The great book

for MECRISP Forth

version 1.0 - 25 décembre 2023

Author

- Marc PETREMANN

Content

Install MECRISP on the RP pico card

Preparing the RP PICO card

Unpack and install the RP PICO board on a breadboard.

Figure 1: connection to PC via USB cable

Then connect a USB cable to the RP PICO card side:

On the RP PICO board there is a push button. Hold this button and plug the cord into the PC at the same time. In WINDOWS 11, this action immediately opens a window in File

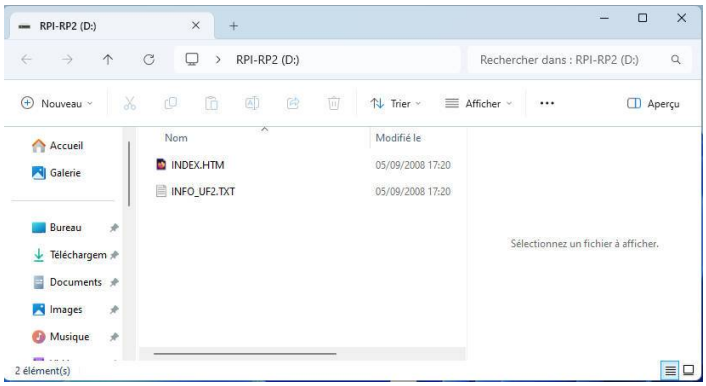


Figure 2: storage space in the RP pico card

Explorer:

Downloading and installing MECRISP on RP PICO

You can download MECRISP for RP PICO from this link:
https://mecrisp.arduino-forth.com/public/fichiers/mecrisp-stellaris-pico-with-tools.uf2

Once the file is downloaded, open the download folder.

Select the **mecrisp-stellaris-pico-with-tools.uf2 file** .

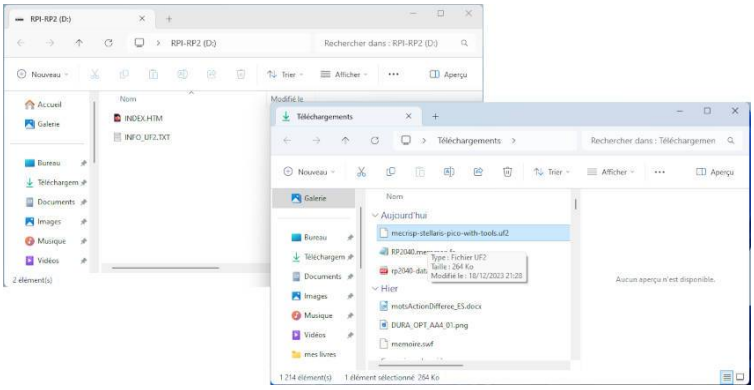


Figure 3: copy of UF2 extension file to the RP pico card space

Drag and drop this **mecrisp-stellaris-pico-with-tools.uf2 file** to the RP PICO board folder.

After the file is copied, the RP PICO files window closes. METRISP for RP PICO is installed.

Plug in and power the RP Pico board

MECRISP Forth is not accessible via the mini USB connector of the RP pico card which was used to install MECRISP Forth.

To be able to communicate with MECRSIP Forth, it is necessary to carry out some installations.

Power the RP pico card

Figure 4: battery holder on AMAZON

The choice fell on a power interface of this type:

Here, on AMAZON, the 5 units cost less than €30. It is indicated as a battery holder, but works perfectly with lithium batteries in 18650 format. These lithium batteries offer very long autonomy.

The battery holder of this type has:

- a standard USB output

- a mini USB input to allow the battery to be recharged from a conventional USB charger
- a mini switch cutting power to the standard USB output
- three 5V outputs
- three 3V3 outputs

The 3V3 and 5V outputs are not interrupted by the mini switch. There is no fuse **protection .** I therefore advise being careful when soldering connection wires to the 3V3 or 5V outputs. Carry out these operations without battery or connection to the electronic card.

Here, I soldered two *dupont wires* to a 3V3 output:

- red wire on 3V output

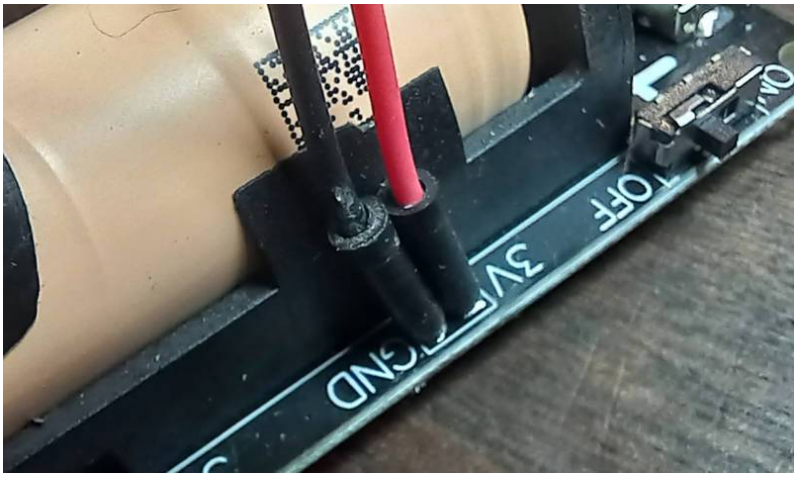


Figure 5: soldering wires to a 3V3 output

- black wire on type GND

In the photo, you can clearly see the mini switch. At the risk of insisting, this switch **does not cut the 3VE and 5V outputs** .

The connection of the black and red wires coming from the power supply must be connected to the RP pico card at pins 38 and 39:

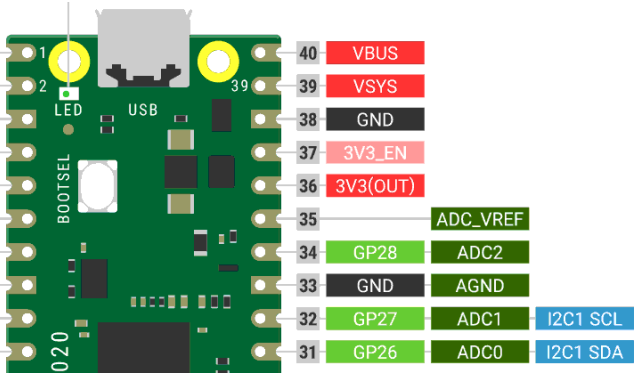


Figure 6: pins 38 and 39 to 3V3 power supply

The power supply must be connected to the RP pico card via a breadboard:

- **red wire** first to connect to pin 39. This precaution is essential, because if the plug accidentally touches another pin, it will have no consequences for the RP pico card;
- **black wire** then connect to pin 38.

