Le grand livre
de eForth RP pico

version 1.0 - 23 décembre 2023

Auteur

- Marc PETREMANN

Table des matières

Utiliser les nombres avec eForth

Nous avons démarré eForth sans souci. Nous allons maintenant approfondir quelques manipulations sur les nombres pour comprendre comment maîtriser la carte RP pico en langage FORTH.

Comme beaucoup d'ouvrages, nous pourrions commencer par un exemple de programme trivial, clignotement de LED par exemple. Dans ce genre par exemple :

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

Vous pouvez tester ce code. Il fonctionnera en faisant clignoter la LED implantée sur la carte RP pico rattachée à GPIO25.

Mais ce code, simple en apparence, nécessite déjà une base de connaissances, comme la notion d'adresse mémoire, registre, masques binaires, nombres hexadécimaux.

Nous allons donc commencer par aborder ces notions élémentaires en vous invitant à effectuer des manipulations simples.

Les nombres avec l'interpréteur FORTH

Au démarrage de eForth, la fenêtre du terminal TERA TERM (ou tout autre programme de terminal de votre choix) doit indiquer la disponibilité de eForth. Appuyez une ou deux fois sur la touche *ENTER* du clavier. eForth répond avec la confirmation de bonne exécution `ok.`.

On va tester l'entrée de deux nombres, ici `25` et `33`. Tapez ces nombres, puis *ENTER* au clavier. eForth répond toujours par `ok.`. Vous venez d'empiler deux nombres sur la pile du langage FORTH. Entrez maintenant `+ .` puis sur la touche *ENTER*. eForth affiche le résultat :

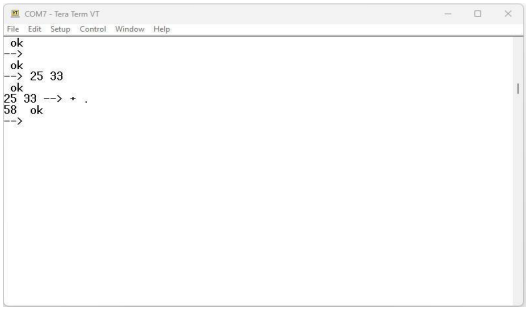

Figure 1: première opération avec eForth

Cette opération a été traitée par l'interpréteur FORTH.

eForth, comme toutes les versions du langage FORTH a deux états :

- **interpréteur** : l'état que vous venez de tester en effectuant une simple somme de deux nombres ;
- **compilateur** : un état qui permet de définir de nouveaux mots. Cet aspect sera approfondi ultérieurement.

Saisie des nombres avec différentes base numérique

Afin de bien assimiler les explications, vous êtes invité à tester tous les exemples via la fenêtre du terminal TERA TERM.

Les nombres peuvent être saisis de manière naturelle. En décimal, ce sera TOUJOURS une séquence de chiffres, exemple :

```
-1234 5678 + .
```

Le résultat de cet exemple affichera `4444`. Les nombres et mots FORTH doivent être séparés par au moins un caractère *espace*. L'exemple fonctionne parfaitement si on tape un nombre ou mot par ligne :

```
-1234
5678
+
.
```

Les nombres peuvent être préfixés si on souhaite saisir des valeurs autrement que sous leur forme décimale :

- le signe `$` pour indiquer que le nombre est une valeur hexadécimale ;

Exemple :

```
decimal
255 .        \ display 255
$ff .        \ display 255
```

L'intéret de ce préfixe est d'éviter toute erreur d'interprétation en cas de valeurs similaires :

```
decimal
$0305
0305
```

ne sont **pas** des nombres **égaux** !

Changement de base numérique

eForth dispose de mots permettant de changer de base numérique :

- `hex` pour sélectionner la base numérique hexadécimale ;
- `binary` pour sélectionner la base numérique binaire ;
- `decimal` pour sélectionner la base numérique décimale.

