

# Le grand livre de MECRISP Forth

version 1.2 - 31 décembre 2023



Auteur

- Marc PETREMANN

## Table des matières

<b>Introduction.....</b>	<b>4</b>
Aide à la traduction.....	4
<b>Installer MECRISP sur la carte RP pico.....</b>	<b>5</b>
Préparation de la carte RP PICO.....	5
Téléchargement et installation de MECRISP sur RP PICO.....	5
Brancher et alimenter la carte RP Pico.....	6
Alimenter la carte RP pico.....	6
Brancher le port série UART0.....	8
Communiquer avec la carte RP pico.....	9
<b>Installer et utiliser le terminal Tera Term sous Windows.....</b>	<b>11</b>
Installer Tera Term.....	11
Paramétrage de Tera Term.....	11
Utilisation de Tera Term.....	13
Transmettre du code source FORTH vers MECRISP Forth.....	14
<b>Utiliser les nombres avec MECRISP Forth.....</b>	<b>16</b>
Les nombres avec l'interpréteur FORTH.....	16
Saisie des nombres avec différentes base numérique.....	17
Changement de base numérique.....	18
Binaire et hexadécimal.....	18
Taille des nombres sur la pile de données FORTH.....	20
Accès mémoire et opérations logiques.....	22
<b>Un vrai FORTH 32 bits avec MECRISP.....</b>	<b>24</b>
Les valeurs sur la pile de données.....	24
Les valeurs en mémoire.....	24
Traitement par mots selon taille ou type des données.....	25
Conclusion.....	26
<b>Rendre le code FORTH accessible de manière permanente.....</b>	<b>28</b>
Fixer le code FORTH en mémoire FLASH.....	28
Cas pratique, extension du dictionnaire.....	29
<b>Commentaires et mise au point.....</b>	<b>31</b>
Ecrire un code FORTH lisible.....	31
Indentation du code source.....	32
Les commentaires.....	33
Les commentaires de pile.....	33
Signification des paramètres de pile en commentaires.....	34
Commentaires des mots de définition de mots.....	35
Les commentaires textuels.....	35
Commentaire en début de code source.....	36
Outils de diagnostic et mise au point.....	36
Le décompilateur.....	36
Dump mémoire.....	37
Moniteur de pile.....	37

<b>Dictionnaire / Pile / Variables / Constantes.....</b>	<b>39</b>
Étendre le dictionnaire.....	39
Piles et notation polonaise inversée.....	39
Manipulation de la pile de paramètres.....	41
La pile de retour et ses utilisations.....	41
Utilisation de la mémoire.....	42
Variables.....	42
Constantes.....	42
Outils de base pour l'allocation de mémoire.....	43
<b>Les nombres réels avec MECRISP Forth.....</b>	<b>44</b>
Nombres réels sur 32 bits.....	44
<b>Affichage des nombres et chaînes de caractères.....</b>	<b>46</b>
Changement de base numérique.....	46
Définition de nouveaux formats d'affichage.....	47
Affichage des caractères et chaînes de caractères.....	49
<b>Initiation aux ports GPIO.....</b>	<b>52</b>
Les ports GPIO.....	52
GPIO et registres.....	53
Utiliser les labels de registres.....	54
Fonctionnement des registres associés à GPIO_OUT.....	56
Gestion de l'utilisation des GPIOs.....	57
<b>Commande de relais Reed par GPIO.....</b>	<b>61</b>
Piloter un relais Reed.....	61
Le relais REED.....	61
Pilotage du relais REED par la carte RP pico.....	62
Gestion d'un relais industriel bistable.....	63
<b>Les ports GPIO en entrée.....</b>	<b>65</b>
L'interrupteur.....	65
Gestion par GPIO.....	66
Exercice pratique avec notre contacteur.....	66
<b>Le multitâche avec MECRISP Forth.....</b>	<b>68</b>
Définition du multi-tâche.....	68
Activer et désactiver des tâches.....	69
<b>Contenu détaillé des mots MECRISP Forth.....</b>	<b>71</b>
<b>Les registres du processeur RP2040.....</b>	<b>73</b>
Les registres SIO.....	73
Les registres PWM.....	76
Les registres timer.....	78
<b>Ressources.....</b>	<b>80</b>

# Introduction

Je gère depuis 2019 plusieurs sites web consacrés aux développements en langage FORTH pour les cartes ARDUINO et ESP32, ainsi que la version eForth web :

- ARDUINO : <https://arduino-forth.com/>
- ESP32 : <https://esp32.arduino-forth.com/>
- eForth web : <https://eforth.arduino-forth.com/>
- MECRISP Forth : <https://mecrisp.arduino-forth.com/>

Ces sites sont disponibles en deux langues, français et anglais. Chaque année je paie l'hébergement du site principal **arduino-forth.com**.

Il arrivera tôt ou tard – et le plus tard possible – que je ne sois plus en mesure d'assurer la pérennité de ces sites. La conséquence sera que les informations diffusées par ces sites disparaissent.

Ce livre est la compilation du contenu de mes sites web. Il est diffusé librement depuis un dépôt Github. Cette méthode de diffusion permettra une plus grande pérennité que des sites web.

Accessoirement, si certains lecteurs de ces pages souhaitent apporter leur contribution, ils sont bienvenus :

- pour proposer des chapitres ;
- pour signaler des erreurs ou suggérer des modifications ;
- pour aider à la traduction...

## Aide à la traduction

Google Translate permet de traduire des textes facilement, mais avec des erreurs. Je demande donc de l'aide pour corriger les traductions.

En pratique, je fournis, les chapitres déjà traduits, dans le format LibreOffice. Si vous voulez apporter votre aide à ces traductions, votre rôle consistera simplement à corriger et renvoyer ces traductions.

La correction d'un chapitre demande peu de temps, de une à quelques heures.

**Pour me contacter :**     [petremann@arduino-forth.com](mailto:petremann@arduino-forth.com)

# Installer MECRISP sur la carte RP pico

## Préparation de la carte RP PICO

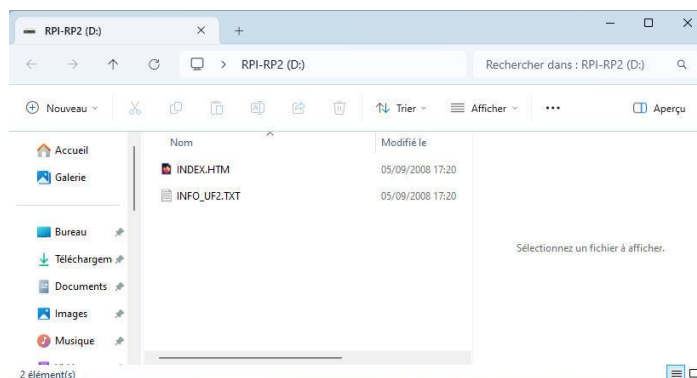
Déballiez et installez la carte RP PICO sur une plaque d'essai.



*Figure 1: branchement au PC via un câble USB*

Branchez ensuite un cordon USB coté carte RP PICO:

Sur la carte RP PICO, il y a un bouton poussoir. Maintenez ce bouton enfoncé et branchez simultanément le cordon au PC. Sous WINDOWS 11, cette action ouvre immédiatement



*Figure 2: espace de stockage  
dans la carte RP pico*

une fenêtre dans l'explorateur de fichiers:

## Téléchargement et installation de MECRISP sur RP PICO

Vous pouvez télécharger MECRISP pour RP PICO depuis ce lien:

<https://mecrisp.arduino-forth.com/public/fichiers/mecrisp-stellaris-pico-with-tools.uf2>

Une fois le fichier téléchargé, ouvrez le dossier de téléchargement.

Sélectionnez le fichier **mecrisp-stellaris-pico-with-tools.uf2**.

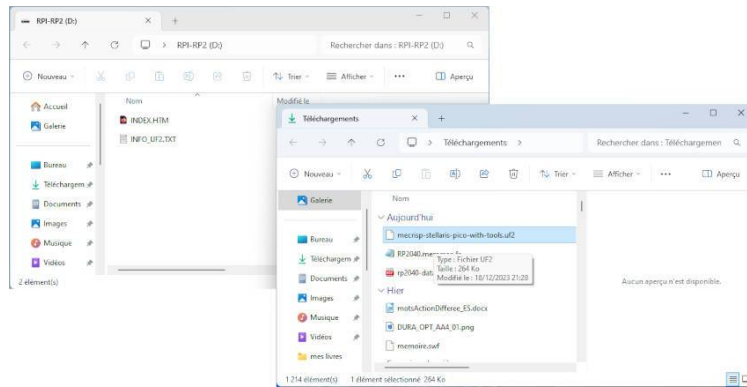


Figure 3: copie du fichier d'extension UF2 vers l'espace de la carte RP pico

Glissez et déposez ce fichier **mecrisp-stellaris-pico-with-tools.uf2** vers le dossier de la carte RP PICO.

Une fois le fichier copié, la fenêtre de fichiers RP PICO se ferme. MECRISP pour RP PICO est installé.

## Brancher et alimenter la carte RP Pico

MECRISP Forth n'est pas accessible par le mini connecteur usb de la carte RP pico qui a servi à l'installation de MECRISP Forth.

Pour pouvoir communiquer avec MECRISP Forth, il est nécessaire de procéder à quelques installations.

## Alimenter la carte RP pico



Figure 4: porte pile sur AMAZON

Le choix s'est porté sur un interface d'alimentation de ce type:

Ici, sur AMAZON, les 5 unités coûtent moins de 30€. Il est indiqué porte piles, mais fonctionne parfaitement avec des batteries lithium au format 18650. Ces batteries lithium offrent une très grande autonomie.