To interrupt power to the RP pico board, disconnect the black wire from pin 38.

Connect the UART0 serial port

The connection between the RP pico card and the PC is made via this interface which is a TTL <--> USB adapter.

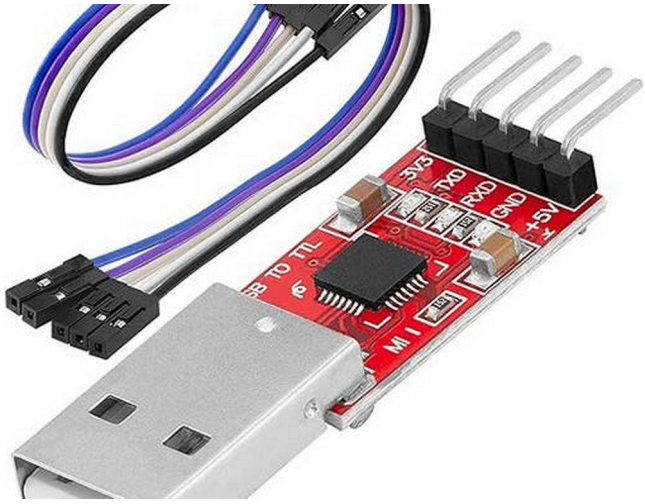


Figure 7: TTL <-> USB adapter

On this adapter there are 5 outputs:

- 2 outputs: 3V3 and 5V which we will not use. The risk of accidentally damaging the PC will thus be reduced to a minimum;
- 2 TXD and RXD input/output pins. These are the outputs that we will exploit;
- a GND output which must also be used.

Here are the connections to be made between the RP pico card and this interface:

- GND --> pin 3 (GND)
- TXD --> pin 2 (UART0 RX)

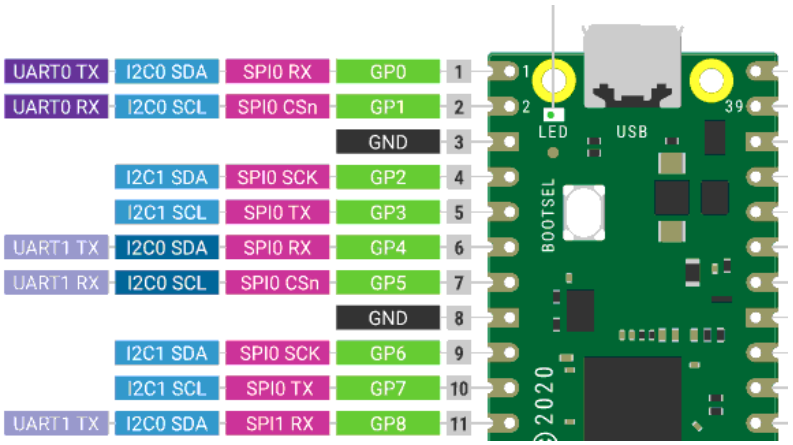


Figure 8: pins 1 à 3 vers adaptateur USB / TTL

- RXD --> pin 1 (UART0 TX)

You can now connect the adapter to the PC and power up the RP pico board via battery:

Communicate with the RP pico card

To communicate the RP pico card, you must use a TERMINAL program. I recommend TERATERM if you are on Windows.

Launch TERATERM. The parameters to communicate with the RP PICO card under eForth PICO ICE are as follows:

- PORT: USB com port available
- Speed: 115200
- Data: 8 bit

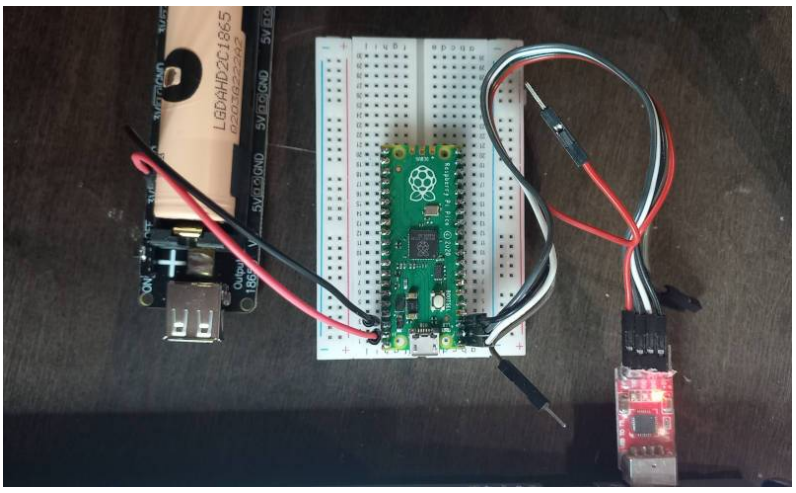


Figure 9: connecting RP pico card to PC

- Parity: none
- Stop bits: 1 bit
- Flow control: none

If the serial link is well established, the RP pico card should respond to pressing the **ENTER** key on the PC keyboard.

Test the good responsiveness of MECRISP¨by typing the `list` command . This command will display the contents of the FORTH dictionary.

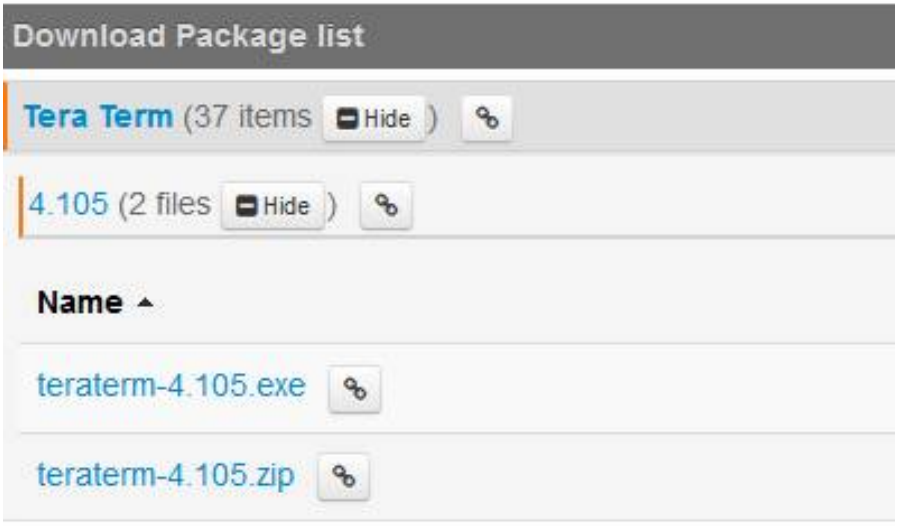
Install and use the Tera Term terminal on Windows