Tout nombre saisi dans une base numérique doit respecter la syntaxe des nombres dans cette base :

```
3E7F
```

provoquera une erreur si vous êtes en base décimale.

```
hex 3e7f
```

fonctionnera parfaitement en base hexadécimale. La nouvelle base numérique reste valable tant qu'on ne sélectionne pas une autre base numérique :

```
hex
$0305
0305
```

sont des nombres **égaux**!

Une fois un nombre déposé sur la pile de données dans une base numérique, sa valeur ne change plus. Par exemple, si vous déposez la valeur `$ff` sur la pile de données, cette valeur qui est `255` en décimal, ou `11111111` en binaire, ne changera pas si on revient en décimal :

```
hex ff decimal .    \ display : 255
```

Au risque d'insister, `255` en décimal est **la même valeur** que `$ff` en hexadécimal !

On peut définir une constante en hexadécimal :

```
$d0000000 constant SIO_BASE
```

Si on tape :

```
decimal SIO_BASE .
```

Ceci affichera le contenu de cette constante sous sa forme décimale. Le changement de base n'a **aucune conséquence** sur le fonctionnement final du programme FORTH.

Binaire et hexadécimal

Le système de numération binaire moderne, base du code binaire, a été inventé par Gottfried Leibniz en 1689 et apparaît dans son article Explication de l'Arithmétique Binaire en 1703.

Dans son article, LEIBNITZ se sert des seuls caractères `0` et `1` pour décrire tous les nombres :

```
: bin0to15 ( -- )
    binary
    $10 0 do
        cr i .
    loop
    cr  decimal  ;
bin0to15   \ display :
```

```
0
1
10
11
100
101
110
111
1000
1001
1010
1011
1100
1101
1110
1111
```

Est-ce nécessaire de comprendre le codage binaire ? Je dirai oui et non. **Non** pour les usages de la vie courante. **Oui** pour comprendre la programmation des micro-contrôleurs et la maîtrise des opérateurs logiques.

C'est Georges Boole qui a décrit de manière formelle la logique. Ses travaux ont été oubliés jusqu'à l'apparition des premiers ordinateurs. C'est Claude Shannon qui se rend compte qu'on peut appliquer cet algèbre dans la conception et l'analyse de circuits électriques.

L'algèbre de Boole manipule exclusivement des `0` et des `1`.

Les composants fondamentaux de tous nos ordinateurs et mémoires numériques utilisent le codage binaire et l'algèbre de Boole.

La plus petite unité de stockage est l'octet. C'est un espace constitué de 8 bits. Un bit ne peut avoir que deux états : `0` ou `1`. La valeur la plus petite pouvant être stockée dans un octet est `%00000000`, la plus grande étant `%11111111`. Si on coupe en deux un octet, on aura :

- quatre bits de poids faible, pouvant prendre les valeurs `%0000` à `%1111` ;
- quatre bits de poids fort pouvant prendre une de ces mêmes valeurs.

Si on numérote toutes les combinaisons entre %0000 et %1111, en partant de 0, on arrive à 15 :

```
: bin0to15 ( -- )
    binary
    $10 0 do
        cr i .
        i hex . binary
    loop
```

```
    cr  decimal  ;
bin0to15   \ display :
0 0
1 1
10 2
11 3
100 4
101 5
110 6
111 7
1000 8
1001 9
1010 A
1011 B
1100 C
1101 D
1110 E
1111 F
```

Dans la partie droite de chaque ligne, on affiche la même valeur que dans la partie gauche, mais en hexadécimal : `1101` et `D` sont les mêmes valeurs !

La représentation hexadécimale a été choisie pour représenter des nombres en informatique pour des raisons pratiques. Pour la partie de poids fort ou faible d'un octet, sur 4 bits, les seuls combinaisons de représentation hexadécimale seront comprises ente `0` et `F`. Ici, les lettres A à F **sont des chiffres** hexadécimaux !

```
$3E    \ is more readable as%00111110
```

La représentation hexadécimale offre donc l'avantage de représenter le contenu d'un octet dans un format fixe, de `$00` à `$FF`. En décimal, il aurait fallu utiliser 0 à 255.