Le porte pile de ce type dispose de:

- une sortie USB standard

- une entrée mini USB pour permettre de recharger la batterie depuis un chargeur USB classique
- un mini interrupteur coupant l'alimentation vers la sortie USB standard
- trois sorties 5V
- trois sorties 3V3

Les sorties 3V3 et 5V ne sont pas interrompues par le mini interrupteur. Il n'y a **pas de protection** par fusible. Je conseille donc d'être prudent en soudant des fils de raccord aux sorties 3V3 ou 5V. Effectuez ces opérations sans batterie ni connexion à la carte électronique.

Ici, j'ai soudé deux fils *dupont* sur une sortie 3V3:

- fil rouge sur la sortie 3V

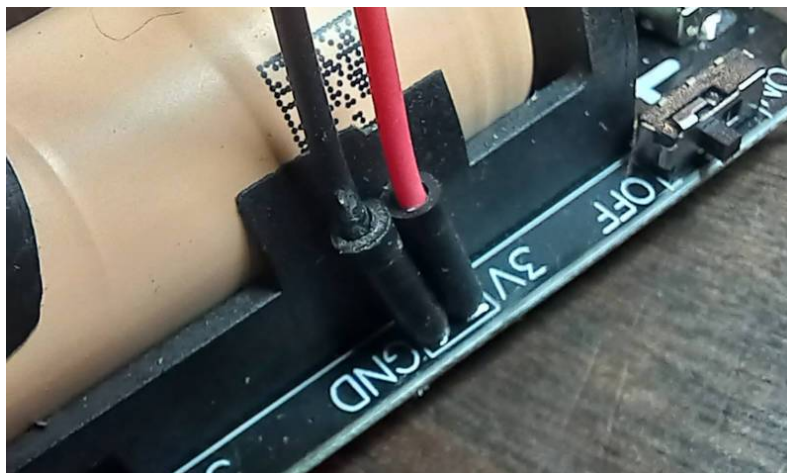


Figure 5: soudage de fils à une sortie 3V3

- fil noir sur la sorte GND

Sur la photo, on distingue parfaitement le mini interrupteur. Au risque d'insister, cet interrupteur **ne coupe pas les sorties 3VE et 5V**.

Le branchement des fils noir et rouge venant de l'alimentation doivent être connectés à la carte RP pico aux pins 38 et 39:



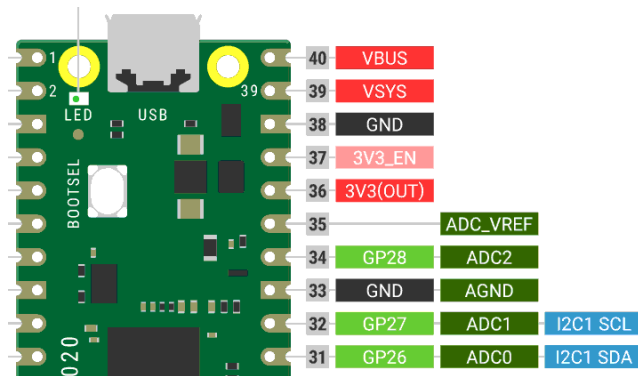


Figure 6: pins 38 et 39  
vers alimentation 3V3

L'alimentation doit être raccordée à la carte RP pico par l'intermédiaire d'une plaque d'essai:

- **fil rouge** en premier à connecter au pin 39. Cette précaution est essentielle, car si la fiche touche accidentellement un autre pin, ce sera sans conséquence pour la carte RP pico;
- **fil noir** ensuite à connecter au pin 38.

Pour interrompre l'alimentation de la carte RP pico, déconnectez le fil noir du pin 38.

## Brancher le port série UART0

Le raccordement entre la carte RP pico et le PC s'effectue via cet interface qui est un adaptateur TTL <--> USB.



Figure 7: adaptateur TTL <--> USB

Sur cet adaptateur, il y a 5 sorties:

- 2 sorties: 3V3 et 5V que nous n'exploiterons pas. Le risque d'endommager le PC accidentellement sera ainsi réduit au maximum;
- 2 pin d'entrée/sortie TXD et RXD. Ce sont ces sorties que nous exploiterons;



- une sortie GND qui doit également être exploitée.

Voici les raccordements à effectuer entre la carte RP pico et cet interface:

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

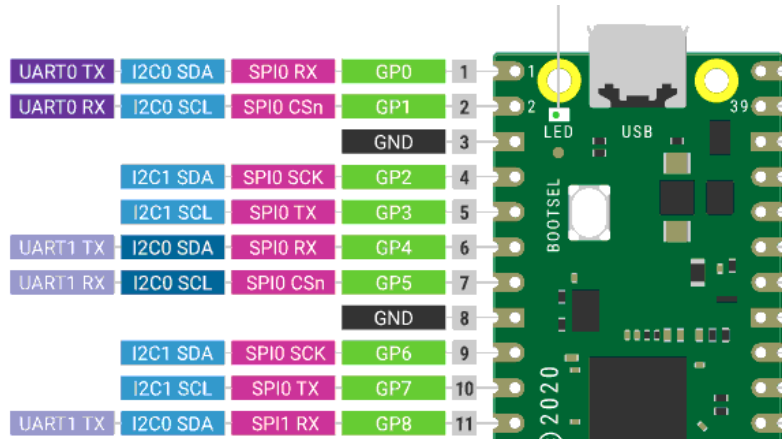


Figure 8: pins 1 à 3 vers adaptateur USB / TTL

- RXD --> pin 1 (UART0 TX)

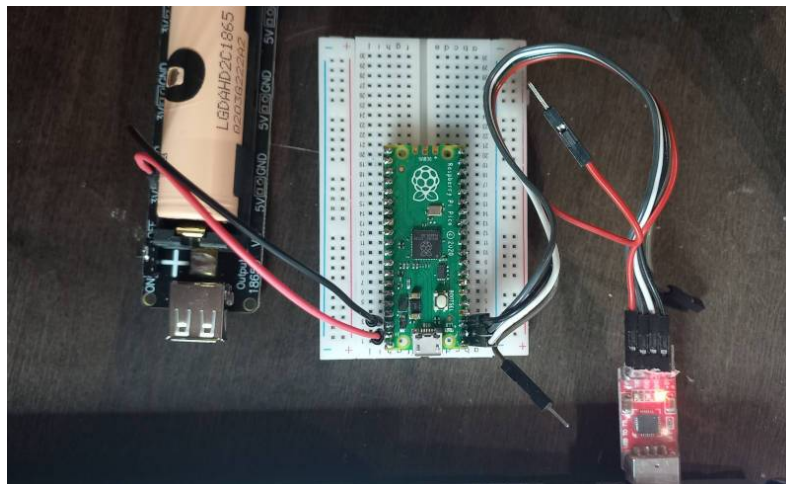


Figure 9: branchement carte RP pico au PC

Vous pouvez maintenant connecter l'adaptateur au PC et mettre la carte RP pico sous tension via la batterie:

## Communiquer avec la carte RP pico

Pour communiquer la carte RP pico, il faut utiliser un programme TERMINAL. Je vous conseille TERATERM si vous êtes sous Windows.

Lancez TERATERM. Les paramètres pour communiquer avec la carte RP PICO sous eForth PICO ICE sont les suivants:

- PORT: le port com USB disponible
- Speed: 115200
- Data: 8 bit
- Parity: none
- Stop bits: 1 bit
- Flow control: none

Si la liaison série est bien établie, la carte RP pico doit répondre à l'appui sur la touche **ENTER** sur le clavier du PC.

Testez la bonne réactivité de MECRISP en tapant la commande **list**. Cette commande affichera le contenu du dictionnaire FORTH.

# Installer et utiliser le terminal Tera Term sous Windows

## Installer Tera Term

La page en anglais pour Tera Term, c'est ici:

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

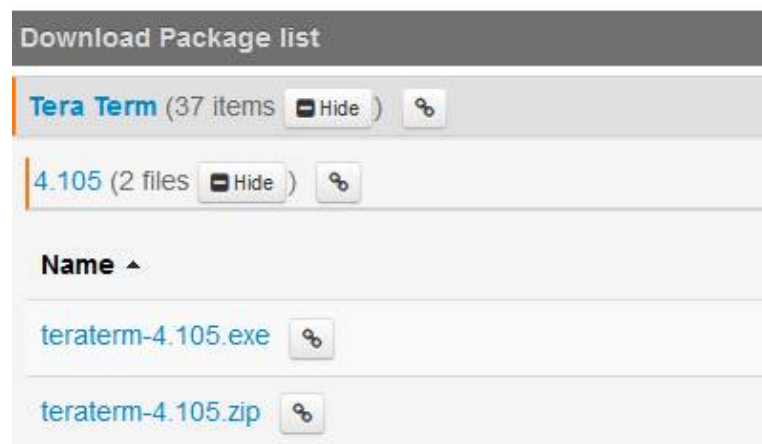


Figure 10: invite de téléchargement de TERA TERM

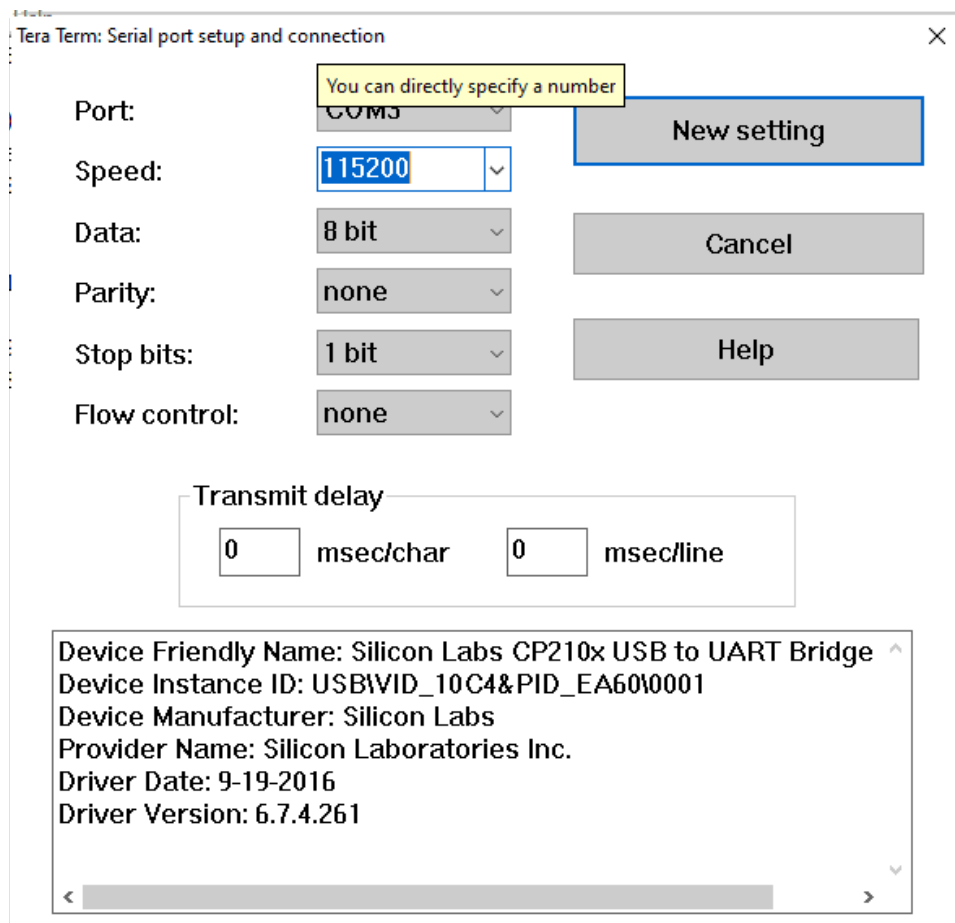
Allez sur la page téléchargement, récupérez le fichier exe ou zip:

Installez Tera Term. L'installation est simple et rapide.

## Paramétrage de Tera Term

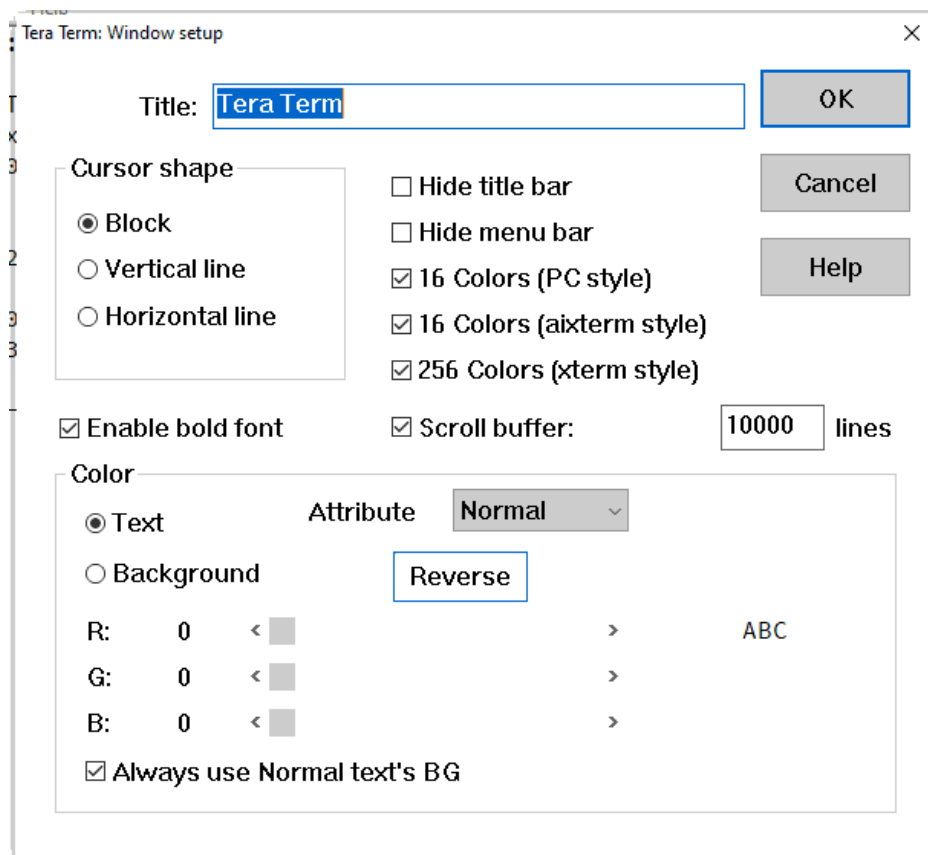
Pour communiquer avec MECRISP Forth, il faut régler certains paramètres:

- cliquez sur Configuration -> port série

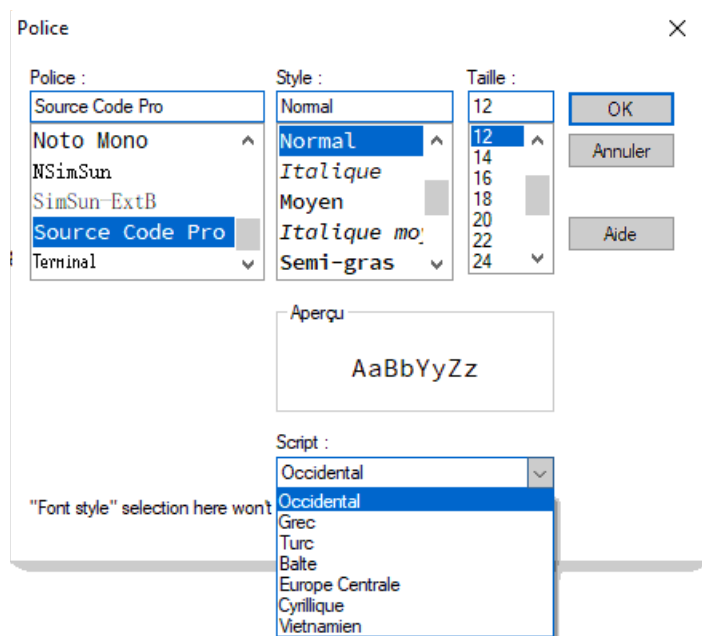


Pour un affichage confortable:

- cliquez sur Configuration -> fenêtre



Pour des caractères lisibles:



- cliquez sur Configuration -> police

Pour retrouver tous ces réglages au prochain lancement du terminal Tera Term, sauvegardez la configuration:

- cliquez sur *Setup* -> *Save setup*

- acceptez le nom **TERATERM.INI**.

## Utilisation de Tera Term

Une fois paramétré, fermez Tera Term.

Connectez votre carte RP pico à un port USB disponible de votre PC.

Relancez Tera Term, puis cliquez sur *fichier -> nouvelle connexion*

Sélectionnez le port série:

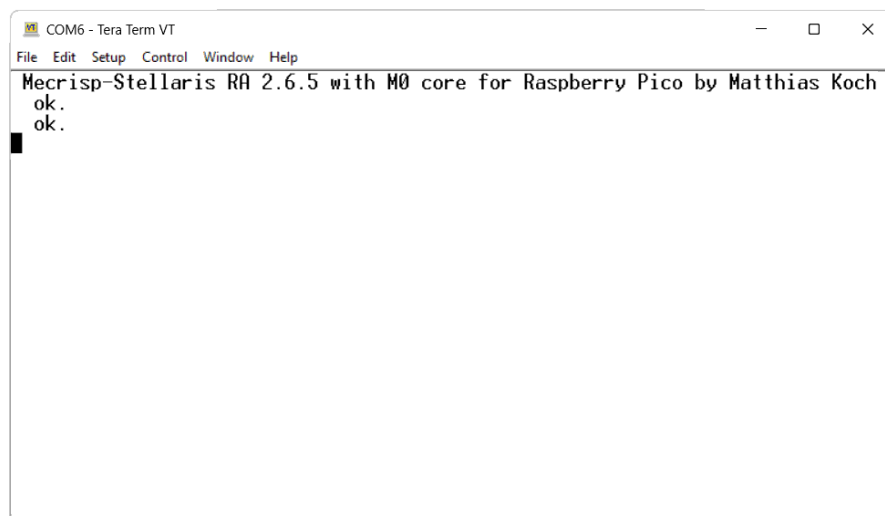
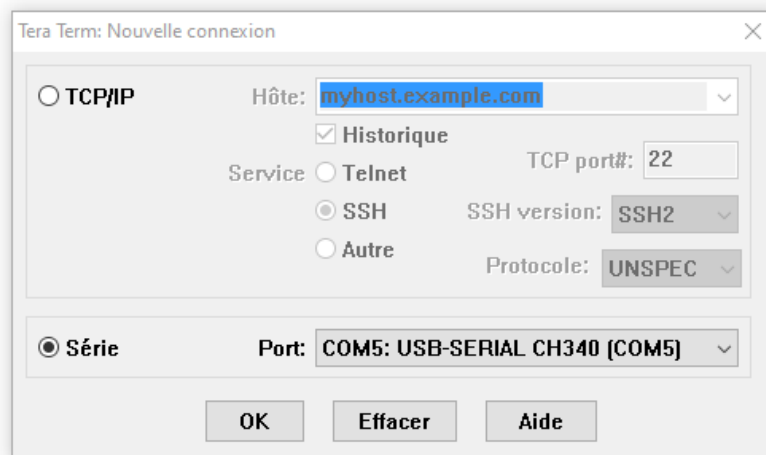