Install Tera Term

The English page for Tera Term is here:

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

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

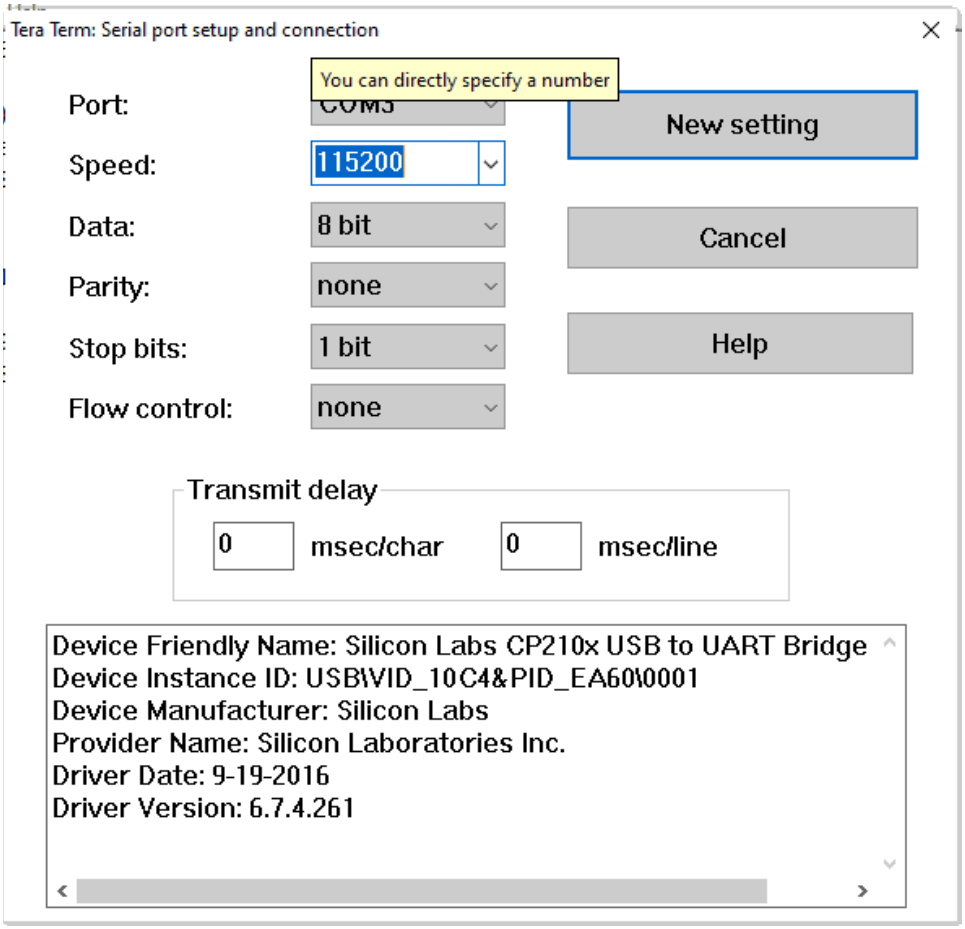


Install Tera Term. Installation is quick and easy.

Setting up Tera Term

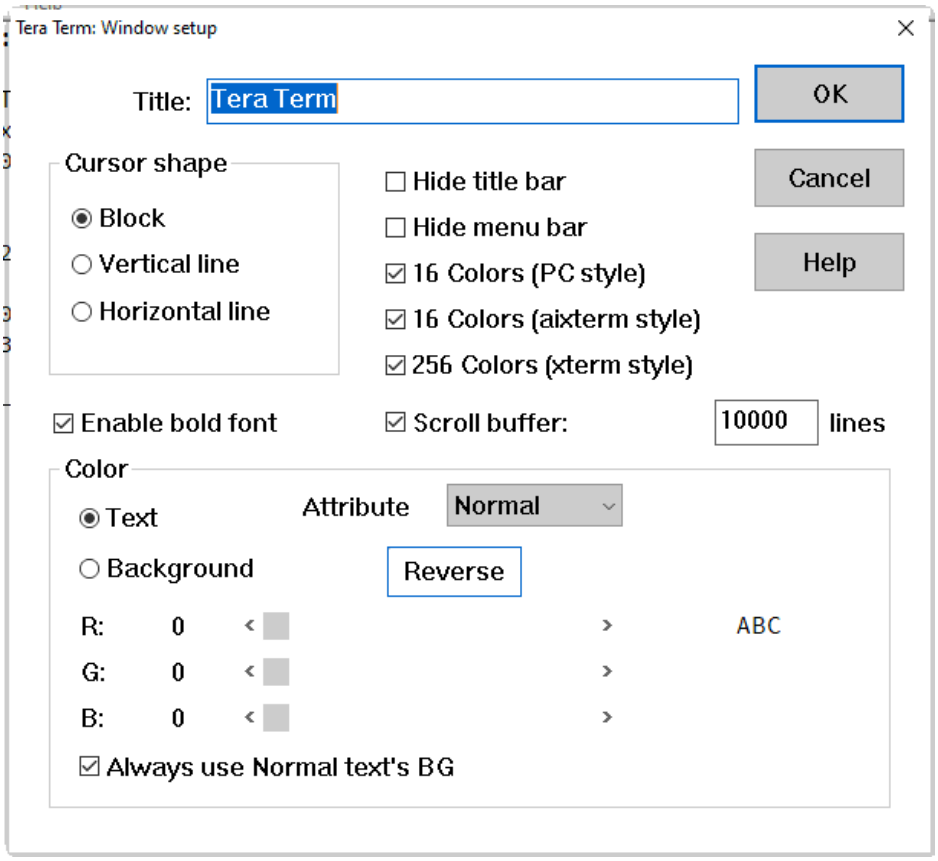
To communicate with the RP pico card, you must adjust certain parameters:

- click on Configuration -> serial port



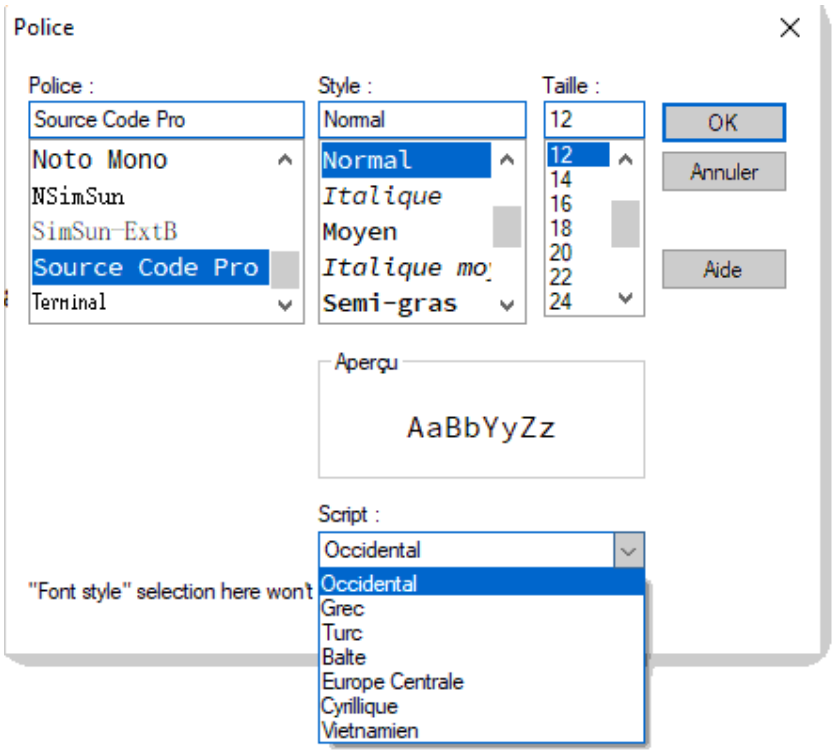
For comfortable viewing:

- click on Configuration -> window



For readable characters:

- click on Configuration -> font



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

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

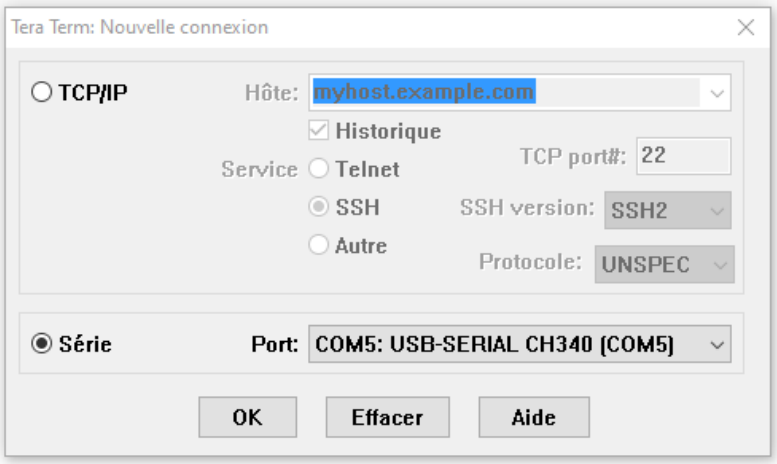
Using Tera Term

Once configured, close Tera Term.

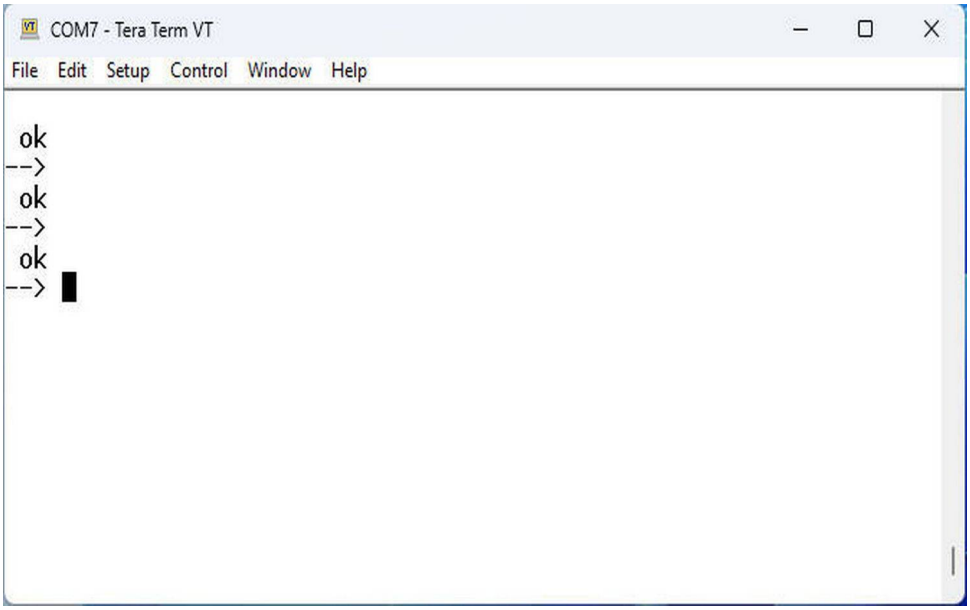
Connect your RP pico board to an available USB port on your PC.

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

Select the serial port :



If everything went well, you should see this:

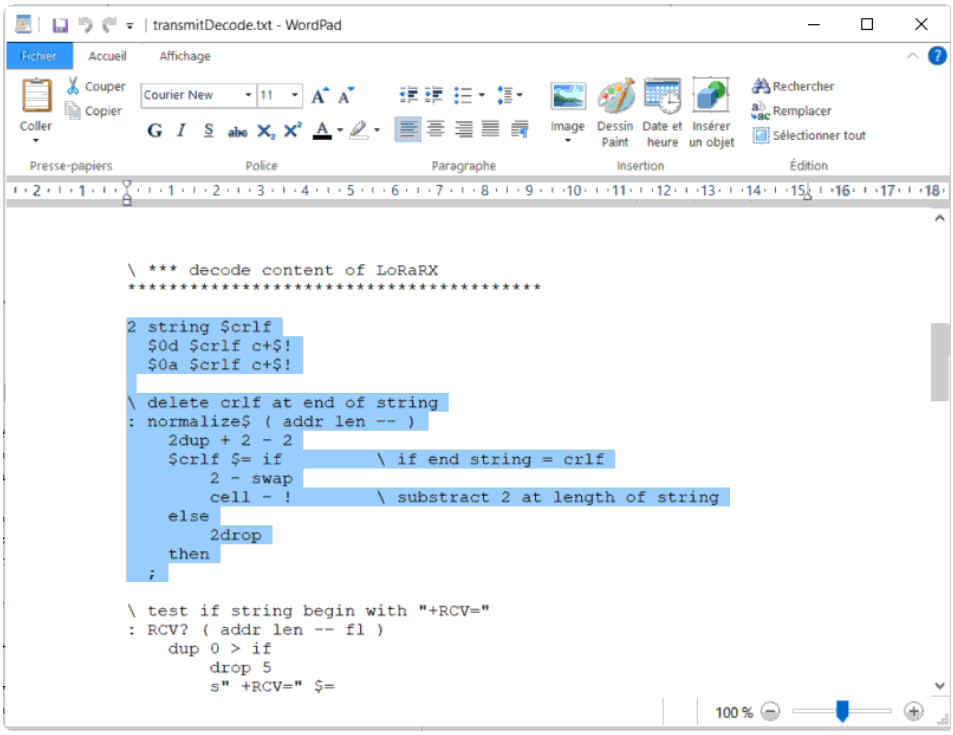


Compile source code in Forth language

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

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

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



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

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

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

Click on the Tera Term terminal window. Make Paste:

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

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

Getting started with GPIO ports

All apprentice programmers are eager to test their card. The most trivial example is flashing an LED. It turns out that there is an LED already mounted on the RP pico card,

LED associated with PIO port 25. Here is the code that will allow you to flash this LED with MECRISP Forth:

```
: blink ( -- )
    5 $400140CC !
    $02000000 $D0000024 !
    begin
        $02000000 $D000001C !
        300 ms
    key? until
  ;
```

You can copy this code and transmit it to the RP pico card via the TeraTem terminal. It will work.

Once this was tested, nothing was explained. Because if you want to master input/output ports, you have to start by understanding what this code is supposed to do.

GPIO ports

The Raspberry Pi Pico has 30 GPIO pins, of which 26 are usable.

- 2x SPI
- 2 x UARTs
- 2 x I2C
- 8 x two-channel PWM

there is also :

- 4 x general purpose clock output
- 4 x ADC input
- PIO on all pins

The card has an onboard LED, connected to the GPIO 25. It allows you to check the correct operation of the card or any other use at your convenience.

For the remainder of this chapter, we will only see the case of the LED installed on the RP pico card and connected to the GPIO 25.

GPIO and registers

The RP pico card has an RP2040 microcontroller, the technical data of which is accessible in this document in pdf format:
https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

The content of this document is very technical and may put off an amateur. We will return to our example given at the start of the chapter to explain step by step what makes it work, by associating this information with the content of the "RP2040 datasheet" document.

Let's start by explaining what a microcontroller register is.

A register is quite simply a memory area accessible by an address, but whose use is reserved for the operation of the microcontroller. If we write anything in a register, at best we cause malfunctions, at worst, we block the microcontroller. If this happens, you will need to power off the RP pico card to reset it.

MECRISP Forth accesses 32-bit addresses with words **!** And **@** . Example :

```
variable myScore
3215 myScore !
myScore @ .    \ display 3215
```

Here, we define a **myScore** variable. When we type the name of this variable, we actually stack the memory address pointing to the contents of this variable:

```
myScore  .    \ display adress of myScore
```

Manipulation of data to this address:

- **3215 myScore !** stores a value at the memory address reserved by **myScore**
- **myScore @** retrieves content stored at the memory address reserved by **myScore**

If the address placed on the data stack by **myScore** is addr (for example **$20013D48**), we can replace **myScore @** by **addr @**.

To access data from a registry, it's not much different. Let's take the case of this register:

```
: blink ( -- )
    5 $400140CC !
    $02000000 $D0000024 !
    begin
        $02000000 $D000001C !
        300 ms
    key? until
  ;
```

Here its memory address is **$D0000024**. The sequence **$02000000 $D0000024 !** Simply stores the value **$02000000** in memory address **$D0000024**.

In the "RP2040 datasheet" document, this address has the label **GPIO_OE_SET**.

0x020	GPIO_OE	GPIO output enable
0x024	GPIO_OE_SET	GPIO output enable set
0x028	GPIO_OE_CLR	GPIO output enable clear
0x02c	GPIO_OE_XOR	GPIO output enable XOR

Figure 10: extract from technical document RP2040

We can therefore make our FORTH code a little more readable by defining this label as a FORTH constant:

```
$d0000024 constant GPIO_OE_SET
: blink ( -- )
    5 $400140CC !
    $02000000 GPIO_OE_SET !
    begin
        $02000000 $D000001C !
        300 ms
    key? until
  ;
```

With practice, we learn to decipher the labels. Here, `GPIO_OE_SET` almost means " **GPIO** "Output **Enable SET** " (Valid GPIO Output Position).

Use registry labels

The `GPIO_OE_SET` label is itself a subset of the registers defined in the global `SIO_BASE` label . Moreover, in the technical documentation, in the first column mentioning the `GPIO_OE_SET` label we find the offset relative to `SIO_BASE`. We will take this into account to rewrite part of the FORTH code. We take advantage of this to define two other labels as constants :

```
$d0000000 constant SIO_BASE
SIO_BASE $020 + constant GPIO_OE
SIO_BASE $024 + constant GPIO_OE_SET
SIO_BASE $028 + constant GPIO_OE_CLR
```

At this stage, our FORTH code still remains dependent on GPIO25 through this mask:

```
: blink ( -- )
    5 $400140CC !
    $02000000 GPIO_OE_SET !
    begin
        $02000000 $D000001C !
        300 ms
    key? until
  ;
```

We define a constant containing our GPIO number :

```
25 constant ONBOARD_LED
```

And a word transforming this number into a binary mask:

```
: PIN_MASK ( n -- mask )
    1 swap lshift
  ;
```

Along the way, we will also define a constant corresponding to the label having the address `$D000001C` :

```
SIO_BASE $01c + constant GPIO_OUT_XOR
```

This register toggles the state of the GPIO pointed to by its binary mask. Here is the **blink** code integrating these new labels:

```
: blink ( -- )
    5 $400140CC !
    ONBOARD_LED PIN_MASK GPIO_OE_SET !
    begin
        ONBOARD_LED PIN_MASK GPIO_OUT_XOR !
        300 ms
    key? until
  ;
```