Taille des nombres sur la pile de données FORTH

eForth utilise une pile de données de 32 bits de taille mémoire, soit 4 octets (8 bits x 4 = 32 bits). La plus petite valeur pouvant être empilée sur la pile FORTH sera `$00000000`, la plus grande sera `$FFFFFFFF`. Toute tentative d'empiler une valeur de taille supérieure se solde par un écrétage de cette valeur :

```
hex
abcdefabcdefabcdef .    \ display :    -10543211
```

Empilons la plus grande valeur possible au format hexadécimal sur 32 bits (4 octets) :

```
$ffffffff .     \ display :    -1
```

Je vous voit surpris, mais ce résultat est **normal** ! Le mot `.` Affiche la valeur qui est au sommet de la pile de données sous sa forme signée. Pour afficher la même valeur non signée, il faut utiliser le mot `u.` :

```
$ffffffff u.    \ display :    FFFFFFFF
```

C'est parce que sur les 32 bits utilisés par eForth pour représenter un nombre entier, le bit de poids fort est utilisé comme signe :

- si le bit de poids fort est à `0`, le nombre est positif ;
- si le bit de poids fort est à `1`, le nombre est négatif.

Donc, si vous avez bien suivi, nos valeurs décimales 1 et -1 sont représentées sur la pile, au format binaire sous cette forme :

```
%00000000000000000000000000000001  \ push 1 on stack
%11111111111111111111111111111111  \ push -1 on stack
```

Et c'est là qu'on va faire appel à notre mathématicien, Mr LEIBNITZ, pour additionner en binaire ces deux nombres. Si on fait comme à l'école, en commençant par la droite, il faudra simplement respecter cette règle : 1 + 1 = 10 en binaire. Il faut donc mettre une troisième ligne pour y reporter le résultat :

```
00000000000000000000000000000001
11111111111111111111111111111111
                              10
```

Étape suivante :

```
00000000000000000000000000000001
11111111111111111111111111111111
                              10
                             100
```

Arrivé à la fin, on aura comme résultat :

```
 00000000000000000000000000000001
 11111111111111111111111111111111
100000000000000000000000000000000
```

Mais comme ce résultat a un 33ème bit de poids fort à 1, sachant que le format des entiers est strictement limité à 32 bits, le résultat final est `0`. C'est surprenant ? C'est pourtant ce que fait toute horloge digitale. Masquez les heures. Arrivé à 59, rajoutez 1, l'horloge affichera 0.

Les règles de l'arithmétique décimale, à savoir `-1 + 1 = 0` ont été parfaitement respectées en logique binaire !

Accès mémoire et opérations logiques

La pile de données n'est en aucun cas un espace de stockage de données. Sa taille est d'ailleurs très limitée. Et la pile est partagée par beaucoup de mots. L'ordre des paramètres est fondamental. Une erreur peut générer des dysfonctionnements. Prenons le cas du mot `dump` qui affiche le contenu d'un espace mémoire :

```
hex
variable score
score 10 dump   \ display :
20013D30            00 00 00 00 33 36 39 35 32 30 34 34 D5 7B 0C 3A
```

En gras et en rouge on retrouve les quatre octets réservés au stockage d'une valeur dans notre variable score. Stockons une valeur quelconque dans **score** :

```
decimal
1900 score !
hex
Score 10 dump   \ partial display :
20013D30            6C 07 00 00 33 36 39 35 32 30 34 34 D5 7B 0C 3A
```

On retrouve les quatre octets contenant notre valeur décimale **1900**, **0000076C** en hexadécimal. Encore surpris ? Alors c'est l'effet du codage binaire et ses subtilités qui en sont la cause. En mémoire, les octets sont stockés en commençant par ceux de poids faible. A la récupération, le mécanisme de transformation est transparent :

```
decimal
score @ .    \ display 1900
```

