terminal.

In the original scenario, this is materialized by a short beep.

Finally, we initialize the button and the hardware interrupt on this button:

```
\ initialize button and interrupt vectors
button input pinMode                \ selects G17 in input mode
button gpio_pulldown_en drop        \ activates internal resistance of
G17
' getButton button pinchange
intPosEdge

forth
```

Conclusion

Watch the assembly video: https://www.youtube.com/watch?v=OHWMh_bIWz0

This very simple case study shows how to simultaneously manage the timer and a hardware interrupt.

These two mechanisms are very little preemptive. The timer leaves access to the FORTH interpreter available. The hardware interrupt is operational even if FORTH is running another process.

We don't multitask. It’s important to say it!

I only hope that this textbook case will now give you a lot of ideas for your developments...

Software real-time clock

The word MS-TICKS

The word **MS-TICKS** is used in the definition of the word **ms** :

```
DEFINED? ms-ticks [IF]
  : ms ( n -- )
      ms-ticks >r
      begin
          pause ms-ticks r@ - over
      >= until
      rdrop drop
    ;
[THEN]
```

This word **MS-TICKS** is at the heart of our investigations. If we start up the ESP32 card, its execution restores the number of milliseconds elapsed since the ESP32 card was started up. This value is still growing. The saturation value of this count is 2^{32}-1, or 4294967295 milliseconds, or approximately 49 days...

Each time the ESP32 card is restarted, this value restarts from zero.

Managing a software clock

From the **HH MM SS** (Hours, minutes, seconds) data, it is easy to reconstruct an integer value, in milliseconds, corresponding to the time elapsed since 00:00:00. If we subtract the value of **MS-TICKS** from this time , we have a starting time value to determine the real time . We therefore initialize a basic counter **currentTime** from the word **RTC.set-time** :

```
0 value currentTime

\ store current time
: RTC.set-time { hh mm ss -- }
    hh 3600 *
    mm 60 *
    ss + +  1000 *
    MS-TICKS - to currentTime
  ;
```

Initialization example: **22 52 00 RTC.set-time** initializes the time base for 22:52:00...

To properly initialize, prepare the three values **HH MM SS** followed by the word **RTC.set-time**, watch your watch. When the expected time arrives, execute the initialization sequence.

HH MM and **SS** values of the current time, using this word :

```
\ get current time in seconds
: RTC.get-time ( -- hh mm ss )
    currentTime MS-TICKS + 1000 /
    3600 /mod swap 60 /mod swap
  ;
```

Finally, we define the word `RTC.display-time` which allows you to display the current time after initialization of our software clock:

```
\ used for SS and MM part of time display
: :## ( n -- n' )
    #  6 base ! #  decimal  [char] : hold
  ;

\ display current time
: RTC.display-time ( -- )
    currentTime MS-TICKS + 1000 /
    <# :## :## 24 MOD #S #> type
  ;
```

The next step would be to connect to a time server, with the NTP protocol, to automatically initialize our software clock.

Measuring the execution time of a FORTH word

Measuring the performance of FORTH definitions

Let's start by defining the word `measure:` which will perform these execution time measurements:

```
: measure: ( exec: -- <word> )
    ms-ticks >r
    ' execute
    ms-ticks r> -
    cr ." execution time: "
    <# # # # [char] . hold #s #> type ." sec." cr
  ;
```

In this word, we retrieve the time by `ms-ticks` , then we retrieve the execution code of the word that follows `measure:,` we execute this word, we retrieve the new time value by `ms-ticks` . We make the difference, which corresponds to the elapsed time, in milliseconds, taken by the word to execute. Example :

```
measure: words
\ displays: execution time: 0.210sec.
```

The word `words` was executed in 0.2 seconds. This time does not take into account transmission delays by the terminal. This time also does not take into account the delay taken by `measure:` to retrieve the execution code of the word to be measured.

If there are parameters to pass to the word to measure, these must be stacked before calling `measure:` followed by the word to measure:

```
: SQUARE ( n -- n-exp2 )
    dup *
  ;
3 measure: SQUARE
\ poster:
\ execution time: 0.000sec.
```

This result means that our `SQUARE definition` runs in less than a millisecond.

We will repeat this operation a certain number of times:

```
: test-square ( -- )
    1000 for
        3 SQUARE drop
    next
  ;
3 measure: test-square
3 measure: test-square
```

```
\ poster:
\ execution time: 0.001sec.
```

By executing the word `SQUARE` 1000 times , preceded by a stacking of values and unstacking of the result, we arrive at an execution time of 1 millisecond. We can reasonably deduce that `SQUARE` executes in less than a micro-second!

Testing a few loops

We are going to test a few loops, with 1 million iterations. Let's start with a `do-loop` :

```
: test-loop ( -- )
    1000000 0 do
    loop
   ;
measure: test-loop
\ display:
\ execution time: 1.327sec.
```

Now let's see with a `for-next loop` :

```
: test-for ( -- )
    1000000 for
    next
   ;
measure: test-for
\ displays:
\ execution time: 0.096sec.
```

The **for-next loop** runs almost 14 times faster than the **do-loop.**

Let's see what a **begin-until** loop has in store:

```
: test-begin ( -- )
    1000000 begin
        1- dup 0=
    until
   ;
measure: test-begin
\ displays:
\ execution time: 0.273sec.
```

This is more efficient than the **do-loop** , but still three times slower than the **for-next loop** .

You are now equipped to create even more efficient FORTH programs.

Program a sunshine analyzer

Preamble

As part of a solar project using several solar panels and their microinverter, there appear some problems with managing the electrical energy produced.

The main concern is to activate large consumer devices only if the solar panels produce in full sun. One device in particular is concerned, the hot water cumulus:

- activate the device when the panels are in direct sunlight;
- deactivate the device when clouds pass.

Microinverters inject power into the general electrical grid. If a device that consumes a lot of electricity is active when clouds pass, this device will be powered primarily by the general network.

In this article, we present a solution enabling cloud detection using a miniature solar panel and an ESP32 card.

Full code available here:
https://github.com/MPETREMANN11/ESP32forth/blob/main/ADC/solarLightAnalyzer.txt

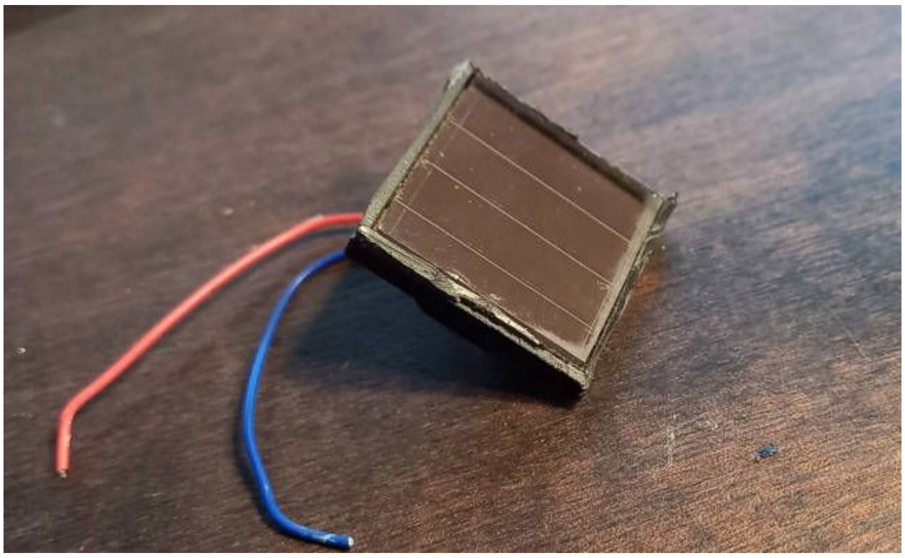
The miniature solar panel

To create our cloud detector, we will use a very small solar panel, here a 25mm x 25mm panel.

Recovery of a miniature solar panel

This miniature solar panel is recovered from a garden lamp that is out of order :

Here is our mini solar panel taken out of this garden lamp :



We sacrifice two dupont connectors to allow various measurements to be carried out on our prototype plate. These connectors are soldered onto the two red and blue wires coming out of the mini solar panel.

Measurement of solar panel voltage

We start by measuring the no-load voltage of our mini solar panel, here with an oscilloscope. This voltage measurement can also be carried out with a voltmeter :

In bright light, the measured voltage amounts to 14.2 Volts!

Under diffused light, the voltage drops to 5.8 Volts.

By covering the mini solar panel with your hand, the voltage drops to almost 0 Volts.

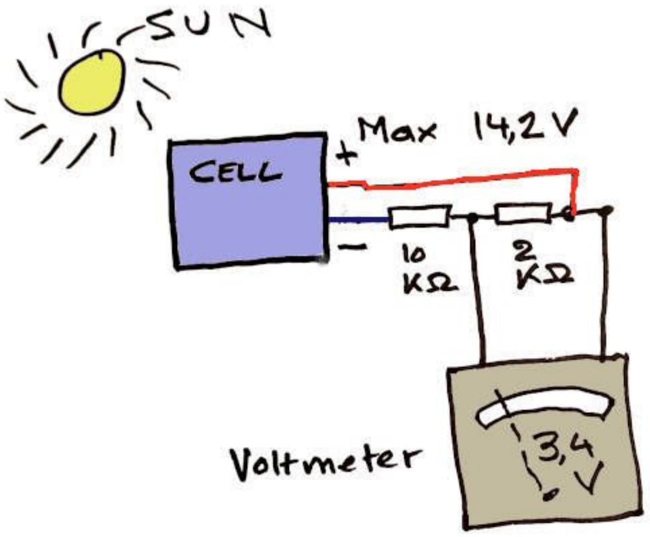
Solar panel current measurement

The current, i.e. the intensity, must be measured using an ammeter. The ammeter function of a universal controller will be suitable. Short-circuiting the mini solar panel in bright light allows a current of 10 mA to be measured.

Our mini solar panel therefore has an approximate power of 0.2 Watt.