Figure 11: fenêtre du terminal Tera Term communiquant avec MECRISP Forth

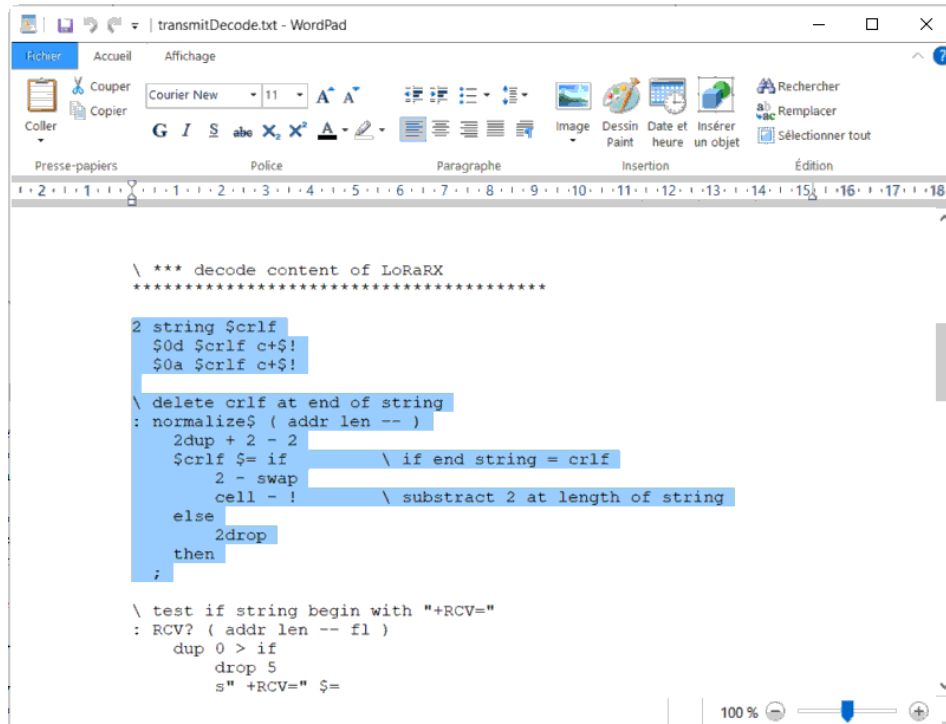
Si tout s'est bien passé, vous devez voir ceci:

## Transmettre du code source FORTH vers MECRISP Forth

Tout d'abord, rappelons que le langage FORTH est sur la carte RP pico! FORTH n'est pas sur votre PC. Donc, on ne peut pas compiler le code source d'un programme en langage FORTH sur le PC.

Pour compiler un programme en langage FORTH, il faut au préalable ouvrir un fichier source sur le PC avec l'éditeur de votre choix.

Ensuite, on copie le code source à compiler. Ici, un code source ouvert avec Wordpad:



Le code source en langage FORTH peut être composé et édité avec n'importe quel éditeur de texte: bloc notes, PSpad, Wordpad..

Personnellement j'utilise l'IDE Netbeans. Cet IDE permet d'éditer et gérer des codes sources dans de nombreux langages de programmation..

Sélectionnez le code source ou la portion de code qui vous intéresse. Puis cliquez sur copier. Le code sélectionné est dans le tampon d'édition du PC.

Cliquez sur la fenêtre du terminal Tera Term. Faites Coller:

Il suffit de valider en cliquant sur OK et le code sera interprété et/ou compilé.

Pour exécuter un code compilé, il suffit de taper le mot FORTH à lancer, ce depuis le terminal Tera Term.



## Utiliser les nombres avec MECRISP Forth

Nous avons démarré MECRISP Forth sans souci. Nous allons maintenant approfondir quelques manipulations sur les nombres pour comprendre comment maîtriser le micro-contrôleur 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 :

```
$d0000000 constant SIO_BASE

SIO_BASE $01c + constant GPIO_OUT_XOR \ GPIO output value XOR
SIO_BASE $020 + constant GPIO_OE      \ GPIO output enable

25 constant ONBOARD_LED

: blink ( -- )
  1 ONBOARD_LED lshift GPIO_OE !
  begin
    1 ONBOARD_LED lshift GPIO_OUT_XOR !
    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.

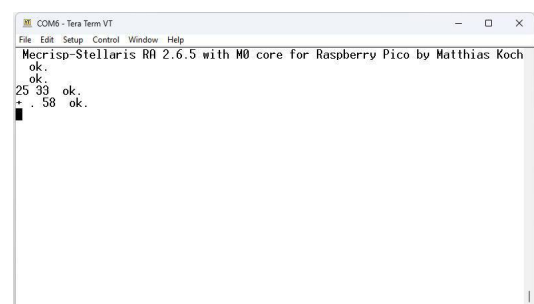
Mais ce code, simple en apparence, nécessite déjà une base de connaissances, comme la notion d'adresse mémoire, registre, masques binaires, nombres hexadécimaux.

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 MECRISP Forth, la fenêtre du terminal TERA TERM (ou tout autre programme de terminal de votre choix) doit indiquer la disponibilité de MECRISP Forth. Appuyez une ou deux fois sur la touche *ENTER* du clavier. MECRISP 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. MECRISP répond toujours par **ok..** Vous venez d'empiler deux



nombres sur la pile du langage FORTH. Entrez maintenant **+** . puis sur la touche *ENTER*.  
MECRISP affiche le résultat :

Cette opération a été traitée par l'interpréteur FORTH.

MECRISP Forth, 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 un nombre décimal ;
- le signe **\$** pour indiquer que le nombre est une valeur hexadécimale ;
- le signe **%** pour indiquer que le nombre est une valeur binaire.

Exemple :

```
#255 .      \ display 255
$ff .       \ display 255
%11111111 . \ display 255
```

L'intérêt de ces préfixes est d'éviter toute erreur d'interprétation en cas de valeurs similaires :

```
$0305
#0305
```

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

## Changement de base numérique

MECRISP Forth 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 !

Dans l'exemple donné en début de chapitre, on définit 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 droite, 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 seules combinaisons de représentation hexadécimale seront comprises entre **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

MECRISP 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 **\$FFFFFFF**. Toute tentative d'empiler une valeur de taille supérieure se solde par un écrêtage 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 FORTH 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 :

[illegible]

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 :

[illegible]

Etape suivante :

[illegible]

Arrivé à la fin, on aura comme résultat :

```
%0000000000000000000000000000000000000000000000000000001
%1111111111111111111111111111111111111111111111111111111111
%1000000000000000000000000000000000000000000000000000000000
```

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 !





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

```
\ 1 25 lshift \ %000000100000000000000000000000000000000
\ 1 17 lshift \ %0000001000000001000000000000000000000000
\ or          \ %0000001000000001000000000000000000000000
```

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

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** :

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

Pour isoler le second octet en partant de la droite :

```
score @
$0000FF00 and hex. \ 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.

# Un vrai FORTH 32 bits avec MECRISP

MECRISP est un vrai FORTH 32 bits. Qu'est-ce que ça signifie ?

Le langage FORTH privilégie la manipulation de valeurs entières. Ces valeurs peuvent être des valeurs littérales, des adresses mémoires, des contenus de registres...

## Les valeurs sur la pile de données

Au démarrage de MECRISP, l'interpréteur FORTH est disponible. Si vous entrez n'importe quel nombre, il sera déposé sur la pile sous sa forme d'entier 32 bits :

```
35
```

Si on empile une autre valeur, elle sera également empilée. La valeur précédente sera repoussée vers le bas d'une position :

```
45
```

Pour faire la somme de ces deux valeurs, on utilise un mot, ici **+** :

```
+
```

Nos deux valeurs entières 32 bits sont additionnées et le résultat est déposé sur la pile. Pour afficher ce résultat, on utilisera le mot **.** :

```
. \ affiche 80
```

En langage FORTH, on peut concentrer toutes ces opérations en une seule ligne:

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

Contrairement au langage C, on ne définit pas de type **int8** ou **int16** ou **int32**.

Avec MECRISP, un caractère ASCII sera désigné par un entier 32 bits, mais dont la valeur sera bornée [32..256[. Exemple :

```
decimal  
67 emit \ display C
```

Avec MECRISP Forth, un entier 32 bits signé sera défini dans l'intervalle -2147483648 à 2147483647.

Parfois, on parle de demi-mot. Un demi-mot numérique concerne la partie 16 bits de poids fort ou poids faible d'un entier 32 bits. Un demi-mot sera défini dans l'intervalle 0 à 65,535

## Les valeurs en mémoire

MECRISP permet de définir des constantes, des variables. Leur contenu sera toujours au format 32 bits. Mais il est des situations où ça ne nous arrange pas forcément. Prenons un exemple simple, définir un alphabet morse. Nous n'avons besoin que de quelques octets :

- un pour définir le nombre de signes du code morse
- un ou plusieurs octets pour chaque lettre du code morse

```
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,
```

Ici, nous définissons seulement 3 mots, **mA**, **mB** et **mC**. Dans chaque mot, on stocke plusieurs octets. La question est: comment va-t-on récupérer les informations dans ces mots?

L'exécution d'un de ces mots dépose une valeur 32 bits, valeur qui correspond à l'adresse mémoire où on a stocké nos informations morse. C'est le mot **c@** qui va nous servir à extraire le code morse de chaque lettre :

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

Le premier octet extrait ainsi va nous servir à gérer une boucle pour afficher le code morse d'une lettre :

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

Il existe plein d'exemples certainement plus élégants. Ici, c'est pour montrer une manière de manipuler des valeurs 8 bits, nos octets, alors qu'on exploite ces octets sur une pile 32 bits.