We will now factorize the line of code which is in red above. Along the way, we will add the other labels associated with **GPIO_OUT**:

```
SIO_BASE $010 + constant GPIO_OUT
SIO_BASE $014 + constant GPIO_OUT_SET
SIO_BASE $018 + constant GPIO_OUT_CLR
: led.toggle ( gpio -- )
    PIN_MASK GPIO_OUT_XOR !
  ;
```

You begin to understand that the goal is to make FORTH code more and more readable. But this code must also be reusable elsewhere. We will treat this line of code with a refactoring:

```
: blink ( -- )
    5 $400140CC !
    ONBOARD_LED PIN_MASK GPIO_OE_SET !
    begin
        ONBOARD_LED led.toggle
        300 ms
    key? until
  ;
```

We create the word **gpio_set_dir**. The name of this word takes up that of an equivalent C language function:

```
1 constant GPIO_OUT
0 constant GPIO_IN

\ set direction for selected gpio
: gpio_set_dir  ( gpio state -- )
    if      PIN_MASK GPIO_OE_SET !
    else    PIN_MASK GPIO_OE_CLR !    then
  ;
```

Our word **gpio_set_dir** acts on two registers.

Operation of registers associated with GPIO_OUT

Let us detail these registers by referring to the technical document:

0x010	GPIO_OUT	GPIO output value
0x014	GPIO_OUT_SET	GPIO output value set
0x018	GPIO_OUT_CLR	GPIO output value clear
0x01c	GPIO_OUT_XOR	GPIO output value XOR

Figure 11:list of actions associated with the GPIO_OUT register

Details of these registers:

- **GPIO_OUT** is accessible for reading and writing. Reading its contents allows you to retrieve the state of the GPIOs.
- **GPIO_OUT_SET** is write-only. The positioning of one or more bits only acts on the GPIOs indicated in the activation mask.
- **GPIO_OUT_CLR** is write-only. The setting of one or more bits only acts on the GPIOs indicated in the deactivation mask.
- **GPIO_OUT_XOR** is write-only. Switches active bits to inactive bits – and vice versa. This toggle only acts on the GPIOs indicated in the inversion mask.

To turn on the LED of the RP pico card attached to the GPIO 25:

```
ONBOARD_LED PIN_MASK GPIO_OUT_SET !
```

To turn off this same LED:

```
ONBOARD_LED PIN_MASK GPIO_OUT_CLR !
```

If we had only had the **GPIO_OUT** register, the manipulation of the bits would be much more complex:

```
GPIO_OUT @
ONBOARD_LED PIN_MASK xor GPIO_OUT !
```

With our three other registers **GPIO_OUT_SET** , **GPIO_OUT_CLR** and **GPIO_OUT_XOR** , it is not necessary to retrieve the state of the other GPIOs before modifying the state of one or more GPIOs.

We can therefore define a new word **gpio_put** :

```
1 constant GPIO_HIGH    \ set GPIO state
0 constant GPIO_LOW     \ set GPIO state

\ set GPIO on/off
: gpio_put ( gpio state -- )
    if      PIN_MASK GPIO_OUT_SET !
    else    PIN_MASK GPIO_OUT_CLR !    then
  ;
```

Managing GPIO usage

Let's go back to the **blink** source code. There is still one register that is not processed:

```
: blink ( -- )
    5 $400140CC !
    ONBOARD_LED GPIO_OUT gpio_set_dir
    begin
        ONBOARD_LED led.toggle
        300 ms
    key? until
  ;
```

Address **$400140CC** corresponds to register **GPIO25_CTRL**.

Extract from the technical manual:

0x0c8	GPIO25_STATUS	GPIO status
0x0cc	GPIO25_CTRL	GPIO control including function select and overrides.

Figure 12: GPIO25 control register

In the left column, an offset **$0CC** . This offset must apply to a base register **IO_BANK0_BASE** which is defined as follows in FORTH:

```
$40014000 constant IO_BANK0_BASE
```

We now define the word GPIO_CTRL which refers to this base address:

```
: GPIO_CTRL ( n -- addr )
    8 * 4 + IO_BANK0_BASE +
  ;
```

If we execute this:

```
ONBOARD_LED GPIO_CTRL
```

We recover the physical address of the **GPIO25_CTRL** register . The low five bits of this address determine the function of a GPIO. List of possible functions:

```
 1 constant GPIO_FUNC_SPI
 2 constant GPIO_FUNC_UART
 3 constant GPIO_FUNC_I2C
 4 constant GPIO_FUNC_PWM
 5 constant GPIO_FUNC_SIO
 6 constant GPIO_FUNC_PIO0
 7 constant GPIO_FUNC_PIO1
 8 constant GPIO_FUNC_GPCK
 9 constant GPIO_FUNC_USB
$f constant GPIO_FUNC_NULL
```

The function to assign to our GPIO25 port is the **GPIO_FUNC_SIO function** . This function indicates that our GPIO25 port will be used simply for input/output.

We create the word `gpio_set_function` responsible for selecting the function of our GPIO port:

```
: gpio_set_function ( gpio function -- )
    swap GPIO_CTRL !
  ;
```

In C language, in the Raspberry pico SDK, we find the same function **gpio_set_function()** . Our FORTH word `gpio_set_function` has taken the parameters in the same order as the equivalent function in C language. This is not obligatory. In FORTH, we do what we want. Here, the interest is to use the methodology applied to the SDK written in C language, because it is a source of information that should not be neglected.