Before connecting our mini solar panel to the ESP32 card, it is essential to lower the output voltage. It is out of the question to inject this voltage of 14.2 Volts into an input of the ESP32 card. Such voltage would destroy the internal circuitry of the ESP32 board.

Lowering the solar panel voltage



The idea is to lower the voltage across our mini solar panel. After a few tests, we choose two resistors, one of 220 Ohms, the other of 1K Ohms. Mounting these resistors:

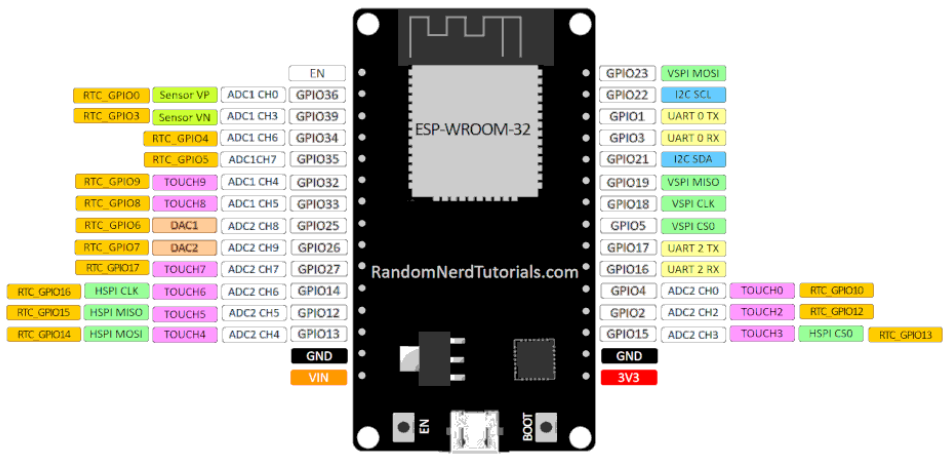
The voltage measurement is taken between the two resistors and the positive terminal of the solar panel.

The voltmeter now indicates a maximum voltage of 3.2V in bright light, a voltage of 0.35V in diffused light.

Programming the solar analyzer

The ESP32 board has 18 12-bit channels enabling analog-to-digital conversion (ADC). To analyze the voltage of our mini solar panel, a single ADC channel is necessary and sufficient.

Only 15 ADC channels are available:


EN
RTC_GPIO0 Sensor VP ADC1 CH0 GPIO36
RTC_GPIO3 Sensor VN ADC1 CH3 GPIO39
RTC_GPIO4 ADC1 CH6 GPIO34
RTC_GPIO5 ADC1CH7 GPIO35
RTC_GPIO9 TOUCH9 ADC1 CH4 GPIO32
RTC_GPIO8 TOUCH8 ADC1 CH5 GPIO33
RTC_GPIO6 DAC1 ADC2 CH8 GPIO25
RTC_GPIO7 DAC2 ADC2 CH9 GPIO26
RTC_GPIO17 TOUCH7 ADC2 CH7 GPIO27
RTC_GPIO16 HSPI CLK TOUCH6 ADC2 CH6 GPIO14
RTC_GPIO15 HSPI MISO TOUCH5 ADC2 CH5 GPIO12
RTC_GPIO14 HSPI MOSI TOUCH4 ADC2 CH4 GPIO13
GND
VIN
ESP-WROOM-32
RandomNerdTutorials.com
EN
BOOT
GPIO23 VSPI MOSI
GPIO22 I2C SCL
GPIO1 UART 0 TX
GPIO3 UART 0 RX
GPIO21 I2C SDA
GPIO19 VSPI MISO
GPIO18 VSPI CLK
GPIO5 VSPI CS0
GPIO17 UART 2 TX
GPIO16 UART 2 RX
GPIO4 ADC2 CH0 TOUCH0 RTC_GPIO10
GPIO2 ADC2 CH2 TOUCH2 RTC_GPIO12
GPIO15 ADC2 CH3 TOUCH3 HSPI CS0 RTC_GPIO13
GND
3V3

We will use one, the **ADC1_CH6 channel** which is attached to pin **G34 :**

```
34 constant SOLAR_CELL

: init-solar-cell ( -- )
    SOLAR_CELL input pinMode
  ;

init-solar-cell
```

To read the voltage at the point between the two resistors, simply run `SOLAR_CELL analogRead` . This sequence drops a value between 0 and 4095. The lowest value corresponds to zero voltage. The highest value corresponds to a voltage of 3.3 Volts.

`solar-cell-read` definition to recover this voltage:

```
: solar-cell-read ( -- n )
    SOLAR_CELL analogRead
  ;
```

Let's test this definition in a loop :

```
: solar-cell-loop ( --)
    init-solar-cell
    begin
        solar-cell-read cr .
        200 ms
    key? until
  ;
```

When running `solar-cell-loop` , every 200 milliseconds, the ADC voltage conversion value is displayed:

```
...
322
331
290
172
39
0
0
0
0
19
79
86
...
```

Here the values were obtained by illuminating the mini solar panel with a high power lamp. Zero values correspond to the absence of lighting.

Tests with the real sun show measurements exceeding 300.

Managing activation and deactivation of a device

To begin, we will define two pins, one pin reserved for managing an activation signal, the other for a deactivation signal:

- G17 pin connected to a green LED. This pin is used to activate a device.
- G16 pin connected to a red LED. This pin is used to deactivate a device.

```
17 constant DEVICE_ON      \ green LED
16 constant DEVICE_OFF     \   red LED

: init-device-state ( -- )
    DEVICE_ON  output pinMode
    DEVICE_OFF output pinMode
  ;
```

We could have used a single pin to manage the remote device. But some devices, like bistable relays, have two coils :

- the first coil is powered so that the contacts switch. The state does not change when the coil is no longer energized ;
- to return to the initial state, the second coil is powered.

For this reason, our programming will take this type of device into account.

```
\ define trigger high state delay
500 value DEVICE_DELAY

\ set HIGH level of trigger
: device-activation { trigger -- }
    trigger HIGH digitalWrite
    DEVICE_DELAY ?dup
    if
        ms
        trigger LOW  digitalWrite
    then
  ;
```

Here, the pseudo-constant `DEVICE_DELAY` is used to indicate the delay during which the control signal should be kept high. After this time, the control signal returns to the low state.

If the value of `DEVICE_DELAY` is zero, the control signal remains high.

It is the word `trigger-activation` which manages the activation of the corresponding pin:

- `TRIGGER_ON trigger-activation` permanently or transiently sets the pin attached to the green LED high;
- `TRIGGER_OFF trigger-activation` sets the pin attached to the red LED permanently or transiently high.

We now define two words, `device-ON` and `device-OFF` , respectively responsible for activating and deactivating the device intended to be controlled by pins G16 and G17:

```
\ define device state: 0=LOW, -1=HIGH
0 value DEVICE_STATE

: enable-device ( -- )
    DEVICE_STATE invert
    if
        DEVICE_OFF LOW  digitalWrite
        DEVICE_ON  device-activation
        -1 to DEVICE_STATE
    then
  ;

: disable-device ( -- )
    DEVICE_STATE
    if
        DEVICE_ON  LOW  digitalWrite
        DEVICE_OFF device-activation
         0 to DEVICE_STATE
    then
  ;
```

The device state is stored in `DEVICE_STATE` . This state is tested before attempting to change state. If the device is active, it will not be reactivated repeatedly. Same if the device is inactive.

```
\ define trigger value for sunny or cloudy sky
300 value SOLAR_TRIGGER

\ if solar light > SOLAR_TRIGGER, activate action
: action-light-level ( -- )
    solar-cell-read SOLAR_TRIGGER >=
    if
        enable-device
    else
        disable-device
    then
  ;
```

Triggered by timer interrupt

The most elegant way is to use a timer interrupt. We will use timer 0:

```
  0 to DEVICE_DELAY
200 to SOLAR_TRIGGER
init-solar-cell
init-device-state

timers
: action ( -- )
    action-light-level
    0 rerun
  ;

' action 1000000 0 interval
```

From now on, the timer will analyze the light flow every second and act accordingly. Link to video: https://youtu.be/lAjeev2u9fc

For this video, we act on two parameters:

- **`0 to DEVICE_DELAY`** lights the LEDs permanently. The red LED indicates that the device is deactivated. The green LED indicates device activation;
- **`200 to SOLAR_TRIGGER`** determines the threshold for triggering the sunshine state. This parameter is adjustable to adapt to the characteristics of the mini solar panel.

The **`action word`** works by timer interrupt. It is therefore not necessary to have a general loop for the detector to work.

Devices controlled by the sunshine sensor

In summary, we have two control wires, one wire corresponding to the green LED in the video, the other wire corresponding to the red LED. The program is designed so that both control wires cannot be active at the same time.

To have a continuous signal on either control wire, the **`DEVICE_DELAY value only needs to`** be zero. Here is how to initialize this scenario :

```
\ start with Constant Command Signal
: start-CCS ( -- )
      0 to DEVICE_DELAY
    200 to SOLAR_TRIGGER
    init-solar-cell
    init-device-state
    disable-device
    [ timers ] ['] action 1000000 0 interval
  ;
```

And to have timed commands, we will assign to **DEVICE_DELAY** the delay of the level of the activation or deactivation command of the device.

```
\ start with Temporized Command Signal
: start-TCS ( -- )
    300 to DEVICE_DELAY
    200 to SOLAR_TRIGGER
    init-solar-cell
    init-device-state
    disable-device
    [ timers ] ['] action 1000000 0 interval
  ;
```

start-TCS scenario is typical of a pulse-operated bistable relay control. The relay activates if it receives an activation command. Even if the activation signal falls, the bistable relay remains active. To deactivate the bistable relay, a deactivation command must be transmitted to it on the deactivation line.

In conclusion, our solar light analyzer can control a wide variety of devices. It is enough to adapt the control interfaces of these devices to the characteristics of the GPIO ports of the ESP32 card.