## Traitement par mots selon taille ou type des données

Dans tous les autres langages, on a un mot générique, genre **echo** (en PHP) qui affiche n'importe quel type de donnée. Que ce soit entier, réel, chaîne de caractères, on utilise toujours le même mot. Exemple en langage PHP :

```
$bread = "Pain cuit";
```

```
$price = 2.30;
echo $bread . " : " . $price;
// affiche    Pain cuit: 2.30
```

Pour tous les programmeurs, cette manière de faire est LA NORME! Alors comment ferait FORTH pour cet exemple en PHP?

```
: pain s" Pain cuit" ;
: prix s" 2.30" ;
pain type    s" : " type    prix type
\ affiche    Pain cuit: 2.30
```

Ici, le mot **type** nous indique qu'on vient de traiter une chaîne de caractères.

Là où PHP (ou n'importe quel autre langage) a une fonction générique et un analyseur syntaxique, FORTH compense avec un type de donnée unique, mais des méthodes de traitement adaptées qui nous informent sur la nature des données traitées.

Voici un cas absolument trivial pour FORTH, afficher un nombre de secondes au format HH:MM:SS:

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

J'adore cet exemple, car, à ce jour, **AUCUN AUTRE LANGAGE DE PROGRAMMATION** n'est capable de réaliser cette conversion HH:MM:SS de manière aussi élégante et concise.

Vous l'avez compris, le secret de FORTH est dans son vocabulaire.

## Conclusion

FORTH n'a pas de typage de données. Toutes les données transitent par une pile de données. Chaque position dans la pile est TOUJOURS un entier 32 bits !

### C'est tout ce qu'il y a à savoir.

Les puristes de langages hyper structurés et verbeux, tels C ou Java, crieront certainement à l'hérésie. Et là, je me permettrai de leur répondre : pourquoi avez-vous besoin de typer vos données ?

Car, c'est dans cette simplicité que réside la puissance de FORTH: une seule pile de données avec un format non typé et des opérations très simples.

Et je vais vous montrer ce que bien d'autres langages de programmation ne savent pas faire, définir de nouveaux mots de définition :

```
: morse: ( comp: c -- | exec -- )
  <builds
    ,
  does>
    dup 1 cells + swap @ 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   .- -... -.-.
```

Ici, le mot **morse:** est devenu un mot de définition, au même titre que **constant** ou **variable**...

Car FORTH est plus qu'un langage de programmation. C'est un méta-langage, c'est à dire un langage pour construire **votre propre langage** de programmation....

# Rendre le code FORTH accessible de manière permanente

MECRISP Forth écrit le code FORTH compilé en deux endroits bien distincts:

- en mémoire RAM par défaut ou si on sélectionne **compiletoram**
- en mémoire FLASH si on sélectionne **compiletoflash**

Si vous compilez du code FORTH, ce code disparaît de la carte RP pico (ou toute autre carte) dès sa mise hors tension on un **reset**.

Un exemple simple en compilant juste cette définition:

```
: myTest  ." This is my test" ;
```

Si on compile cette définition, puis on exécute **list**, on retrouve notre mot:

```
list
myTest --- Mecrisp-Stellaris RA 2.6.5 --- 2dup 2drop 2swap 2nip 2over 2tuck 2rot...
```

Ce code est compilé dans la mémoire RAM. Il ne peut donc pas être conservé après une mise hors tension de la carte RP pico ou l'exécution de **reset**.

## Fixer le code FORTH en mémoire FLASH

L'idée est de rendre toute ou une partie du code FORTH accessible en permanence comme le sont déjà tous les mots du dictionnaire FORTH.

Pour réaliser cela, on doit d'abord donner comme directive à MECRISP que le code qui suit doit être enregistré en mémoire FLASH:

```
compiletoflash
/ ...here my new words
```

Une fois notre code compilé en mémoire FLASH, on exécute le mot **save**:

```
compiletoflash
/ ...here my new words
save
```

Ceci fait, on peut reprogrammer la compilation des définitions vers la mémoire RAM avec **compiletoram**:

```
compiletoflash
/ ...here my new words
save
compiletoram
```

## Cas pratique, extension du dictionnaire

Voici un lien vers des définitions FORTH permettant une gestion simplifiée des ports GPIO:

<https://github.com/MPETREMAN11/MECRISP-Stellaris/blob/main/tools/gpio.fs>

Les directives **compiletoflash**, **save** et **compiletoram** sont déjà intégrées au code source. Il suffit d'exécuter quelques copié/collé entre le code source et le terminal connecté à la carte RP pico pour intégrer ce code à MECRISP Forth.

Une fois ce code chargé, coupez l'alimentation de la carte RP pico, ou exécutez **reset**.

Après redémarrage de MECRISP Forth, vous devez retrouver ces mots dans le dictionnaire FORTH:

```
list \ display:
.....singletask task-in-list? previous insert remove task: preparetask activate
background tasks catch throw TIMEHW TIMEHW TIMEHR TIMEHR TIMERAWH TIMERAWL cycles
delay-cycles us ms u.4 u.2 dump16 dump test SIO_BASE GPIO_OE GPIO_OE_SET GPIO_OE_CLR
GPIO_IN GPIO_OUT GPIO_OUT_SET GPIO_OUT_CLR GPIO_OUT_XOR PIN_MASK GPIO_OUT GPIO_IN
gpio_set_dir GPIO_HIGH GPIO_LOW gpio_put gpio_get IO_BANK0_BASE GPIO_CTRL
GPIO_FUNC_SPI GPIO_FUNC_UART GPIO_FUNC_I2C GPIO_FUNC_PWM GPIO_FUNC_SIO GPIO_FUNC_PIO0
GPIO_FUNC_PIO1 GPIO_FUNC_GPCK GPIO_FUNC_USB GPIO_FUNC_NULL gpio_set_function
IO_BANK0_GPIO0_CTRL_FUNCSEL_BITS gpio_get_function
```

Ces définitions sont devenues permanentes. Elles font autant partie de FORTH que tous les autres mots déjà définis. Pour rappel, le principe de base de la compilation FORTH consiste à **étendre le dictionnaire** FORTH.

Contrairement aux autres langages de programmation, avec FORTH il n'y a pas de distinction entre le compilateur, l'interpréteur, le code compilé. Exécutez juste ces quelques lignes:

```
25 constant BOARD_LED
BOARD_LED GPIO_FUNC_SIO gpio_set_function
BOARD_LED GPIO_OUT gpio_set_dir
BOARD_LED GPIO_HIGH gpio_put
500 ms
BOARD_LED GPIO_LOW gpio_put
```

La LED intégrée à la carte RP pico doit s'allumer pendant une demi seconde.

Tous ces tests sont exécutables via l'interpréteur. On peut également compiler en mémoire RAM ces même tests:

```
25 constant BOARD_LED
: LED.init ( -- )
    BOARD_LED GPIO_FUNC_SIO gpio_set_function
    BOARD_LED GPIO_OUT gpio_set_dir
;
: blink ( -- )
    begin
        BOARD_LED GPIO_HIGH gpio_put
```



```
500 ms  
BOARD_LED GPIO_LOW gpio_put  
500 ms  
key? until  
;
```

Pour conclure, ne fixer en mémoire FLASH que du code FORTH testé et fiable. Nous verrons dans un autre chapitre comment rendre un mot exécutable au démarrage de MECRISP Forth.

## Commentaires et mise au point

Il n'existe pas d'IDE<sup>2</sup> pour gérer et présenter le code écrit en langage FORTH de manière structurée. Au pire, vous utilisez un éditeur de texte ASCII, au mieux un vrai IDE et des fichiers texte :

- **edit** ou **wordpad** sous Windows
- **edit** sous Linux
- **PsPad** sous windows
- **Netbeans** sous Windows ou Linux...

Voici un extrait de code qui pourrait être écrit par un débutant :

```
: cycle.stop -1 +to MAX_LIGHT_TIME MAX_LIGHT_TIME 0 = if  
LOW myLIGHTS pin else 0 rerun then ;
```

Ce code sera parfaitement compilé par MECRISP Forth. Mais restera-t-il compréhensible dans le futur s'il faut le modifier ou le réutiliser dans une autre application ?

## Ecrire un code FORTH lisible

Commençons par le nomage du mot à définir, ici **cycle.stop**. MECRISP Forth permet d'écrire des noms de mots très longs. La taille des mots définis n'a aucune influence sur les performances de l'application finale. On dispose donc d'une certaine liberté pour écrire ces mots :

- à la manière de la programmation objet en JavaScript: **cycle.stop**
- à la manière CamelCoding **cycleStop**
- pour programmeur voulant un code très compréhensible **cycle-stop-lights**
- programmeur qui aime le code concis **cs1**

Il n'y a pas de règle. L'essentiel est que vous puissiez facilement relire votre code FORTH. Cependant, les programmeurs informatique en langage FORTH ont certaines habitudes :

- constantes en caractères majuscules **MAX\_LIGHT\_TIME\_NORMAL\_CYCLE**
- mot de définition d'autres mots **defPin:**, c'est à dire mot suivi de deux points ;
- mot de transformation d'adresse **>date**, ici le paramètre d'adresse est incrémenté d'une certaine valeur pour pointer sur la donnée adéquate ;
- mot de stockage mémoire **date@** ou **date!**

---

2 Integrated Development Environment = Environnement de Développement Intégré

- Mot d’affichage de donnée **.date**

Et qu’en est-il du nommage des mots FORTH dans une langue autre qu’en anglais ? Là encore, une seule règle : **liberté totale** ! Attention cependant, MECRISP Forth n’accepte pas les noms écrits dans des alphabets différents de l’alphabet latin. Vous pouvez cependant utiliser ces alphabets pour les commentaires :

```
: .date      \ Плакат сегодняшней даты
...code...  ;
```

OU

```
: .date      \ 海報今天的日期
...code...  ;
```

## Indentation du code source

Que le code soit sur deux lignes, dix lignes ou plus, ça n’a aucun effet sur les performances du code une fois compilé. Donc, autant indenter son code de manière structurée :

- une ligne par mot de structure de contrôle **if else then, begin while repeat...** Pour le mot **if**, on peut de faire précéder du test logique qu’il traitera ;
- une ligne par exécution d’un mot prédéfini, précédé le cas échéant des paramètres de ce mot.