Finally, here is the final code which allows our LED to flash:

```
$D0000000 constant SIO_BASE
SIO_BASE $020 + constant GPIO_OE
SIO_BASE $024 + constant GPIO_OE_SET
SIO_BASE $028 + constant GPIO_OE_CLR

SIO_BASE $010 + constant GPIO_OUT
SIO_BASE $014 + constant GPIO_OUT_SET
SIO_BASE $018 + constant GPIO_OUT_CLR
SIO_BASE $01c + constant GPIO_OUT_XOR

25 constant ONBOARD_LED

\ transform GPIO number in his binary mask
: PIN_MASK ( n -- mask )
    1 swap lshift
  ;

1 constant GPIO_OUT   \ set direction OUTput mode
0 constant GPIO_IN    \ set direction INput mode

\ set direction for selected gpio
: gpio_set_dir  ( gpio direction -- )
    if     PIN_MASK GPIO_OE_SET !
    else   PIN_MASK GPIO_OE_CLR !    then
  ;

1 constant GPIO_HIGH    \ set GPIO state
0 constant GPIO_LOW     \ set GPIO state

\ set GPIO on/off
: gpio_put ( gpio state -- )
    if     PIN_MASK GPIO_OUT_SET !
    else   PIN_MASK GPIO_OUT_CLR !    then
  ;
```

```
\ toggle led
: led.toggle ( gpio -- )
    PIN_MASK GPIO_OUT_XOR !
  ;

$40014000 constant IO_BANK0_BASE

: GPIO_CTRL ( n -- addr )
    8 * 4 + IO_BANK0_BASE +
  ;

 1 constant GPIO_FUNC_SPI
 2 constant GPIO_FUNC_UART
 3 constant GPIO_FUNC_I2C
 4 constant GPIO_FUNC_PWM
 5 constant GPIO_FUNC_SIO
 6 constant GPIO_FUNC_PIO0
 7 constant GPIO_FUNC_PIO1
 8 constant GPIO_FUNC_GPCK
 9 constant GPIO_FUNC_USB
$f constant GPIO_FUNC_NULL

: gpio_set_function ( gpio function -- )
    swap GPIO_CTRL !
  ;

: blink ( -- )
    ONBOARD_LED GPIO_FUNC_SIO gpio_set_function
    ONBOARD_LED GPIO_OUT gpio_set_dir
    begin
        ONBOARD_LED led.toggle
        300 ms
    key? until
  ;
```

Obviously that's a lot of code just to make an LED blink. Conversely, successive modifications have made it possible to add general-use definitions, such as `gpio_set_function` , `gpio_put` or `gpio_set_dir` .

The example of our word `blink` also allowed us to learn how GPIO registers work. We have only skimmed the incredible possibilities of the RP pico card. All this coding, just to make an LED flash, to lay the first stones of a huge building.

Registers

SIO registers

SIO registers start at base address `$d0000000`. Example definition :

```
$ d0000000 constant SIO_BASE
```

To define a register, we then exploit the offset by simply adding it to this base address :

```
SIO_BASE $020 + constant GPIO_OE
SIO_BASE $024 + constant GPIO_OE_SET
SIO_BASE $028 + constant GPIO_OE_CLR
```

Décalage	Nom	Info
$00	CPUID	Processor core identifier
$004	GPIO_IN	Input value for GPIO pins
$008	GPIO_HI_IN	Input value for QSPI pins
$010	GPIO_OUT	GPIO output value
$014	GPIO_OUT_SET	GPIO output value set
$018	GPIO_OUT_CLR	GPIO output value clear
$01C	GPIO_OUT_XOR	GPIO output value XOR
$020	GPIO_OE	GPIO output enable
$024	GPIO_OE_SET	GPIO output enable set
$028	GPIO_OE_CLR	GPIO output enable clear
$02C	GPIO_OE_XOR	GPIO output enable XOR
$030	GPIO_HI_OUT	QSPI output value
$034	GPIO_HI_OUT_SET	QSPI output value set
$038	GPIO_HI_OUT_CLR	QSPI output value clear
$03C	GPIO_HI_OUT_XOR	QSPI output value XOR
$040	GPIO_HI_OE	QSPI output enable
$044	GPIO_HI_OE_SET	QSPI output enable set
$048	GPIO_HI_OE_CLR	QSPI output enable clear
$04C	GPIO_HI_OE_XOR	QSPI output enable XOR
$050	FIFO_ST	Status register for inter-core FIFOs (mailboxes).
$054	FIFO_WR	Write access to this core’s TX FIFO
$058	FIFO_RD	Read access to this core’s RX FIFO
$05C	SPINLOCK_ST	Spinlock state
$060	DIV_UDIVIDEND	Divider unsigned dividend

Décalage	Nom	Info
$064	DIV_UDIVISOR	Divider unsigned divisor
$068	DIV_SDIVIDEND	Divider signed dividend
$06C	DIV_SDIVISOR	Divider signed divisor
$070	DIV_QUOTIENT	Divider result quotient
$074	DIV_REMAINDER	Divider result remainder
$078	DIV_CSR	Control and status register for divider.
$080	INTERP0_ACCUM0	Read/write access to accumulator 0
$084	INTERP0_ACCUM1	Read/write access to accumulator 1
$088	INTERP0_BASE0	Read/write access to BASE0 register.
$08C	INTERP0_BASE1	Read/write access to BASE1 register.
$090	INTERP0_BASE2	Read/write access to BASE2 register.
$094	INTERP0_POP_LANE0	Read LANE0 result, and simultaneously write lane results to both accumulators (POP).
$098	INTERP0_POP_LANE1	Read LANE1 result, and simultaneously write lane results to both accumulators (POP)
$09C	INTERP0_POP_FULL	Read FULL result, and simultaneously write lane results to both accumulators (POP).
$0A0	INTERP0_PEEK_LANE0	Read LANE0 result, without altering any internal state (PEEK).
$0A4	INTERP0_PEEK_LANE1	Read LANE1 result, without altering any internal state (PEEK).
$0A8	INTERP0_PEEK_FULL	Read FULL result, without altering any internal state (PEEK).
$0AC	INTERP0_CTRL_LANE0	Control register for lane 0
$0B0		INTERP0_CTRL_LANE1 Control register for lane 1
$0B4	INTERP0_ACCUM0_ADD	Values written here are atomically added to ACCUM0
$0B8	INTERP0_ACCUM1_ADD	Values written here are atomically added to ACCUM1
$0BC	INTERP0_BASE_1AND0	On write, the lower 16 bits go to BASE0, upper bits to BASE1 simultaneously.
$0C0	INTERP1_ACCUM0	Read/write access to accumulator 0
$0C4	INTERP1_ACCUM1	Read/write access to accumulator 1
$0C8	INTERP1_BASE0	Read/write access to BASE0 register.
$0CC	INTERP1_BASE1	Read/write access to BASE1 register.
$0D0	INTERP1_BASE2	Read/write access to BASE2 register.
$0D4	INTERP1_POP_LANE0	Read LANE0 result, and simultaneously write lane results to both accumulators (POP).
$0D8	INTERP1_POP_LANE1	Read LANE1 result, and simultaneously write lane results to both accumulators (POP).
$0DC	INTERP1_POP_FULL	Read FULL result, and simultaneously write lane results to both accumulators (POP).