Management of N/A (Digital/Analog) outputs

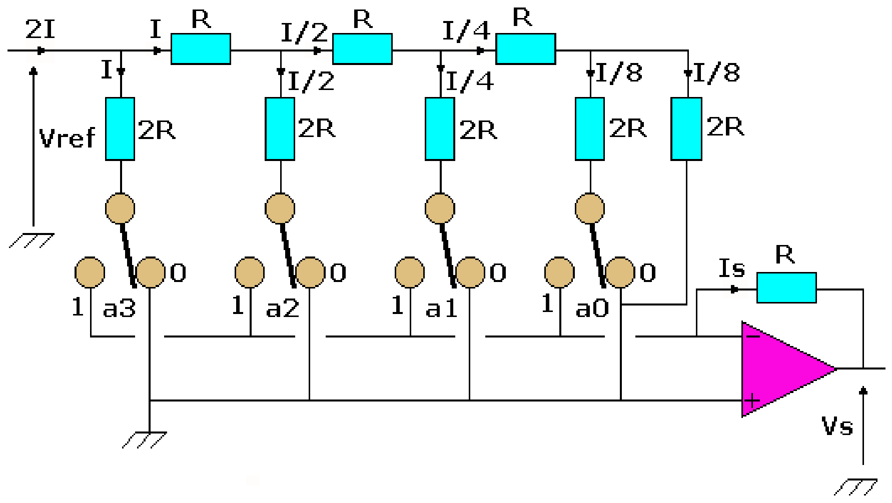
Digital/analog conversion

The conversion of a digital quantity into an electrical voltage proportional to this digital quantity is a very interesting functionality on a micro-controller.

When you use the Internet and make a VOIP phone call, your voice is transformed into numerical values. That of your correspondent will be inversely transformed from numbers to sound signals. This process uses analog to digital conversion and vice versa.

D/A conversion with R2R circuit

Here is the basic diagram of a 4-bit digital to analog converter :



The value to convert, on 4 bits, is distributed over 4 pins a0 to a3. The reference voltage is injected at the top left of the circuit. This voltage generates an intensity 2I if this current does not pass through any resistance.

Depending on the activated bits, for each bit the voltage is divided and added to that of the other active bits. For example, if a2 and a0 are active, the output current Is will be the sum I/2 and I/8.

For this 4-bit circuit, the conversion step is I/16. With ESP32, the conversion is done on 8 bits. The conversion step will therefore be I/256.

D/A conversion with ESP32

No ARDUINO board has a D/A conversion output. To perform a D/A conversion with an ARDUINO board, you must use an external component.

With the ESP32 card, we have two pins, G25 and G26, corresponding to D/A conversion outputs.

For our first D/A conversion experiment with the ESP32 board, we will connect two LEDs to pins G25 and G26:

```
\ define Gx to LEDs
25 constant ledBLUE      \  blue led on G25
26 constant ledWHITE     \ white led on G26
```

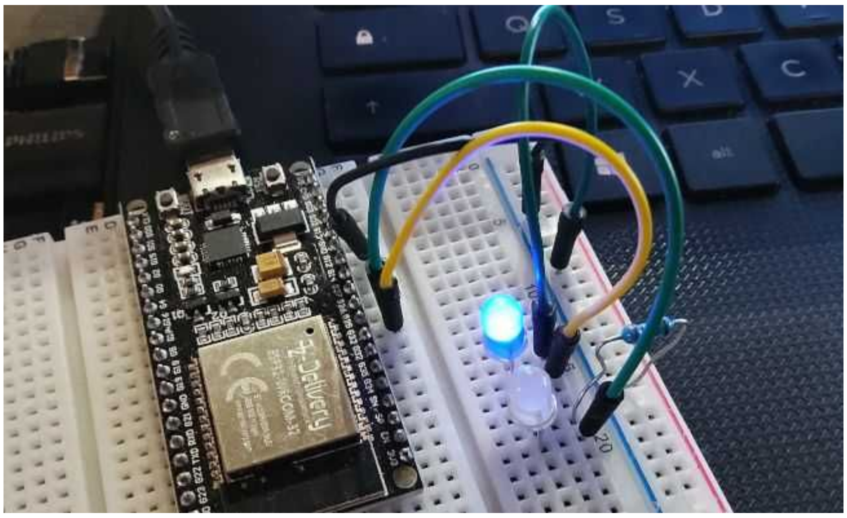
Before performing a D/A conversion, we plan to initialize pins G25 and G26:

```
\ init Gx as output
: initLeds ( -- )
    ledBLUE  output pinMode
    ledWHITE output pinMode
    ;
```

And we define two words allowing us to control the intensity of our two LEDs:

```
\ set intensity for BLUE led
: BLset ( val -- )
    ledBLUE   swap dacWrite
    ;

\ set intensity for WHITE led
: WHset ( val -- )
    ledWHITE  swap dacWrite
    ;
```

The words `BLset` and `WHset` accept as parameters a numeric value in the range 0..255.



In the photo, after `initLeds` , the sequence `200 BLset` lights the blue LED at reduced power.

To turn it on at full power, we will use the sequence `255 BLset`

To turn it off completely, we will send this sequence `0 BLset`

Possibilities of D/A conversion

Here, with our two LEDs, we have created a simple and uninteresting assembly.

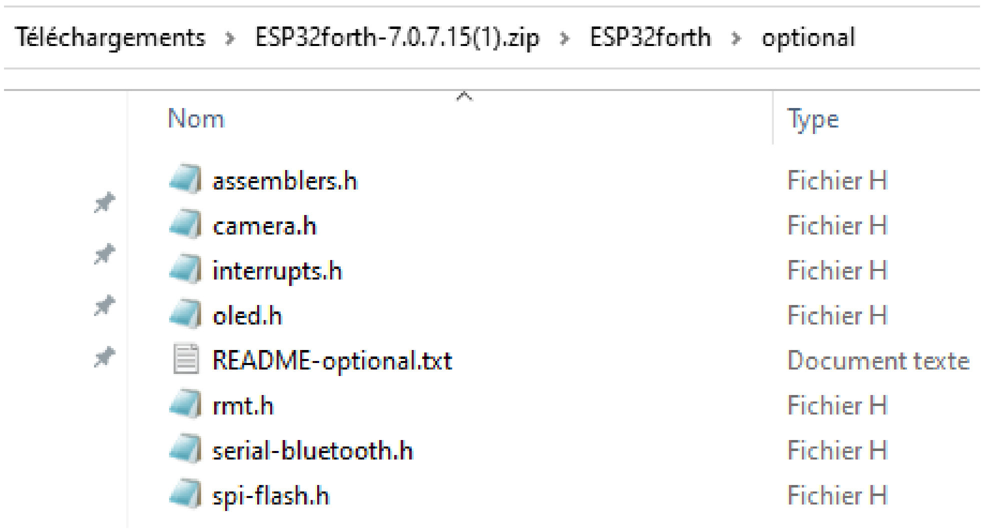
This montage has the merit of showing that the D/A conversion works perfectly. The D/A conversion allows:

- power control through a dedicated circuit, a variator for an electric motor for example;
- generation of signals: sinusoid, square, triangle, etc...
- sound file conversion
- sound synthesis...

Full code available here:
https://github.com/MPETREMANN11/ESP32forth/blob/main/DAC/DAoutput.txt

Installing the OLED library for SSD1306

Since ESP32forth version 7.0.7.15, the options are available in the optional folder :

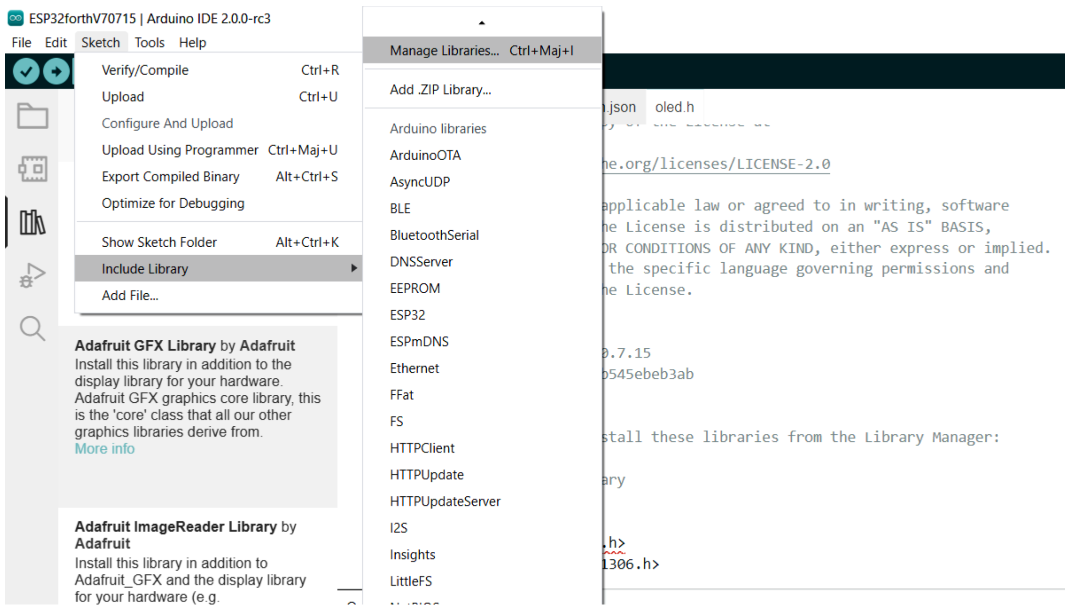

Téléchargements > ESP32forth-7.0.7.15(1).zip > ESP32forth > optional

Nom	Type
assemblers.h	Fichier H
camera.h	Fichier H
interrupts.h	Fichier H
oled.h	Fichier H
README-optional.txt	Document texte
rmt.h	Fichier H
serial-bluetooth.h	Fichier H
spi-flash.h	Fichier H

To have the `oled vocabulary, copy the` **oled.h** file to the folder containing the **ESP32forth.ino** file.