Exemple :

```
60 constant MAX_LIGHT_TIME_NORMAL_CYCLE
: cycle.stop
  -1 +to MAX_LIGHT_TIME
  MAX_LIGHT_TIME 0 =
  if
    LOW myLIGHTS pin
  else
    0 rerun
  then
;

```

Si le code traité dans une structure de contrôle est peu fourni, le code FORTH peut être compacté :

```
: cycle.stop
  -1 +to MAX_LIGHT_TIME
  MAX_LIGHT_TIME 0 =
  if      LOW myLIGHTS pin
  else    0 rerun          then
;

```

C’est d’ailleurs souvent le cas avec des structures **case of endof endcase** ;

```

: socketError ( -- )
  errno dup
  case
    2 of      ." No such file "      endof
    5 of      ." I/O error "        endof
    9 of      ." Bad file number "   endof
    22 of     ." Invalid argument "  endof
  endcase
  . quit
;

```

## Les commentaires

Comme tout langage de programmation, le langage FORTH permet le rajout de commentaires dans le code source. Le rajout de commentaires n'a aucune conséquence sur les performances de l'application après compilation du code source.

En langage FORTH, nous disposons de deux mots pour délimiter des commentaires :

- le mot **(** suivi impérativement d'au moins un caractère espace. Ce commentaire est achevé par le caractère **)** ;
- le mot **\** suivi impérativement d'au moins un caractère espace. Ce mot est suivi d'un commentaire de taille quelconque entre ce mot et la fin de la ligne.

Le mot **(** est largement utilisé pour les commentaires de pile. Exemples :

```

dup   ( n - n n )
swap  ( n1 n2 - n2 n1 )
drop  ( n -- )
emit  ( c -- )

```

## Les commentaires de pile

Comme nous venons de le voir, ils sont marqués par **(** et **)**. Leur contenu n'a aucune action sur le code FORTH en compilation ou en exécution. On peut donc mettre n'importe quoi entre **(** et **)**. Pour ce qui concerne les commentaires de pile, on restera très concis. Le signe **--** symbolise l'action d'un mot FORTH. Les indications figurant avant **--** correspondent aux données déposées sur la pile de données avant l'exécution du mot. Les indications figurant après **--** correspondent aux données laissées sur la pile de données après exécution du mot. Exemples :

- **words ( -- )** signifie que ce mot ne traite aucune donnée sur la pile de données ;
- **emit ( c -- )** signifie que ce mot traite une donnée en entrée et ne laisse rien sur la pile de données ;
- **bl ( -- 32 )** signifie que ce mot ne traite pas de donnée en entrée et laisse la valeur décimale 32 sur la pile de données ;

Il n'y a aucune limitation sur le nombre de données traitées avant ou après exécution du mot. Pour rappel, les indications entre ( et ) sont seulement là pour information.

## Signification des paramètres de pile en commentaires

Pour commencer, une petite mise au point très importante s'impose. Il s'agit de la taille des données en pile. Avec MECRISP Forth, les données de pile occupent 4 octets. Ce sont donc des entiers au format 32 bits. Cependant, certains mots traitent des données au format 8 bits. Alors on met quoi sur la pile de données ? Avec MECRISP Forth, ce seront **TOUJOURS DES DONNEES 32 BITS** ! Un exemple avec le mot **c!** :

```
create myDelemiter
  0 c,
64 myDelimiter c!    ( c addr -- )
```

Ici, le paramètre **c** indique qu'on empile une valeur entière au format 32 bits, mais dont la valeur sera toujours comprise dans l'intervalle [0..255].

Le paramètre standard est toujours **n**. S'il y a plusieurs entiers, on les numérote : **n1 n2 n3**, etc.

On aurait donc pu écrire l'exemple précédent comme ceci :

```
create myDelemiter
  0 c,
64 myDelimiter c!    ( n1 n2 -- )
```

Mais c'est nettement moins explicite que la version précédente. Voici quelques symboles que vous serez amené à voir au fil des codes sources :

- **addr** indique une adresse mémoire littérale ou délivrée par une variable ;
- **c** indique une valeur 8 bits dans l'intervalle [0..255]
- **d** indique une valeur double précision.  
Non utilisé avec MECRISP Forth qui est déjà au format 32 bits ;
- **fl** indique une valeur booléenne, 0 ou non zéro ;
- **n** indique un entier. Entier signé 32 bits pour MECRISP Forth ;
- **str** indique une chaîne de caractère. Équivaut à **addr len --**
- **u** indique un entier non signé

Rien n'interdit d'être un peu plus explicite :

```
: SQUARE ( n -- n-exp2 )
  dup *
;
```

## Commentaires des mots de définition de mots

Les mots de définition utilisent **create** et **does>**. Pour ces mots, il est conseillé d'écrire les commentaires de pile de cette manière :

```
\ define a command or data stream for SSD1306
: streamCreate: ( comp: <name> | exec: -- addr len )
  create
    here      \ leave current dictionary pointer on stack
    0 c,      \ initial lenght data is 0
  does>
    dup 1+ swap c@
    \ send a data array to SSD1306 connected via I2C bus
    sendDataToSSD1306
;
```

Ici, le commentaire est partagé en deux parties par le caractère **|** :

- à gauche, la partie action quand le mot de définition est exécuté, préfixé par **comp:**
- à droite la partie action du mot qui sera défini, préfixé par **exec:**

Au risque d'insister, ceci n'est pas un standard. Ce sont seulement des recommandations.

## Les commentaires textuels

Ils sont inqués par le mot **\** suivi obligatoirement par au moins un caractère espace et du texte explicatif :

```
\ store at <WORD> addr length of datas compiled beetween
\ <WORD> and here
: ;endStream ( addr-var len ---)
  dup 1+ here
  swap -      \ calculate cdata length
  \ store c in first byte of word defined by streamCreate:
  swap c!
;
```

Ces commentaires peuvent être écrits dans n'importe quel alphabet supporté par votre éditeur de code source :

```
\ 儲存在 <WORD> addr 之間編譯的資料長度
\ <WORD> 和這裡
: ;endStream ( addr-var len ---)
  dup 1+ here
  swap -      \ 計算 cdata 長度
  \ 將 c 儲存在由 StreamCreate 定義的字的第一個位元組中:
  swap c!
;
```

## Commentaire en début de code source

Avec une pratique de programmation intensive, on se retrouve rapidement avec des centaines, voire des milliers de fichiers source. Pour éviter des erreurs de choix de fichiers, il est fortement conseillé de marquer le début de chaque fichier source avec un commentaire :

```
\ *****
\ Manage commands for OLED SSD1306 128x32 display
\   Filename:      SSD10306commands.fs
\   Date:          21 may 2023
\   Updated:       21 may 2023
\   File Version:  1.0
\   MCU:           Raspberry pico
\   Forth:         MECRISP Forth
\   Copyright:     Marc PETREMANN
\   Author:        Marc PETREMANN
\   GNU General Public License
\ *****
```

Toutes ces informations sont à votre libre choix. Elles peuvent devenir très utiles quand on revient des mois ou des années plus tard sur le contenu d'un fichier.

Pour conclure, n'hésitez pas à commenter et indenter vos fichiers sources en langage FORTH.

## Outils de diagnostic et mise au point

Le premier outil concerne l'alerte de compilation ou d'interprétation :

```
3 5 25 --> : TEST ( ---)
ok
3 5 25 --> [ HEX ] ASCII A DDUP \ DDUP don't exist
```

Ici, le mot **DDUP** n'existe pas. Toute compilation après cette erreur sera vouée à l'échec.

## Le décompilateur

Dans un compilateur conventionnel, le code source est transformé en code exécutable contenant les adresses de référence à une bibliothèque équipant le compilateur. Pour disposer d'un code exécutable, il faut linker le code objet. A aucun moment le programmeur ne peut avoir accès au code exécutable contenu dans ses bibliothèque avec les seules ressources du compilateur.

Avec MECRISP Forth, le développeur peut décompiler ses définitions. Pour décompiler un mot, il suffit de taper **see** suivi du mot à décompiler :

```
: C>F ( °C --- °F) \ Conversion Celsius in Fahrenheit
  9 5 */ 32 +
;
```



```

see c>f
\ display:
200204F6: 3F04 subs r7 #4
200204F8: 603E str r6 [ r7 #0 ]
200204FA: 2609 movs r6 #9
200204FC: 3F04 subs r7 #4
200204FE: 603E str r6 [ r7 #0 ]
20020500: 2605 movs r6 #5
20020502: B500 push { lr }
20020504: F7E0 bl 200009A6 --> */
20020506: FA4F
20020508: 3620 adds r6 #20
2002050A: BD00 pop { pc }

```

Beaucoup de mots du dictionnaire FORTH de MECRISP Forth peuvent être décompilés. MECRISP Forth transforme le code source FORTH en instruction compilées en assembleur RP2040 pour ce qui concerne la carte RP pico.