Décalage	Nom	Info
$0E0	INTERP1_PEEK_LANE0	Read LANE0 result, without altering any internal state (PEEK).
$0E4	INTERP1_PEEK_LANE1	Read LANE1 result, without altering any internal state (PEEK).
$0E8	INTERP1_PEEK_FULL	Read FULL result, without altering any internal state (PEEK).
$0EC	INTERP1_CTRL_LANE0	Control register for lane 0
$0F0	INTERP1_CTRL_LANE1	Control register for lane 1
$0F4	INTERP1_ACCUM0_ADD	Values written here are atomically added to ACCUM0
$0F8	INTERP1_ACCUM1_ADD	Values written here are atomically added to ACCUM1
$0FC	INTERP1_BASE_1AND0	On write, the lower 16 bits go to BASE0, upper bits to BASE1 simultaneously.
$100	SPINLOCK0	Spinlock register 0
$104	SPINLOCK1	Spinlock register 1
$108	SPINLOCK2	Spinlock register 2
$10C	SPINLOCK3	Spinlock register 3
$110	SPINLOCK4	Spinlock register 4
$114	SPINLOCK5	Spinlock register 5
$118	SPINLOCK6	Spinlock register 6
$11C	SPINLOCK7	Spinlock register 7
$120	SPINLOCK8	Spinlock register 8
$124	SPINLOCK9	Spinlock register 9
$128	SPINLOCK10	Spinlock register 10
$12C	SPINLOCK11	Spinlock register 11
$130	SPINLOCK12	Spinlock register 12
$134	SPINLOCK13	Spinlock register 13
$138	SPINLOCK14	Spinlock register 14
$13C	SPINLOCK15	Spinlock register 15
$140	SPINLOCK16	Spinlock register 16
$144	SPINLOCK17	Spinlock register 17
$148	SPINLOCK18	Spinlock register 18
$14C	SPINLOCK19	Spinlock register 19
$150	SPINLOCK20	Spinlock register 20
$154	SPINLOCK21	Spinlock register 21
$158	SPINLOCK22	Spinlock register 22
$15C	SPINLOCK23	Spinlock register 23
$160	SPINLOCK24	Spinlock register 24
$164	SPINLOCK25	Spinlock register 25

Décalage	Nom	Info
$168	SPINLOCK26	Spinlock register 26
$16C	SPINLOCK27	Spinlock register 27
$170	SPINLOCK28	Spinlock register 28
$174	SPINLOCK29	Spinlock register 29
$178	SPINLOCK30	Spinlock register 30
$17C	SPINLOCK31	Spinlock register 31

PWM registers

The PWM registers start at a base address of `$40050000`. Example of definition :

```
$40050000  constant PWM_BASE
```

Offset	Name	Info
$00	CH0_CSR	Control and status register
$04	CH0_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
$08	CH0_CTR	Direct access to the PWM counter
$0C	CH0_CC	Counter compare values
$10	CH0_TOP	
$14	CH1_CSR	Control and status register
$18	CH1_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
$1C	CH1_CTR	Direct access to the PWM counter
$20	CH1_CC	Counter compare values
$24	CH1_TOP	Counter wrap value
$28	CH2_CSR	Control and status register
$2C	CH2_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
$30	CH2_CTR	Direct access to the PWM counter
$34	CH2_CC	Counter compare values
$38	CH2_TOP	Counter wrap value
$3C	CH3_CSR	Control and status register
$40	CH3_DIV	0x40 INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.

Offset	Name	Info
$44	CH3_CTR	Direct access to the PWM counter
$48	CH3_CC	Counter compare values
$4C	CH3_TOP	Counter wrap value
$50	CH4_CSR	Control and status register
$54	CH4_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
$58	CH4_CTR	Direct access to the PWM counter
$5C	CH4_CC	Counter compare values
$60	CH4_TOP	Counter wrap value
$64	CH5_CSR	Control and status register
$68	CH5_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
$6C	CH5_CTR	Direct access to the PWM counter
$70	CH5_CC	Counter compare values 0x74
$74	CH5_TOP	Counter wrap value
$78	CH6_CSR	Control and status register
$7C	CH6_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
$80	CH6_CTR	Direct access to the PWM counter
$84	CH6_CC	Counter compare values
$88	CH6_TOP	Counter wrap value
$8C	CH7_CSR	Control and status register
$90	CH7_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
$94	CH7_CTR	Direct access to the PWM counter
$98	CH7_CC	Counter compare values
$9C	CH7_TOP	Counter wrap value
$A0	EN	This register aliases the CSR_EN bits for all channels. Writing to this register allows multiple channels to be enabled or disabled simultaneously, so they can run in perfect sync. For each channel, there is only one physical EN register bit, which can be accessed through here or CHx_CSR.
$A4	INTR	Raw Interrupts
$A8	INTE	Interrupt Enable
$AC	INTF	Interrupt Force

Offset	Name	Info
$B0	INTS	Interrupt status after masking & forcing

TIMER registers

The Timer registers start at a base address of `$40054000`. Exemple de définition :

```
$40054000 constant TIMER_BASE
```

Offset	Name	Info
$00	TIMEHW	Write to bits 63:32 of time always write timelw before timehw
$04	TIMELW	Write to bits 31:0 of time writes do not get copied to time until timehw is written
$08	TIMEHR	Read from bits 63:32 of time always read timelr before timehr
$0C	TIMELR	Read from bits 31:0 of time
$10	ALARM0	Arm alarm 0, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM0 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
$14	ALARM1	Arm alarm 1, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM1 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
$18	ALARM2	Arm alarm 2, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM2 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
$1C	ALARM3	Arm alarm 3, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM3 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
$20	ARMED	Indicates the armed/disarmed status of each alarm. A write to the corresponding ALARMx register arms the alarm. Alarms automatically disarm upon firing, but writing ones here will disarm immediately without waiting to fire
$24	TIMERAWH	Raw read from bits 63:32 of time (no side effects)
$28	TIMERAWL	Raw read from bits 31:0 of time (no side effects)
$2C	DBGPAUSE	Set bits high to enable pause when the corresponding debug ports are active
$30	PAUSE	Set high to pause the timer
$34	INTR	Raw Interrupts
$38	INTE	Interrupt Enable
$3C	INTF	Interrupt Force

Offset	Name	Info
$40	INTS	Interrupt status after masking & forcing

Lexical index