Then launch ARDUINO IDE and select the most recent **ESP32forth.ino** file**.**

If the OLED library has not been installed, in ARDUINO IDE, click *Sketch* and select *Include Library*, then select *Manage Libraries*.



Adafruit GFX Library by **Adafruit**
Install this library in addition to the display library for your hardware. Adafruit GFX graphics core library, this is the 'core' class that all our other graphics libraries derive from.
More info

Adafruit ImageReader Library by **Adafruit**
Install this library in addition to Adafruit_GFX and the display library for your hardware (e.g.

```
e.org/licenses/LICENSE-2.0
applicable law or agreed to in writing, software
e License is distributed on an "AS IS" BASIS,
R CONDITIONS OF ANY KIND, either express or implied.
 the specific language governing permissions and
e License.

.7.15
545ebeb3ab

tall these libraries from the Library Manager:

ry

.h>
306.h>
```

In the left sidebar, look for the **Adafruit SSD1306 by Adafruit** library.

You can now start compiling the sketch by clicking on *Sketch* and selecting *Upload*.

Once the sketch is uploaded to the ESP32 board, launch the TeraTerm terminal. Check that the `OLED` vocabulary is present :

```
oled vlist    \ display:
OledInit SSD1306_SWITCHCAPVCC SSD1306_EXTERNALVCC WHITE BLACK
OledReset
HEIGHT WIDTH OledAddr OledNew OledDelete OledBegin OledHOME OledCLS
OledTextc
OledPrintln OledNumln OledNum OledDisplay OledPrint OledInvert
OledTextsize
OledSetCursor OledPixel OledDrawL OledCirc OledCircF OledRect
OledRectF
OledRectR OledRectRF oled-builtins
```

The I2C interface on ESP32

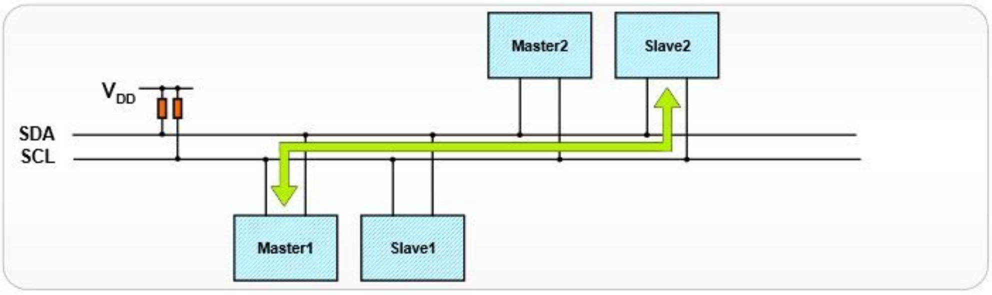
Introduction

I2C (means: Inter-Integrated Circuit, in English) is a computer bus which emerged from the “war of standards” launched by players in the electronic world. Designed by Philips for home automation and domestic electronics applications, it makes it possible to easily connect a microprocessor and various circuits, notably those of a modern television: remote control receiver, low frequency amplifier settings, tuner, clock, management of scart socket, etc.

There are countless peripherals using this bus, it can even be implemented by software in any microcontroller. The weight of the consumer electronics industry has enabled very low prices thanks to these numerous components.

This bus is sometimes called TWI (Two Wire Interface) or TWSI (Two Wire Serial Interface) by certain manufacturers.

Exchanges always take place between a single master and one (or all) slave(s), always at the initiative of the master (never from master to master or from slave to slave). However, nothing prevents a component from going from master to slave status and vice versa.



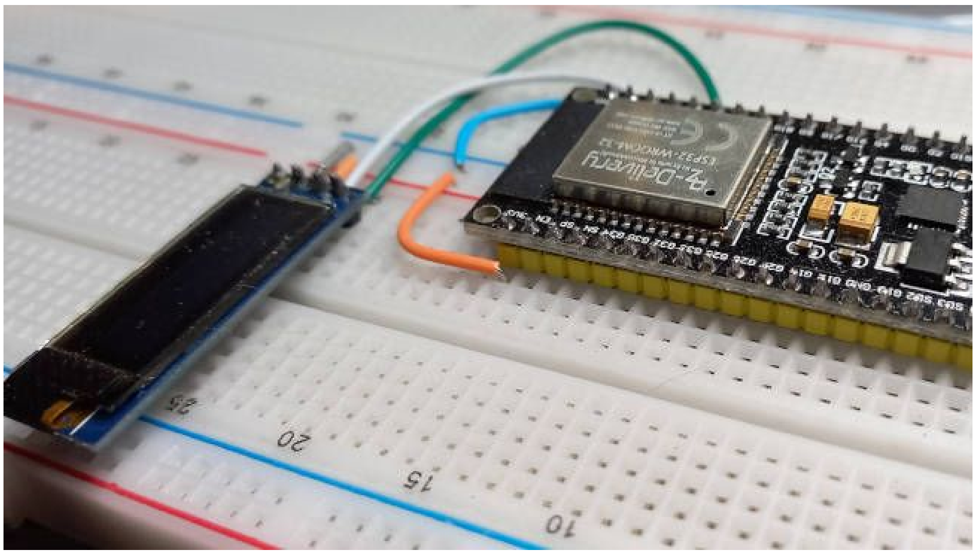
principle of an I2C bus

The connection is made via two lines:

- SDA (Serial Data Line): bidirectional data line,
- SCL (Serial Clock Line): bidirectional synchronization clock line.

We must not forget the mass which must be common to the equipment.

Both lines are pulled at voltage level VDD through pull-up resistors (RP).

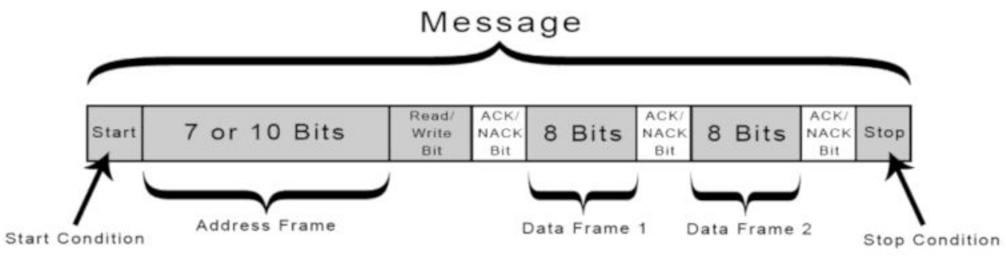

OLED display connected to the I2C bus

Master slave exchange

The message can be broken down into two parts:

- The master is the transmitter, the slave is the receiver:
  - emission of a START condition by the master ("S"),
  - transmission of the address byte or bytes by the master to designate a slave, with the R/W bit at 0 (see the section on addressing below),
  - response from the slave with an ACK acknowledgment bit (or NACK non-acknowledgement bit),
  - after each acknowledgment, the slave can request a pause ("PA").
  - emission of a command byte by the master for the slave,
  - response from the slave with an ACK acknowledgment bit (or NACK non-acknowledgement bit),
  - emission of a RESTART condition by the master ("RS"),
  - transmission of the address byte or bytes by the master to designate the same slave, with the R/W bit at 1.
  - response from the slave with an ACK acknowledgment bit (or NACK non-acknowledgement bit).

- The master becomes a receiver, the slave becomes a transmitter:
  - emission of a data byte by the slave for the master,
  - response from the master with an ACK acknowledgment bit (or NACK non-acknowledgement bit),
  - transmission of other data bytes by the slave with acknowledgment from the master,
  - for the last byte of data expected by the master, it responds with a NACK to end the dialogue,
  - emission of a STOP condition by the master (“P”).



Start condition : The SDA line goes from a high voltage level to a low voltage level before the SCL line goes from high to low.

Shutdown condition : The SDA line changes from a low voltage level to a high voltage level after the SCL line changes from low to high.

Address frame : a unique 7 or 10 bit sequence for each slave that identifies the slave when the master wants to talk to it.

Read/Write Bit : A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

ACK/NACK bit : each frame of a message is followed by an acknowledgment/non-acknowledgment bit. If an address frame or data frame has been successfully received, an ACK bit is returned to the sender.

Addressing

I2C doesn't have slave select lines like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by addressing. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave with which it wants to communicate to each slave connected to it. Each slave then compares the address sent by the master to its own. If the address matches, it returns a low voltage ACK bit to the master. If the address does not match, the slave does nothing and the SDA line remains high.

This is how the `Wire.detect word` detects devices connected to the i2c bus.

You can connect several different devices to the i2c bus. You cannot connect several copies of the same device to the same i2c bus.

Setting GPIO ports for I2C

Setting up the GPIO ports for the I2C bus is very simple:

```
\activate the wire vocabulary
wire
\ start the I2C interface using pin 21 and 22 on ESP32 DEVKIT V1
\ with 21 used as sda and 22 as scl.
21 22 wire.begin
```

I2C bus protocols

The dialogue is only between a master and a slave. This dialogue is always initiated by the master (Start condition): the master sends the address of the slave with whom it wants to communicate on the I2C bus.

The dialogue is always terminated by the master (Stop condition).

The clock signal (SCL) is generated by the master.

Detecting an I2C device

This part is used to detect the presence of a device connected to the I2C bus.