## Dump mémoire

Parfois, il est souhaitable de pouvoir voir les valeurs qui sont en mémoire. Le mot **dump** accepte deux paramètres: l'adresse de départ en mémoire et le nombre d'octets à visualiser :

```

create myDATAS 01 c, 02 c, 03 c, 04 c,
hex
myDATAS 4 dump \ displays :
20020520 : 00 30 00 02 2D 30 00 02 4D 30 80 47 01 02 03 04 | .0..-0.. M0.G....
|
20020530 : 93 72 A4 FB 5D 5D 14 A7 E0 4C 1D 6D 9F 69 C2 E9 | .r..]].. .L.m.i..
|

```

## Moniteur de pile

Le contenu de la pile de données peut être affiché à tout moment grâce au mot **.s**. Voici la définition du mot **.DEBUG** qui exploite **.s** :

```

0 variable debugStack

: debugOn ( -- )
  -1 debugStack !
;

: debugOff ( -- )
  0 debugStack !
;

: .debug
  debugStack @
  if

```

```

    cr ." STACK: " .s
    key drop
  then
;

```

Pour exploiter **.DEBUG**, il suffit de l'insérer dans un endroit stratégique du mot à mettre au point :

```

\ example of use:
: myTEST
  128 32 do
    i .DEBUG
    emit
  loop
;

```

Ici, on va afficher le contenu de la pile de données après exécution du mot **i** dans notre boucle **do loop**. On active la mise au point et on exécute **myTEST** :

```

debugOn
myTest
\ displays:
STACK: Stack: [1 ] -74DCA62D  TOS: 32  *>
2
STACK: Stack: [1 ] -74DCA62D  TOS: 33  *>
3
STACK: Stack: [1 ] -74DCA62D  TOS: 34  *>
4
STACK: Stack: [1 ] -74DCA62D  TOS: 35  *>

```

Quand la mise au point est activée par **debugOn**, chaque affichage du contenu de la pile de données met en pause notre boucle **do loop**. Exécuter **debugOff** pour que le mot **myTEST** s'exécute normalement.

# Dictionnaire / Pile / Variables / Constantes

## Étendre le dictionnaire

Forth appartient à la classe des langages d'interprétation tissés. Cela signifie qu'il peut interpréter les commandes tapées sur la console, ainsi que compiler de nouveaux sous-programmes et programmes.

Le compilateur FORTH fait partie du langage et des mots spéciaux sont utilisés pour créer de nouvelles entrées de dictionnaire (c'est-à-dire des mots). Les plus importants sont **:** (commencer une nouvelle définition) et **;** (termine la définition). Essayons ceci en tapant:

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

Ce qui s'est passé? L'action de **:** est de créer une nouvelle entrée de dictionnaire nommée **\*+** et passer du mode interprétation au mode compilation. En mode compilation, l'interpréteur recherche les mots et, plutôt que de les exécuter, installe des pointeurs vers leur code. Si le texte est un nombre, au lieu de le pousser sur la pile, MECRISP construit le nombre dans le dictionnaire l'espace alloué pour le nouveau mot, suivant le code spécial qui met le numéro stocké sur la pile chaque fois que le mot est exécuté. L'action d'exécution de **\*+** est donc d'exécuter séquentiellement les mots définis précédemment **\*** et **+**.

Le mot **;** est spécial. C'est un mot immédiat et il est toujours exécuté, même si le système est en mode compilation. Ce que fait **;** est double. Tout d'abord, il installe le code qui renvoie le contrôle au niveau externe suivant de l'interpréteur et, deuxièmement, il revient du mode compilation au mode interprétation.

Maintenant, essayez votre nouveau mot :

```
decimal 5 6 7 *+ . \ affiche 47
```

Cet exemple illustre deux activités principales de travail dans Forth: ajouter un nouveau mot au dictionnaire, et l'essayer dès qu'il a été défini.

## Piles et notation polonaise inversée

FORTH a une pile explicitement visible qui est utilisée pour passer des nombres entre les mots (commandes). Utiliser FORTH efficacement vous oblige à penser en termes de pile. Cela peut être difficile au début, mais comme pour tout, cela devient beaucoup plus facile avec la pratique.

En FORTH, La pile est analogue à une pile de cartes avec des nombres écrits dessus. Les nombres sont toujours ajoutés au sommet de la pile et retirés du sommet de la pile.

MECRISP intègre deux piles: la pile de paramètres et la pile de retour, chacune composée d'un certain nombre de cellules pouvant contenir des nombres de 16 bits.

La ligne d'entrée FORTH:

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

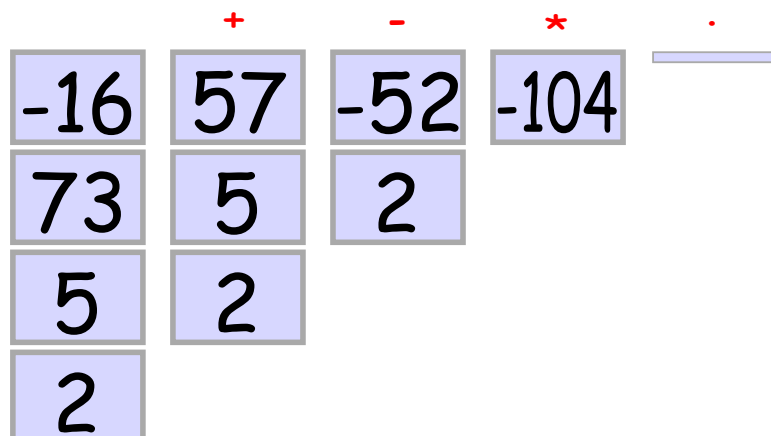
laisse la pile de paramètres dans l'état

Cellule	contenu	commentaire
0	-16	(TOS) Sommet pile
1	73	(NOS) Suivant dans la pile
2	5	
3	2	

Nous utiliserons généralement une numérotation relative à partir de zéro dans les structures de données FORTH telles que piles, tableaux et tables. Notez que, lorsqu'une séquence de nombres est saisie comme celle-ci, le nombre le plus à droite devient *TOS* et le nombre le plus à gauche se trouve au bas de la pile.

Suivons la ligne d'entrée d'origine avec la ligne

```
+ - * .
```



Les opérations produiraient les opérations de pile successives:

Après les deux lignes, la console affiche :

```
decimal 2 5 73 -16 .s \ affiche: Stack: [4 ] 4 2 5 73 TOS: -16
+ - * . \ affiche: -104 ok
```

MECRISP n'affiche pas les éléments de la pile lors de l'interprétation de chaque ligne. Pour voir le contenu de la pile, il faut exécuter **.s**. La valeur de -16 est affichée sous la forme d'entier non signé 32 bits. Le mot **.** consomme la valeur de données -104, laissant la pile vide. Si nous exécutons **.** sur la pile maintenant vide, l'interpréteur externe abandonne avec une erreur de pointeur de pile **Stack underflow**.

La notation de programmation où les opérandes apparaissent en premier, suivis du ou des opérateurs est appelée Notation polonaise inverse (RPN).

## Manipulation de la pile de paramètres

Étant un système basé sur la pile, MECRISP Forth doit fournir des moyens de mettre des nombres sur la pile, pour les supprimer et réorganiser leur ordre. On a déjà vu comment mettre des nombres sur la pile simplement en les tapant. Nous pouvons également intégrer les nombres dans la définition d'un mot FORTH.

Le mot **drop** supprime un numéro du sommet de la pile mettant ainsi le suivant au sommet. Le mot **swap** échange les 2 premiers numéros. **dup** copie le nombre au sommet, poussant tout les autres numéros vers le bas. **rot** fait pivoter les 3 premiers nombres. Ces



actions sont présentées ci-dessous.

## La pile de retour et ses utilisations

Lors de la compilation d'un nouveau mot, MECRISP établit des liens entre le mot appelant et les mots définis précédemment qui doivent être invoqués par l'exécution du nouveau mot. Ce mécanisme de liaison, lors de l'exécution, utilise la pile de retour (rstack).

L'adresse du mot suivant à invoquer est placée sur la pile de retour de sorte que, lorsque l'exécution du mot courant est terminée, le système sait où passer au mot suivant. Comme les mots peuvent être imbriqués, il doit y avoir une pile de ces adresses de retour.

En plus de servir de réservoir d'adresses de retour, l'utilisateur peut également stocker et récupérer à partir de la pile de retour, mais cela doit être fait avec soin car la pile de retour est essentielle à l'exécution du programme. Si vous utilisez la pile de retour pour le stockage temporaire, vous devez la remettre dans son état d'origine, sinon vous ferez probablement planter le système MECRISP. Malgré le danger, il y a des moments où l'utilisation de pile de retour comme stockage temporaire peut rendre votre code moins complexe.

Pour stocker dans la pile, utilisez **>r** pour déplacer le sommet de la pile de paramètres vers le haut de la pile de retour. Pour récupérer une valeur, **r>** déplace la valeur supérieure de la pile de retour vers le sommet de la pile de paramètres. Pour supprimer

simplement une valeur du haut de la pile, il y a le mot **rdrop**. Le mot **r@** copie le haut de la pile de retour dans la pile de paramètres.

## Utilisation de la mémoire

Dans MECRISP, les nombres 32 bits sont extraits de la mémoire vers la pile par le mot **@** (fetch) et stocké du sommet à la mémoire par le mot **!** (store). **@** attend une adresse sur la pile et remplace l'adresse par son contenu. **!** attend un nombre et une adresse pour le stocker. Il place le numéro dans l'emplacement de mémoire référencé par l'adresse, consommant les deux paramètres dans le processus.