Revenons au code qui fait clignoter notre LED sur le GPIO25. Extrait :

```
$02000000 $D0000024 !
```

Ce code active GPIO25 en sortie. On aurait aussi bien pu écrire ceci :

```
1 25 lshift $D0000024 !
```

Ici, le mot **lshift** effectue un décalage logique de 25 bits vers la gauche :

```
\ before lshift : %00000000000000000000000000000001
\  after lshift : %00000010000000000000000000000000
```

Pour rappel, les GPIOs[1] sont numérotés de 0 à 31. Pour activer un autre GPIO, par exemple GPIO17, on aurait exécuté ceci :

```
1 17 lshift GPIO_OE !
```

Supposons que nous souhaitions activer en une seule commande les GPIOs 17 et 25. On exécutera ceci :

```
1 25 lshift
1 17 lshift or  $D0000024 !
```

Qu'est-ce que nous avons fait ? Voici le détail des opérations :

```
\ 1 25 lshift  \ %00000010000000000000000000000000
\ 1 17 lshift  \ %00000010000000010000000000000000
\  or          \ %00000010000000010000000000000000
```

Le mot **or** a réalisé une opération qui combine les deux décalages en un seul masque binaire.

1 General Purpose Input/Output = Entrée-sortie à usage général

Revenons à notre variable **score**. On souhaite isoler l'octet de poids faible. Plusieurs solutions s'offrent à nous. Une solution exploite le masquage binaire avec l'opérateur logique **and** :

```
hex
score @ .           \ display :  0000076C
score @
$000000FF and .     \ display :  0000006C
```

Pour isoler le second octet en partant de la droite :

```
score @
$0000FF00 and .     \ display :  00000700
```

Ici, nous nous sommes amusés avec le contenu d'une variable. Pour maîtriser un micro-contrôleur comme celui monté sur la carte RP pico, les mécanismes ne sont guère différents. Le plus difficile est de trouver les bons registres. Ce sera l'objet d'un autre chapitre.

Pour conclure ce chapitre, il y a encore beaucoup à apprendre sur la logique binaire et les différents codages numériques possibles. Si vous avez testé les quelques exemples donnés ici, vous comprenez certainement que FORTH est un langage intéressant :

- grâce à son interpréteur qui permet d'effectuer de nombreux tests, ce de manière interactive sans nécessiter de recompilation en téléversement de code ;
- un dictionnaire dont la plupart des mots sont accessibles depuis l'interpréteur ;
- un compilateur permettant de rajouter de nouveaux mots *à la volée*, puis les tester immédiatement.

Enfin, ce qui ne gâche rien, le code FORTH, une fois compilé, est certainement aussi performant que son équivalent en langage C.

Initiation aux ports GPIO

Tous les apprentis programmeurs sont pressés de tester leur carte. L'exemple le plus trivial consiste à faire clignoter une LED. Il se trouve qu'il y a une LED déjà montée sur la carte RP pico, LED associée au port PIO 25. Voici le code qui vous permettra de faire clignoter cette LED avec eForth :

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

Vous pouvez copier ce code et le transmettre à la carte RP pico par l'intermédiaire du terminal TeraTem. Ca fonctionnera.

Une fois ceci testé, on n'a rien expliqué. Car si vous souhaitez maîtriser les ports d'entrée/sortie, il faut commencer par comprendre ce que ce code est censé faire.

Les ports GPIO

Le Raspberry Pi Pico possède 30 broches GPIO, dont 26 sont utilisables.

- 2x SPI
- 2 x UART
- 2 x I2C
- 8 x PWM à deux canaux

il y a aussi :

- 4 x sortie d'horloge à usage général
- 4 x entrée ADC
- PIO sur toutes les broches

La carte comporte une LED embarquée, reliée au GPIO 25. Elle permet de vérifier le bon fonctionnement de la carte ou tout autre utilisation à votre convenance.

Pour la suite de ce chapitre, nous n'allons voir que le cas de la LED implantée sur la carte RP pico et reliée au GPIO 25.