You can compile this code to test for the presence of connected and active modules on the I2C bus.

```
\ activates wire vocabulary
wire
\ starts I2C interface using pin 21 and 22 on ESP32 DEVKIT V1
\ with 21 for sda and 22 for scl.
21 22 wire.begin

: spaces (n --)
for
space
next
;
```

```
: .## ( not -- )
<# # # #> type
;

\ not all bitpatterns are valid 7bit i2c addresses
: Wire.7bitaddr? (a-f)
dup $07 >=
swap $77 <= and
;

: Wire.detect(--)
base @ >r hex
cr
"00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f"
$80 $00 do
i $0f and 0=
if
shout .## ." : "
then
i Wire.7bitaddr? if
i Wire.beginTransmission
-1 Wire.endTransmission 0 =
if
i .## space
else
." -- "
then
else
2 spaces
then
loop
cr r> base!
;
```

Here, running the word `Wire.detect` indicates the presence of the OLED display device at hexadecimal address 3C:

```
Wire.detect
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00: -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- 3c -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Here we detected a module at hexadecimal address 3c. This is the address that we will use to address this module..... @TODO: to be completed

Add the SPI library

The SPI library is not natively implemented in ESP32forth. To install it, you must first create the **spi.h file** which must be installed in the same folder as that containing the **ESP42forth.ino** file.

Content of the **spi.h file** (in C language) :

```c
# include <SPI.h>

#define OPTIONAL_SPI_VOCABULARY V(spi)
#define OPTIONAL_SPI_SUPPORT \
  XV(internals, "spi-source", SPI_SOURCE, \
      PUSH spi_source; PUSH sizeof(spi_source) - 1) \
  XV(spi, "SPI.begin", SPI_BEGIN, SPI.begin((int8_t) n3, (int8_t) n2, (int8_t) n1, (int8_t) n0); DROPn(4)) \
  XV(spi, "SPI.end", SPI_END, SPI.end();) \
  XV(spi, "SPI.setHwCs", SPI_SETHWCS, SPI.setHwCs((boolean) n0); DROP) \
  XV(spi, "SPI.setBitOrder", SPI_SETBITORDER, SPI.setBitOrder((uint8_t) n0); DROP) \
  XV(spi, "SPI.setDataMode", SPI_SETDATAMODE, SPI.setDataMode((uint8_t) n0); DROP) \
  XV(spi, "SPI.setFrequency", SPI_SETFREQUENCY, SPI.setFrequency((uint32_t) n0); DROP) \
  XV(spi, "SPI.setClockDivider", SPI_SETCLOCKDIVIDER, SPI.setClockDivider((uint32_t) n0); DROP) \
  XV(spi, "SPI.getClockDivider", SPI_GETCLOCKDIVIDER, PUSH SPI.getClockDivider();) \
  XV(spi, "SPI.transfer",   SPI_TRANSFER, SPI.transfer((uint8_t *) n1, (uint32_t) n0); DROPn(2)) \
  XV(spi, "SPI.transfer8",  SPI_TRANSFER_8,  PUSH (uint8_t)  SPI.transfer((uint8_t) n0); NIP) \
  XV(spi, "SPI.transfer16", SPI_TRANSFER_16, PUSH (uint16_t) SPI.transfer16((uint16_t) n0); NIP) \
  XV(spi, "SPI.transfer32", SPI_TRANSFER_32, PUSH (uint32_t) SPI.transfer32((uint32_t) n0); NIP) \
  XV(spi, "SPI.transferBytes", SPI_TRANSFER_BYTES, SPI.transferBytes((const uint8_t *) n2, (uint8_t *) n1, (uint32_t) n0);
DROPn(3)) \
  XV(spi, "SPI.transferBits", SPI_TRANSFER_BITES, SPI.transferBits((uint32_t) n2, (uint32_t *) n1, (uint8_t) n0); DROPn(3)) \
  XV(spi, "SPI.write", SPI_WRITE, SPI.write((uint8_t) n0); DROP) \
  XV(spi, "SPI.write16", SPI_WRITE16, SPI.write16((uint16_t) n0); DROP) \
  XV(spi, "SPI.write32", SPI_WRITE32, SPI.write32((uint32_t) n0); DROP) \
  XV(spi, "SPI.writeBytes", SPI_WRITE_BYTES, SPI.writeBytes((const uint8_t *) n1, (uint32_t) n0); DROPn(2)) \
  XV(spi, "SPI.writePixels", SPI_WRITE_PIXELS, SPI.writePixels((const void *) n1, (uint32_t) n0); DROPn(2)) \
  XV(spi, "SPI.writePattern", SPI_WRITE_PATTERN, SPI.writePattern((const uint8_t *) n2, (uint8_t) n1, (uint32_t) n0);
DROPn(3))

const char spi_source[] = R"""(
vocabulary spi   spi definitions
transfer spi-builtins
forth definitions
)""";
```

The full file is also available here:

https://github.com/MPETREMANN11/ESP32forth/blob/main/optional/spi.h

Changes to the ESP32forth.ino file

spi.h file cannot be integrated into ESP32forth without making some changes to the **ESP32forth.ino** file. Here are the few modifications to be made to this file. These changes were made on version 7.0.7.15, but should be applicable to other recent or future versions.

First modification

Code added in red :

```
#define VOCABULARY_LIST \
  V(forth) V(internals) \
  V(rtos) V(SPIFFS) V(serial) V(SD) V(SD_MMC) V(ESP) \
  V(ledc) V(Wire) V(WiFi) V(sockets) \
  OPTIONAL_CAMERA_VOCABULARY \
  OPTIONAL_BLUETOOTH_VOCABULARY \
  OPTIONAL_INTERRUPTS_VOCABULARIES \
  OPTIONAL_OLED_VOCABULARY \
  OPTIONAL_SPI_VOCABULARY \
  OPTIONAL_RMT_VOCABULARY \
  OPTIONAL_SPI_FLASH_VOCABULARY \
  USER_VOCABULARIES
```

Second modification

Addition in red after this code :

```
// Hook to pull in optional Oled support.
# if __has_include("oled.h")
#  include "oled.h"
# else
#  define OPTIONAL_OLED_VOCABULARY
#  define OPTIONAL_OLED_SUPPORT
# endif

// Hook to pull in optional SPI support.
# if __has_include("spi.h")
#  include "spi.h"
# else
#  define OPTIONAL_SPI_VOCABULARY
#  define OPTIONAL_SPI_SUPPORT
# endif
```

Third modification

Added in red :

```
#define EXTERNAL_OPTIONAL_MODULE_SUPPORT \
  OPTIONAL_ASSEMBLERS_SUPPORT \
  OPTIONAL_CAMERA_SUPPORT \
  OPTIONAL_INTERRUPTS_SUPPORT \
  OPTIONAL_OLED_SUPPORT \
  OPTIONAL_SPI_SUPPORT \
  OPTIONAL_RMT_SUPPORT \
  OPTIONAL_SERIAL_BLUETOOTH_SUPPORT \
  OPTIONAL_SPI_FLASH_SUPPORT
```

Fourth modification

Addition in red :

```
internals DEFINED? oled-source [IF]
  oled-source evaluate
[THEN] forth

internals DEFINED? spi-source [IF]
  spi-source evaluate
[THEN] forth
```

If you follow these instructions carefully, you will be able to compile ESP32forth with ARDUINO IDE and upload it to the ESP32 board. Once these operations are done, launch the terminal. You need to find the ESP32forth welcome prompt. Type :

```
spinnaker vlist
```

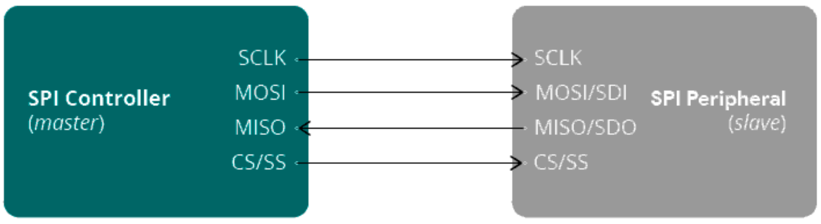
You must find the words defined in this `spi vocabulary` :

```
SPI.begin SPI.end SPI.setHwCs SPI.setBitOrder SPI.setDataMode SPI.setFrequency
SPI.setClockDivider SPI.getClockDivider SPI.transfer SPI.transfer8 SPI.transfer16
SPI.transfer32 SPI.transferBytes SPI.transferBits SPI.write SPI.write16
SPI.write32 SPI.writeBytes SPI.writePixels SPI.writePattern spi-builtins
```

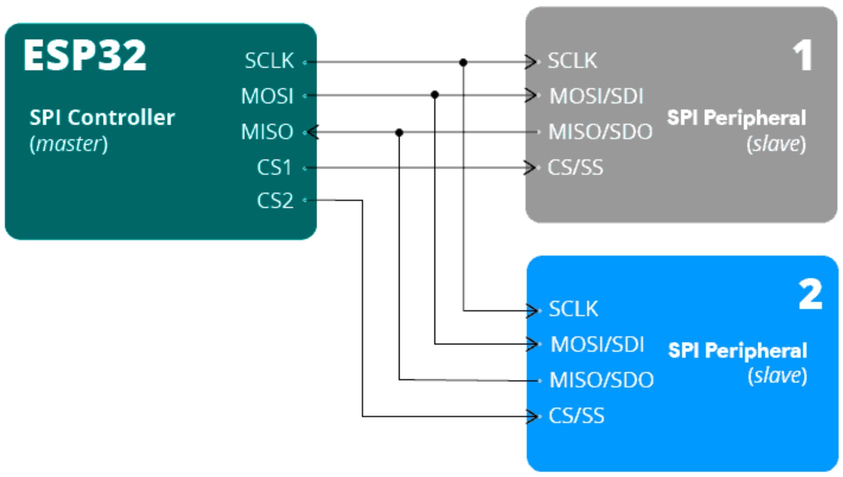
You can now drive extensions via the SPI port, such as the MAX7219 LED displays.

Communicate with the MAX7219 display module

In SPI communication, there is always a master *who* controls the peripherals (also called *slaves*). Data can be sent and received simultaneously. This means that the master can send data to a slave and a slave can send data to the master at the same time.



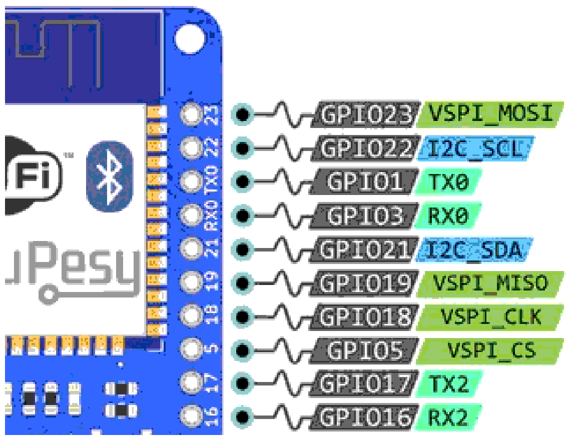
You can have several slaves. A slave can be a sensor, display, microSD card, etc., or another microcontroller. This means you can have your ESP32 connected to **multiple devices** .



A slave is selected by setting the CS1 or CS2 selector to low level. It will take as many CS selectors as there are slaves to manage.

Locating the SPI port on the ESP32 board

There are two SPI ports on an ESP32 board: HSPI and VSPI. The SPI port that we will manage is the one whose pins are prefixed VSPI:



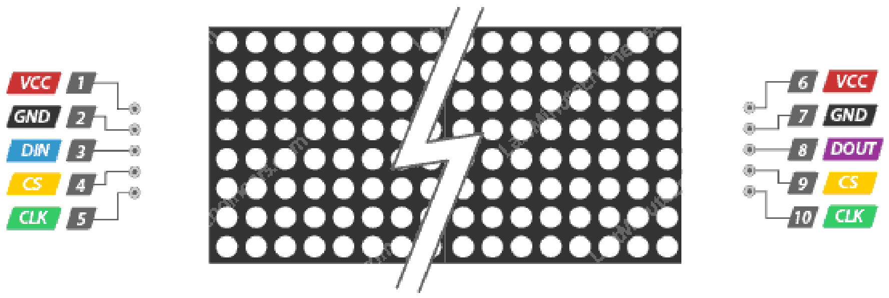
With ESP32forth, we can therefore define the constants pointing to these VSPI pins:

```
\ define VSPI pins
19 constant VSPI_MISO
23 constant VSPI_MOSI
18 constant VSPI_SCLK
05 constant VSPI_CS
```

To communicate to the MAX7219 display module, we will only need to wire the VSPI_MOSI, VSPI_SCLK and VSPI_CS pins.

SPI connectors on the MAX7219 display module

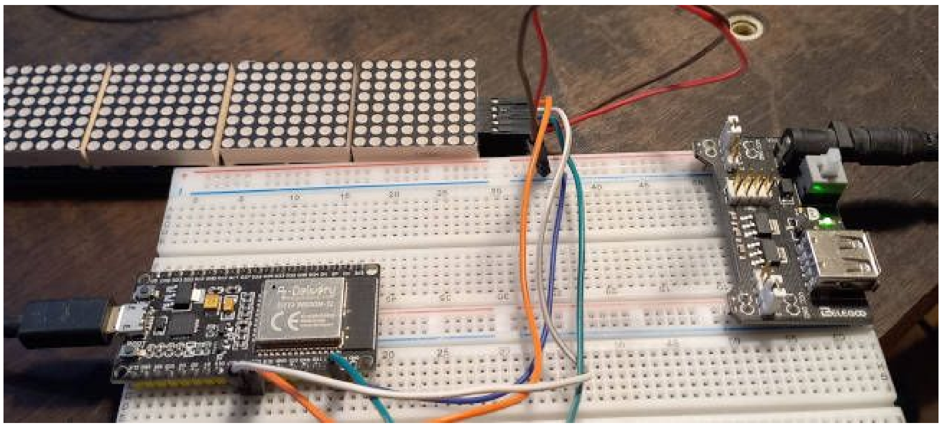
Here is the SPI port connector map on the MAX7219 module:



Connection between the MAX7219 module and the ESP32 card:

```
                MAX7219                 ESP32
                  DIN     <---->    VSPI_MOSI
                  CS      <---->    VSPI_CS
                  CLK     <---->    VSPI_SCLK
```

The VCC and GND connectors are connected to an external power supply:



The GND part of this external power supply is shared with the GND pin of the ESP32 card.

SPI port software layer

All the words for managing the SPI port are already available in the `spi vocabulary` .

The only thing to define is the initialization of the SPI port:

```
\ define SPI port frequency
4000000 constant SPI_FREQ

\ select SPI vocabulary
only FORTH  SPI also

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

We are now ready to use our MAX7219 display module.

Program in XTENSA assembler

Preamble

For those unfamiliar with assembly language, it is the lowest level layer in programming. In assembler, we address the processor directly.

It is also a difficult language, not very readable. But on the other hand, the performance is exceptional.

We program in assembler:

- when there is no other solution to access certain functionalities of a processor;
- to make certain parts of the program faster. Code generated by an assembler is the fastest!
- for fun. Assembler programming is an intellectual challenge;
- because no evolved language can do everything. Sometimes, you can program functions in assembly that are too complex to write in another language.

As an example, here is the Huffman decoding code carried out in XTENSA assembler:

```
/* input in t0, value out in t1, length out in t2 */
    srl t1, t0, 6
    li t3, 3
    beq t3, t4, 2f
    li t2, 2
    andi t3, t0, 0x20
    beq t3, r0, 1f
    li t2, 3
    andi t3, t0, 0x10
    beq t3, r0, 1f
    li t2, 4
    andi t3, t0, 0x08
    beq t3, r0, 1f
    li t2, 5
    andi t3, t0, 0x04
    beq t3, r0, 1f
    li t2, 6
    andi t3, t0, 0x02
    beq t3, r0, 1f
    li t2, 7
    andi t3, t0, 0x01
    beq t3, r0, 1f
    li t2, 8
    b 2f
    li t1, 9
1: /* length = value */
    move t1, t2
2: /* done *
```

Since version 7.0.7.4, ESP32forth includes a complete XTENSA assembler. This assembler uses infix notation:

```
\ in conventional assembler:
\   andi t3, t0, 0x01

\ in XTENSA assembler with ESP32forth:
    a3 a0 $01 ANDI,
```

ESP32forth is the **very first high-level programming language** for ESP32 that integrates an XTENSA assembler.

This feature allows the programmer to define his assembly macros.

Any word written in XTENSA assembly language from ESP32forth is immediately usable in any definition in FORTH language.

Compile the XTENSA assembler

Since version 7.0.7.15, ESP32forth offers the XTENSA assembler as an option. To compile this option :

- **optional** folder in the folder where you unzipped the ZIP file of the ESP32forth version;
- **assemblers.h** file to the root folder containing the **ESP32forth.ino file**;
- run ARDUINO IDE, compile **ESP32forth.ino** and upload to ESP32 board;

If everything went well, you access the XTENSA assembler by typing once :

```
xtensa-assembler
```

To check the correct availability of the XTENSA instruction set:

```
assemble xtensa vlist
```

Programming in assembler

In order to clearly understand what was stated previously, here is a definition proposed as an example by Brad NELSON :

```
\ example proposed by Brad NELSON
code my2*
    a1 32 ENTRY,
    a8 a2 0 L32I.N,
    a8 a8 1 SLLI,
    a8 a2 0 S32I.N,
    RETW.N,
end-code
```

We have just defined the word **my2*** which has exactly the same action as the word **2*** . Assembling the code is immediate. We can therefore test our definition of **my2*** from the terminal :

```
--> 3 my2*
OK
6 --> 21 my2*
OK
6 42 -->
```

This possibility of immediately testing an assembled code allows it to be tested in situ. If we have to write somewhat complex code, it will be easy to cut it into fragments and test each part of this code from the ESP32forth interpreter.

The XTENSA assembly code is placed after the word to be defined. It is the code sequence **my2*** which creates the word **my2*** .

The following lines contain the XTENSA assembly code. The assembly definition ends with the execution of **end-code**.

Summary of basic instructions

List of basic instructions included in all versions of the Xtensa architecture. The remainder of this section provides an overview of the basic instructions.

Load / loading

```
L8UI, L16SI, L16UI, L32I, L32R,
```

Store / storage

```
S8I, S16I, S32I,
```

Memory ordering

```
MEMW, EXTW,
```

Jumps

```
CALL0, CALLX0, RET, J, JX,
```

Conditional branching

```
BALL, BNALL, BANY, BNONE, BBC, BBCI, BBS, BBSI, BEQ, BEQI, BEQZ,
BNE,
BNEI, BNEZ,BGE, BGEI, BGEU, BGEUI, BGEZ, BLT, BLTI, BLTU, BLTUI,
BLTZ,
```

Shift

```
MOVI, MOVEQZ, MOVGEZ, MOVLTZ, MOVNEZ,
```

Arithmetic

```
ADDMI, ADD, ADDX2, ADDX4, ADDX8, SUB, SUBX2, SUBX4, SUBX8, NEG, ABS,
```

Binary logic

```
AND, OR,
```

Shift

```
EXTUI, SRLI, SRAI, SLLI, SRC, SLL, SRL, SRA, SSL, SSR, SSAI, SSA8B,
SSA8L,
```

Processor control

```
RSR, WSR, XSR, RUR, WUR, ISYNC, RSYNC, ESYNC, DSYNC, NOP,
```

A bonus disassembler

An assembler is very good. Easy code to integrate with FORTH definitions is wonderful. But having an XTENSA disassembler is royal!

Let's take the definition of **my2*** previously assembled. It is easy to disassemble :

```
' my2* cell+ @ 20 disasm
\ displays:
\ 1074338656  --  a1 32 ENTRY,            -- 004136
\ 1074338659  --  a8 a2 0 L32I.N,                 -- 0288
\ 1074338661  --  a8 a8 1 SLLI,           -- 1188F0
\ 1074338664  --  a8 a2 0 S32I.N,                 -- 0289
\ 1074338666  --  RETW.N,                 -- F01D
\ 1074338668  -- ......
```

The code of our word **my2*** is only accessible by indirection, the address of which is placed in the parameters field.

