The great book for eForth RP PICO

version 1.0 - 23 décembre 2023

Author

- Marc PETREMANN

Content

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 eForth:

```
: 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.

eForth 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`

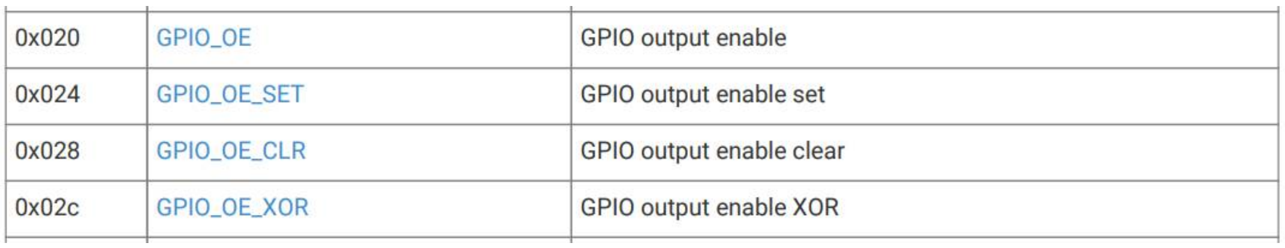
If the address placed on the data stack by `myScore` is addr (for example `$20013D48`), we can replace `myScore @` by `addr @`.

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 1: 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:

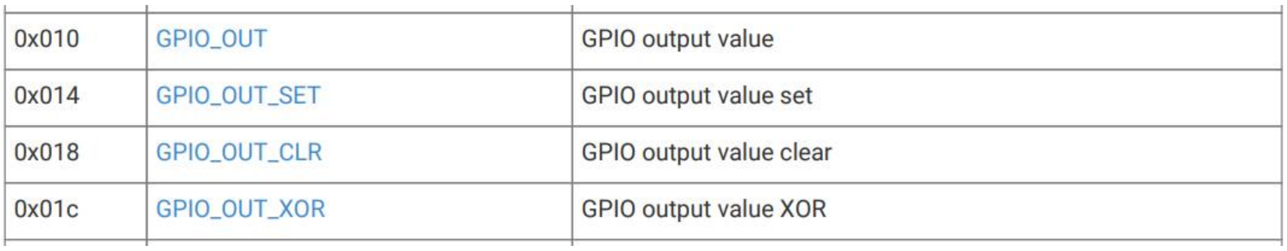
```
1 constant GPIO_OUT
0 constant GPIO_IN
```

```
\ 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 2: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
  ;
```

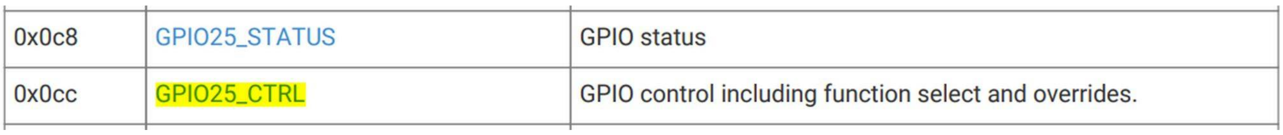
Managing GPIO usage

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 3: 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.

Lexical index