GPIO et registres

La carte RP pico dispose d'un micro-contrôleur RP2040 dont les données techniques sont accessibles dans ce document au format pdf :
https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

Le contenu de ce document est très technique et peut rebuter un amateur. On va reprendre notre exemple donné en début de chapitre pour expliquer pas à pas ce qui le fait fonctionner, en associant ces informations au contenu du document « RP2040 datasheet ».

Commençons par expliquer ce qu'est un registre de micro-contrôleur.

Un registre est tout simplement une zone de mémoire accessible par une adresse, mais dont l'usage est réservé au fonctionnement du micro-contrôleur. Si on écrit n'importe quoi dans un registre, au mieux on provoque des dysfonctionnements, au pire, on bloque le micro-contrôleur. Si ça se produit, il faudra mettre la carte RP pico hors tension pour la réinitialiser.

eForth accède aux adresses 32 bits avec les mots `!` Et `@`. Exemple :

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

Ici, on définit une variable `myScore`. Quand on tape le nom de cette variable, on empile en fait l'adresse mémoire pointant sur le contenu de cette variable :

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

Manipulation des données vers cette adresse :

- `3215 myScore !` stocke une valeur à l'adresse mémoire réservée par `myScore`
- `myScore @` récupère le contenu stocké à l'adresse mémoire réservée par `myScore`

Si l'adresse déposée sur la pile de données par `myScore` est addr (par exemple `$20013D48`), on peut remplacer `myScore @` par `addr @`.

Pour accéder aux données d'un registre, ce n'est pas très différent. Prenons le cas de ce registre :

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

Ici, son adresse mémoire est **$D0000024**. La séquence **$02000000 $D0000024 !** Stocke simplement la valeur **$02000000** dans l'adresse mémoire **$D0000024**.

Dans le document document « RP2040 datasheet », cette adresse a comme 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 2: extrait du document technique RP2040

On peut donc rendre notre code FORTH un peu plus lisible en définissant ce label comme constante FORTH :

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

Avec l'habitude, on append à déchiffrer les labels. Ici, **GPIO_OE_SET** signifie quasiment « **GPIO O**utput **E**nable **SET** » (Valide Position Sortie GPIO).

Utiliser les labels de registres

Le label **GPIO_OE_SET** est lui-même un sous-ensemble des registres définis dans le label global **SIO_BASE**.D'ailleurs, dans la documentation technique, dans la première colonne mentionnant le label **GPIO_OE_SET** on trouve le décalage (offset) par rapport à **SIO_BASE**. On va tenir compte de ceci pour réécrire une partie du code FORTH. On en profite pour définir deux autres labels comme constantes :

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

A cette étape, notre code FORTH reste encore dépendant du GPIO25 au travers de ce masque :

```
: blink ( -- )
    5 $400140CC !
    $02000000 GPIO_OE_SET !
    begin
```

```
        $02000000 $D000001C !
        300 ms
    key? until
  ;
```

On définir une constante reprenant notre numéro de GPIO :

```
25 constant ONBOARD_LED
```

Et un mot effectuant la transformation de ce numéro en un masque binaire :

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

Au passage, on va aussi définir une constante correspondant au label ayant l'adresse **$D000001C** :

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

Ce registre bascule l'état du GPIO pointé par son masque binaire. Voici le code de **blink** intégrant ces nouveaux labels :

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

On va maintenant factoriser la ligne de code qui est en rouge ci-dessus. Au passage, on va rajouter les autres labels associés à **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 !
  ;
```

Vous commencez à comprendre que le but est de rendre le code FORTH de plus en plus lisible. Mais il faut aussi que ce code puisse être réutilisable ailleurs. On va traiter cette ligne de code par une refactorisation :

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

```
  ;
```

On crée le mot **gpio_set_dir**. Le nom de ce mot reprend celui d'une fonction en langage C équivalente :