Each line displays :

- the address of the assembled code
- the disassembled code at this address on 2 or 3 bytes

- the hexadecimal code corresponding to the disassembled code

The disassembler can also act on all code already compiled or assembled. Let's see the code for word **2*** :

```
' 2* @ 20 disasm
\ displays:
\ 1074606252  --  a12 a3 0 L32I.N,                 -- 03C8
\ 1074606254  --  a5 a5 1 SLLI,            -- 1155F0
\ 1074606257  --  a15 a12 0 L32I.N,                -- 0CF8
\ 1074606259  --  a3 a3 4 ADDI.N,                  -- 334B
\ 1074606261  --  1074597318 J,            -- F74346
```

Disassembly indicates that the code leads to an unconditional jump **1074597318 J** ,. It is easy to continue disassembly to this new address :

```
1074597318 20 disasm
\ display:
\ 1074597318  --  a15 JX,                  -- 000FA0
\ 1074597321  --  a10 64672 L32R,                  -- FCA0A1
\ 1074597324  --  a5 a7 1 S32I,            -- 016752
\ 1074597327  --  1074633168 CALL8,                -- 08C025
\ 1074597330  --  a12 a3 0 L32I,           -- 0023C2
\ 1074597333  --  a2 a7 4 ADDI,            -- 04C722
\ 1074597336  ......
```

First steps in XTENSA assembler

Preamble

The assembly code is not portable in another environment, or at the cost of enormous efforts to understand and adapt the assembled code.

A FORTH version is not complete if it does not have an assembler.

Assembler programming is not required. But in some cases, creating a definition in assembler can be much easier than a version in C language or in pure FORTH language.

But above all, a definition written in assembler will have unrivaled speed of execution.

We will see, using very simple and very short examples, how to master the programming of FORTH definitions written in Xtensa assembler.

Invoking the Xtensa assembler

When starting ESP32forth, it is impossible to define words in Xtensa assembly without invoking the word `xtensa-assembler` . This word will load the content of the `xtensa vocabulary`. This word must only be invoked once when starting ESP32forth and before any definition of a word in xtensa code :

```
forth
DEFINED? invert code [IF] xtensa-assembler [THEN]
```

Now, if we type `order` , ESP32forth displays :

```
xtensa >> asm >> FORTH
```

It is this order of vocabularies that must be respected when we want to define a new word in Xtensa assembly using the definition words `code` and `end-code` .

Xtensa and the FORTH stack

The Xtensa processor has 16 registers, a0 to a15. In reality, there are 64 registers, but we can only access a window of 16 registers among these 64 registers, accessible in the interval 00..15.

Register a2 contains the FORTH stack pointer.

Each time a value is stacked, the stack pointer is incremented by four units:

```
SP@ .   \ displays 1073632236
1
SP@ .   \ displays 1073632240
2
SP@ .   \ displays 1073632244
drop drop
SP@ .   \ 1073632236
```

Here is how we could rewrite this word **SP@** in Xtensa assembler :

```
\ get SP Stack Pointer - equivalent to SP@
code mySP@
    a1 32       ENTRY,
    a8 a2       MOV.N,   \ copy contents of a2 into a8
    a2 a2 4     ADDI,    \ increment a2
    a8 a2 0     S32I.N,  \ copy a8 into address pointed to by a2+0
                RETW.N,
end-code
```

Let's test this new word **mySP@** :

```
mySP@.
\ displays 1073632240
SP@.
\ displays 1073632240
```

Writing an Xtensa macro instruction

In our definition of the word **mySP@** , the sequence **a2 a2 4 ADDI,** increments the stack pointer by four units. Without this increment, it is impossible to return a value to the top of the FORTH stack. With FORTH, we will write a macro that automates this operation.

To start, we'll expand the **asm** vocabulary :

```
asm definitions

: macro:
    :
  ;
```

Our **macro definition:** is redundant with **:** but has the advantage of then making the FORTH code a little more readable when we define a macro-instruction which will extend the **xtensa vocabulary** :

```
xtensa definitions

macro: sp++,
    a2 a2 4    ADDI,
  ;
```

sp++ macro instruction , we can rewrite the definition of **mySP@** :

```
forth definitions
```

```
asm xtensa

   \ get Stack Pointer SP - equivalent for SP@
code mySP@
    a1 32       ENTRY,
    a8 a2       MOV.N,  \ copy content of a2 in a8
        sp++,
    a8 a2 0     S32I.N, \ copy a8 in address pointed by a2+0
                RETW.N,
end-code
```

It is perfectly possible to integrate one macro into another. In the `mySP@` code , the code line `a8 a2 0 S32I.N,` copies the contents of register a8 to the address pointed to by a2. Here is this new macro instruction :

```
xtensa definitions

\ increment Stack Pointer and store content of ar in addr pointed by
Stack Pointer
macro: arPUSH, { ar -- }
    sp++,
    ar a2 0 S32I.N,
  ;
```

This macro instruction uses a local variable `ar` . We could have done without it, but the advantage of this variable is that the macro code is more readable.

`mySP@` code with this macro-instruction :

```
forth definitions
asm xtensa

\ get Stack Pointer SP - equivalent to SP@
code mySP@3
    a1 32       ENTRY,
    a8 a2       MOV.N,
    a8  arPUSH,
                RETW.N,
end-code
```

Let's complete our list of macro instructions :

```
xtensa definitions

\ décrémente pointeur de pile
macro: sp--,    ( -- )
    a2 a2 -4    ADDI,
  ;

\ Store content of addr pointed by Stack Pointer in ar and decrement
Stack Pointer
macro: arPOP,   { ar -- }
    ar a2 0     L32I.N,
    sp--,
  ;
```

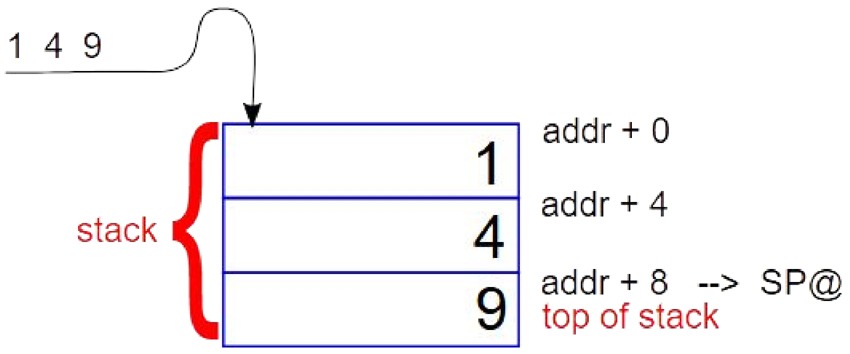
With these new macros, let's rewrite **swap** :

```
forth definitions
asm xtensa

code mySWAP
    a1 32       ENTRY,
    a9  arPOP,
    a8  arPOP,
    a9  arPUSH,
    a8  arPUSH,
                RETW.N,
end-code

17 24 mySWAP
```

Managing the FORTH stack in Xtensa assembler

The position of the FORTH stack pointer can be accessed by **SP@** . Stacking a 32-bit integer (default size for ESP32forth) increments this stack pointer by four units.

We discussed how to manage the increment or decrement of this stack pointer through the **sp++** , and **sp--** , macro-instructions. These macro instructions move the stack pointer four units.



Here, we have stacked three values, **1 4** and **9** . Each time you stack, the stack pointer is incremented automatically. In Xtensa assembler, the stack pointer is found in register a2. We have seen that we can manipulate the contents of this register with the macro-instructions **sp++** , and **sp--** ,. Manipulating this register has a direct action on the stack pointer managed by ESP32forth.

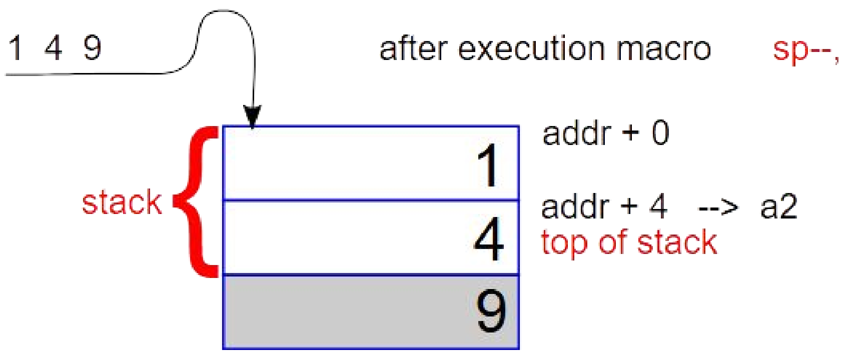
Here is how we rewrote the word **+** in assembly by manipulating the stack pointer through our **arPOP,** and **arPUSH,** macro-instructions :

```
code my+
    a1 32       ENTRY,
    a7  arPOP,
    a8  arPOP,
    a7 a8 a9    ADD,
    a9  arPUSH,
            RETW.N,
end-code
```

There is another way to retrieve data from the stack using the **L32I.N,** instruction . This instruction uses an immediate index :

```
code my+
    a1 32 ENTRY,
        sp--,
    a7 a2 0     L32I.N,
    a8 a2 1     L32I.N,
    a7 a8 a9    ADD,
    a9 a2 0     S32I.N,
    RETW.N,
end-code
```

Before retrieving the data from the stack, we decrement the stack pointer with our macro instruction **sp--** ,. In this way, the pointer moves back 4 units.



But just because the pointer moves back doesn't mean the previously stacked data disappears. Let's see this line of code in detail:

```
a7 a2 0 L32I.N,
```

This instruction loads register a7 with the contents of the address pointed to by (a2)+n*4. Here, n is 0. This instruction will put the value 4 in our register a7.

Let's see the following line:

```
a8 a2 1 L32I.N,
```

Register a8 is loaded with the contents pointed to by (a2)+1*4. This instruction puts the value 9 in our register a8.

```
a9 a2 0 S32I.N,
```

Here, the contents of register a9 are stored at the address pointed by (a2)+1*0. In fact, we overwrite the value 4 with the result of adding the contents of registers a7 and a8.