```
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
  ;
```

Notre mot **gpio_set_dir** agit sur deux registres.

Fonctionnement des registres associés à GPIO_OUT

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 3: liste des actions associées au registre GPIO_OUT

Détaillons ces registres en se référant au document technique :

Détail de ces registres :

- **GPIO_OUT** est accessible en lecture et écriture. La lecture de son contenu permet de récupérer l'état des GPIOs.
- **GPIO_OUT_SET** est accessible en écriture seulement. Le positionnement de un ou plusieurs bits n'agit que sur les GPIOs indiqués dans le masque d'activation.
- **GPIO_OUT_CLR** est accessible en écriture seulement. Le positionnement de un ou plusieurs bits n'agit que sur les GPIOs indiqués dans le masque de désactivation.
- **GPIO_OUT_XOR** est accessible seulement en écriture. Bascule les bits à l'état actif vers les bits à l'état inactif – et inversement. Cette bascule n'agit que sur les GPIOs indiqués dans le masque d'inversion.

Pour allumer la LED de la carte RP pico attachée au GPIO 25 :

```
ONBOARD_LED PIN_MASK GPIO_OUT_SET !
```

Pour éteindre cette même LED :

```
ONBOARD_LED PIN_MASK GPIO_OUT_CLR !
```

Si nous n'avions disposé que du seul registre **GPIO_OUT**, la manipulation des bits serait nettement plus complexe :

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

Avec nos trois autres registres **GPIO_OUT_SET**, **GPIO_OUT_CLR** et **GPIO_OUT_XOR**, il n'est pas nécessaire de récupérer l'état des autres GPIOs avant de modifier l'état de un ou plusieurs GPIOs.

On peut donc définir un nouveau mot **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
  ;
```

Gestion de l'utilisation des GPIOs

Revenons au code source de **blink**. Il y a encore un registre qui n'est pas traité :

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

L'adresse **$400140CC** correspond au registre **GPIO25_CTRL**.

Extrait du manuel technique :

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

Figure 4: Registre de contrôle de GPIO25

Dans la colonne de gauche, un décalage **$0CC**. Ce décalage doit s'appliquer ua registre de base **IO_BANK0_BASE** que l'on définit ainsi en FORTH :

```
$40014000 constant IO_BANK0_BASE
```

On définit maintenant le mot GPIO_CTRL qui se férère à cette adresse de base :

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

```
  ;
```

Si on exécute ceci :

```
ONBOARD_LED  GPIO_CTRL
```

On récupère bien l'adresse physique du registre **GPIO25_CTRL**. Les cinq bits de poids faible de cette adresse déterminent la fonction d'un GPIO. Liste des fonctions possibles :

```
 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
```

La fonction à affecter à notre port GPIO25 est la fonction **GPIO_FUNC_SIO**. Cette fonction indique que notre port GPIO25 sera utilisé simplement en entrée/sortie.

On crée le mot **gpio_set_function** chargé de sélectionner la fonction de notre port GPIO :

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

En langage C, dans le SDK de Raspberry pico, on retrouve la même fonction **gpio_set_function()**. Notre mot FORTH **gpio_set_function** a repris les paramètres dans le même ordre que la fonction équivalente en langage C. Ce n'est pas obligatoire. En FORTH, on fait ce qu'on veut. Ici, l'intéret est de reprendre la méthodologie appliquée au SDK écrit en langage C, car c'est une source d'information qui n'est pas à négliger.

Pour finir, voici le code final qui permet de faire clignoter notre LED :

```
$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
  ;
```

Il est évident que ça fait beaucoup de code juste pour faire clignoter une LED. A contrario, les modifications successives ont permis de rajouter des définitions d'usage général, comme `gpio_set_function`, `gpio_put` ou `gpio_set_dir`.

L'exemple de notre mot `blink` a également permis d'apprendre comment fonctionnent les registres GPIOs. Nous n'avons fait que survoler les incroyables possibilités de la carte RP pico. Tout ce codage, juste pour faire clignoter une LED, permetr de poser les premières pierres d'un immense édifice.

Index lexical