Let's see one last example where we process two parameters and output two of them onto the data stack. In this example, we rewrite the word **/MOD** :

```
code my/MOD ( n1 n2 -- rem quot )
    a1 32       ENTRY,
    a7  arPOP,          \divisor in a7
    a8  arPOP,          \ value to divide in a8
    a7 a8 a9    REMS,   \a9 = a8 MOD a7
    a9  arPUSH,
    a7 a8 a9    QUOS,   \a9 = a8 / a7
    a9  arPUSH,
            RETW.N,
end-code

5 2 my/MOD . .      \ display 2 1
-5 -2 my/MOD . .    \ display 2 -1
```

In the word **my/MOD** we use the same data n1 and n2 placed respectively in registers a8 and a7. It is then the **REMS** and **QUOT** instructions which allow the results returned by **my/MOD** to be calculated.

Efficiency of words written in XTENSA assembler

In our very last example above, we rewrote the word **/MOD** . The question to ask is: "is the word **my/MOD** really faster in execution than the word **/MOD** ?".

To do this, we will use the word **measure:** whose FORTH code is explained in the chapter *Measuring the execution time of a FORTH word* .

```
: test1
    1000000 for
        5 2 /MOD
        drop drop
    next
  ;

: test2
    1000000 for
        5 2 my/MOD
        drop drop
    next
  ;

measure: test1  \ display: execution time: 0.856sec.
measure: test2  \ display: execution time: 0.600sec.
```

The words `test1` and `test2` are similar, except that `test2` executes `my/MOD` . Over 1 million iterations, the time saving amounts to 0.144 seconds. It's not much, but the ratio still seems significant.

Conversely, we see that the FORTH language is very fast in execution time.

Loops and connections in XTENSA assembler

The LOOP instruction in XTENSA assembler

The LOOP loop in XTENSA assembler works by using the **LOOP instruction** to tell the processor to repeat a block of instructions until a specified counter reaches zero. The loop is initialized by setting the initial value of the counter, then executing the **LOOP instruction** with that value as an argument. On each iteration of the loop, the counter is decremented by 1 until it reaches zero, at which point the loop stops. In classic assembler:

```
    ; Initialization of the counter to 10
    MOVI a0, 10

    ; Beginning of the LOOP loop
loop:
    ; Instruction(s) to repeat
    ...
    ; Decrement the counter and test the stop condition
    LOOP a0, loop
```

Here, the LOOP loop repeats the instructions between `loop:` and `LOOP a0,` looping 10 times, decrementing the counter a0 with each iteration. When the counter reaches zero, the loop stops.

When the XTENSA processor encounters the **LOOP** instruction , it initializes three special registers :

- **LCOUNT ← AR[s] – 1**

  The special register LCOUNT is initialized with the contents of the register as, here a0 in our example, decremented by one unit. When the counter reaches the value 0, the LOOP instruction completes the loop;

- **LBEG ← PC + 3**

  The LBEG special register contains the start address of the currently executing LOOP loop. This address is defined by the LOOP instruction.

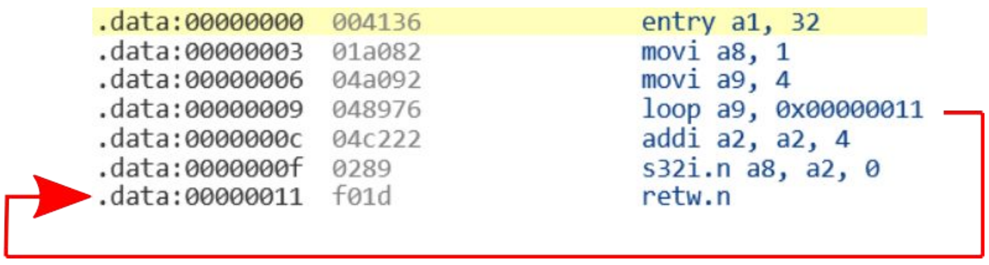
- **LEND ← PC + (024||imm8) + 4** The LEND special register contains the end address of the currently executing LOOP loop. This address is defined by the LOOP instruction.

In XTENSA assembler, the LOOP instruction admits two parameters :

```
  LOOP as, label
```

`label` corresponds to an 8-bit offset after the `LOOP instruction` . You cannot repeat a code of more than 256 bytes in length.

Here is some disassembled XTENSA code using a LOOP loop :



```
.data:00000000  004136          entry a1, 32
.data:00000003  01a082          movi a8, 1
.data:00000006  04a092          movi a9, 4
.data:00000009  048976          loop a9, 0x00000011
.data:0000000c  04c222          addi a2, a2, 4
.data:0000000f  0289            s32i.n a8, a2, 0
.data:00000011  f01d            retw.n
```

The disassembler indicates a branch address. In reality, the assembled code only contains this offset indicated by the label in the form of a positive 8-bit value.

Manage a loop in XTENSA assembler with ESP32forth

The FORTH language cannot resolve a forward reference. Unless you fumble around, it is difficult to use the `LOOP` instruction without finding a trick.

Defining loop management macro instructions

`LOOP,` instruction , we will define two macro instructions, respectively `For,` and `Next,` of which here is the code in FORTH language:

```
: For, { as n -- }
    as n MOVI,
    as 0 LOOP,
    chere 1- to LOOP_OFFSET
  ;

: Next, ( -- )
    chere LOOP_OFFSET - 2 -
    LOOP_OFFSET [ internals ] ca! [ asm xtensa ]
  ;
```

The `For` macro instruction accepts the same parameters as the `LOOP instruction` :

```
as n For,
```

- as is the register that contains the number of iterations of the loop;
- n is the number of iterations.

Using the For, and Next macros,

We define a **myLOOP** word to test the LOOP instruction **,** via the **For, Next,** macro instructions :

```
code myLOOP ( n -- n' )
    a1 32           ENTRY,
    a8 1            MOVI,
    a9 4        For,            \ LOOP start here
        a8 a8 1     ADDI,
        a8      arPUSH,         \ push result on stack
    Next,
                    RETW.N,
end-code
```

Register a8 is initialized with the value 1. The **For, Next loop** increments the contents of a8 and stacks its contents. This is what running **MyLOOP gives** :

```
OK
--> myLoop
OK
2 3 4 5 -->
```

ATTENTION : if the number of iterations is zero, the number of iterations increases to 232.

Connection instructions in XTENSA assembler

The XTENSA assembler in the **xtensa** vocabulary has several types of branch instructions :

- connections using Boolean flags defined in the special register **BR** : **BF, BT,**
- the connections carrying out tests on the registers: **BALL, BANY, BBC, BBS, BEQ, BGE, BLT, BNE, BNONE,**

It is this second category of connections that interests us.

Defining branching macros

The ESP32forth xtensa assembler does not have a label management mechanism as is the case for a classic assembler. To be effective, label management must work in several stages if forward branches need to be resolved. This is incompatible with the operation of the FORTH language which compiles or assembles in a single pass.

We overcome this difficulty by defining two macro instructions, **If,** and **Then,** which will manage these forward connections:

```
: If, ( -- BRANCH_OFFSET )
    chere 1-
  ;

: Then, { BRANCH_OFFSET -- }
    chere BRANCH_OFFSET - 2 -
    BRANCH_OFFSET [ internals ] ca! [ asm xtensa ]
  ;
```

The macro instruction must be preceded by another macro instruction. For our first test, we define the macro `<,` which will assemble an unresolved branch:

```
: <,  ( as at -- )
    0 BGE,
  ;
```

Using these macros in our first example:

```
code my< ( n1 n2 -- fl )    \ fl=1 if n1 < n2
    a1 32           ENTRY,
    a8          arPOP,              \ a8 = n2
    a9          arPOP,              \ a9 = n1
    a7 0            MOVI,           \ a7 = 1
    a8 a9 <, If,
      a7 1          MOVI,           \ a7 = 0
    Then,
    a7          arPUSH,
                    RETW.N,
end-code
```

Syntax of branching macro instructions

In our example, we used the macro instruction `<,` which is associated with the BGE branch instruction `,` and whose meaning is: "Branch if Greater Than or Equal". Normally, it would be translated by ">=". Why was "<" used?

This is because our macro instruction `If, .... Then,` has a logic opposite to that of the branch to be carried out. The code enclosed in `If, .... Then,` will execute if the required condition is not valid. Here is the table which summarizes this reversed logic explaining the choice of the name of these macro instructions used before `If, .... Then` ,:

XTENSA branch instruction			Macro
BEQ	Branch if Equal	AR[s] = AR[t]	**<>,**
BGE	Branch if Greater Than or Equal	AR[s] ≥ AR[t]	**<,**
BLT	Branch if Less Than	AR[s] < AR[t]	**>=,**
BNE	Branch if Not Equal	AR[s] ≠ AR[t]	**=,**

`my<` assembly example . Here is what the execution of the word `my< gives` :

```
10 20 my< .      \ displays: 1
```

```
20 20 my< .     \ displays: 0
20 10 my< .     \ displays: 0
-5 35 my< .     \ displays: 1
-10 -3 my< .    \ displays: 1
-3 -10 my< .    \ displays: 0
```

We see that this reversed logic is respected.

Once this logic is understood, we can define a new macro-instruction **>=** ,:

```
: >=,  ( as at -- )
    0 BLT,
  ;
```

And test this macro-instruction :

```
code my>= ( n1 n2 -- fl )    \ fl=1 if n1 < n2
    a1 32           ENTRY,
    a8          arPOP,              \ a8 = n2
    a9          arPOP,              \ a9 = n1
    a7 0            MOVI,           \ a7 = 1
    a8 a9 >=, If,
        a7 1        MOVI,           \ a7 = 0
    Then,
    a7          arPUSH,
                    RETW.N,
end-code

10 20 my>= .        \ displays: 0
20 20 my>= .        \ displays: 1
20 10 my>= .        \ displays: 1
-5 35 my>= .        \ displays: 0
-10 -3 my>= .       \ displays: 0
-3 -10 my>= .       \ displays: 1
```

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