Les nombres non signés qui représentent des valeurs de 8 bits (octets) peuvent être placés dans des caractères de la taille d'un caractère. cellules de mémoire en utilisant **c@** et **c!**.

```
create testVar
    1 cells allot
    $f7 testVar c!
testVar c@ . \ affiche 247
```

## Variables

Une variable est un emplacement nommé en mémoire qui peut stocker un nombre, tel que le résultat intermédiaire d'un calcul, hors de la pile. Par exemple:

```
variable x
```

crée un emplacement de stockage nommé, **x**, qui s'exécute en laissant l'adresse de son emplacement de stockage au sommet de la pile:

```
x . \ affiche l'adresse
```

Nous pouvons alors aller chercher ou stocker à cette adresse :

```
0 variable x
3 x !
x @ . \ affiche: 3
```

## Constantes

Une constante est un nombre que vous ne voudriez pas changer pendant l'exécution d'un programme. Le résultat de l'exécution du mot associé à une constante est la valeur des données restant sur la pile.

```
\ définit les pins VSPI
6 constant VSPI_MISO
5 constant VSPI_MOSI
4 constant VSPI_SCLK
7 constant VSPI_CS

\ définit la fréquence du port SPI
```

```
4000000 constant SPI_FREQ
```

## Outils de base pour l'allocation de mémoire

Les mots **create** et **allot** sont les outils de base pour réserver un espace mémoire et y attacher une étiquette. Par exemple, la transcription suivante montre une nouvelle entrée de dictionnaire **graphic-array** :

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

Lorsqu'il est exécuté, le mot **graphic-array** poussera l'adresse de la première entrée.

Nous pouvons maintenant accéder à la mémoire allouée à **graphic-array** en utilisant les mots de récupération et de stockage expliqués plus tôt. Pour calculer l'adresse du troisième octet attribué à **graphic-array** on peut écrire **graphic-array 2 +**, en se rappelant que les indices commencent à 0.

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

## Les nombres réels avec MECRISP Forth

Avec MECRISP Forth, les nombres réels sont saisis en mettant le caractère « , » (virgule) dans le nombre saisi :

```
3,2 f.      \ affiche: 3,1999999995343387126922607421875
3,3 f.      \ affiche: 3,29999999981373548507690429687500
4,0 f.      \ affiche: 4,00000000000000000000000000000000
4,5 f.      \ affiche: 4,50000000000000000000000000000000
11,25 f.    \ affiche: 11,25000000000000000000000000000000
11,1234 f.  \ affiche: 11,12339999992400407791137695312500
```

Vous l'avez compris, certaines valeurs décimales induisent des erreurs. On utilisera donc les nombres réels seulement dans certaines situations.

## Nombres réels sur 32 bits

Sur un processeur sans unité de calcul flottant, toutes les opérations sont effectuées par logiciel via la bibliothèque du compilateur C (ou mots Forth) et ne sont pas visibles par le programmeur. Mais les performances sont très faibles. Sur un processeur doté d'une unité de calcul flottante, toutes les opérations sont entièrement réalisées matériellement en un seul cycle, pour la plupart des instructions.

Le compilateur C (ou Forth) n'utilise pas sa propre bibliothèque à virgule flottante mais génère directement des instructions natives FPU.

Voici comment est organisé un nombre réel sur 32 bits dans MECRISP Forth :

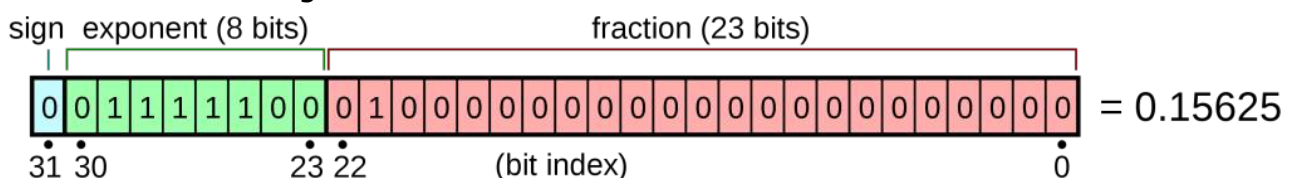


Figure 13: structure d'un nombre réel sur 32 bits

La portée des nombres pouvant être ainsi utilisés est dans l'intervalle  $[1.18E-38 \rightarrow 3.40E38]$ . Plus la partie entière sera grande, moins on peut exploiter de partie décimale. Même un nombre aussi simple que  $1/10$  sera sujet à des défauts de représentation :

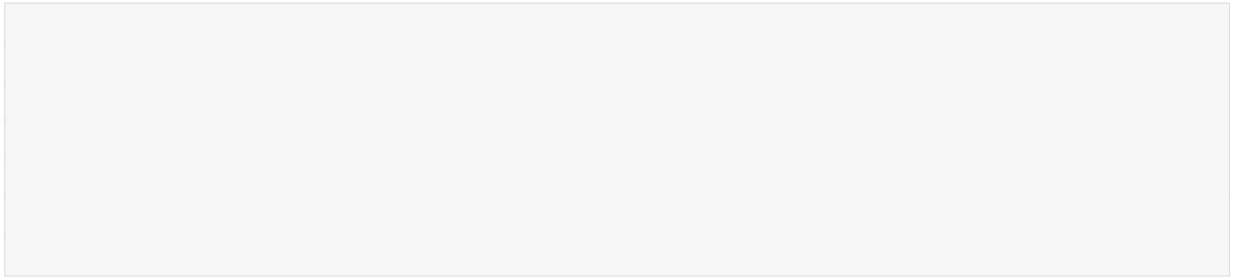
```
0,1 f.      \ affiche: 0,09999999986030161380767822265625
```

On réservera donc l'utilisation des nombres réels pour certaines utilisation. Ici, en trigonométrie :

```
30 sin f.    \ affiche: 0,50000000256113708019256591796875
45 sin f.    \ affiche: 0,71579438890330493450164794921875
```

Ici, si le sinus de  $30^\circ$  est exact, il n'en est pas de même pour celui de  $45^\circ$  qui devrait être 0.70710678...





À éplucher : <https://www.spyr.ch/twiki/bin/view/MecrispCube/FloatingPointUnit>

# Affichage des nombres et chaînes de caractères

## Changement de base numérique

FORTH ne traite pas n'importe quels nombres. Ceux que vous avez utilisés en essayant les précédents exemples sont des entiers signés simple précision. Le domaine de définition des entiers 32 bits est compris -2147483648 à 2147483647. Exemple :

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

Ces nombres peuvent être traités dans n'importe quelle base numérique, toutes les bases numériques situées entre 2 et 36 étant valides :

```
255 HEX . DECIMAL \ affiche FF
```

On peut choisir une base numérique encore plus grande, mais les symboles disponibles sortiront de l'ensemble alpha-numérique [0..9,A..Z] et risquent de devenir incohérents.

La base numérique courante est contrôlée par une variable nommée **BASE** et dont le contenu peut être modifié. Ainsi, pour passer en binaire, il suffit de stocker la valeur **2** dans **BASE**. Exemple:

```
2 BASE !
```

et de taper **DECIMAL** pour revenir à la base numérique décimale.

MECRISP Forth dispose de deux mots pré-définis permettant de sélectionner différentes bases numériques :

- **DECIMAL** pour sélectionner la base numérique décimale. C'est la base numérique prise par défaut au démarrage de MECRISP Forth ;
- **HEX** pour sélectionner la base numérique hexadécimale.
- **BINARY** pour sélectionner la base numérique binaire.

Dès sélection d'une de ces bases numériques, les nombres littéraux seront interprétés, affichés ou traités dans cette base. Tout nombre entré précédemment dans une base numérique différente de la base numérique courante est automatiquement converti dans la base numérique actuelle. Exemple :

```
DECIMAL      \ base en décimal
255          \ empile 255
HEX          \ sélectionne base hexadécimale
1+           \ incrémente 255 devient 256
.            \ affiche 100
```

On peut définir sa propre base numérique en définissant le mot approprié ou en stockant cette base dans **BASE**. Exemple :

```
: BINARY ( ---)      \ sélectionne la base numérique binaire
  2 BASE ! ;
DECIMAL 255 BINARY .  \ affiche      11111111
```

Le contenu de **BASE** peut être empilé comme le contenu de n'importe quelle autre variable :

```
0 VARIABLE RANGE_BASE \ définition de variable RANGE-BASE
BASE @ RANGE_BASE !   \ stockage contenu BASE dans RANGE-BASE
HEX FF 10 + .         \ affiche 10F
RANGE_BASE @ BASE !   \ restaure BASE avec contenu de RANGE-BASE
```

Dans une définition **:**, le contenu de **BASE** peut transiter par la pile de retour:

```
: operation ( ---)
  BASE @ >R          \ stocke BASE sur pile de retour
  HEX FF 10 + .      \ opération du précédent exemple
  R> BASE ! ;        \ restaure valeur initiale de BASE
```

**ATTENTION:** les mots **>R** et **R>** ne sont pas exploitables en mode interprété. Vous ne pouvez utiliser ces mots que dans une définition qui sera compilée.

## Définition de nouveaux formats d'affichage

Forth dispose de primitives permettant d'adapter l'affichage d'un nombre à un format quelconque. Avec MECRISP Forth, ces primitives traitent les nombres entiers :

- **<#** débute une séquence de définition de format ;
- **#** insère un digit dans une séquence de définition de format ;
- **#S** équivaut à une succession de **#** ;
- **HOLD** insère un caractère dans une définition de format ;
- **#>** achève une définition de format et laisse sur la pile l'adresse et la longueur de la chaîne contenant le nombre à afficher.

Ces mots ne sont utilisables qu'au sein d'une définition. Exemple, soit à afficher un nombre exprimant un montant libellé en euros avec la virgule comme séparateur décimal :

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

Exemples d'exécution :

```
35 .EUROS      \ affiche      0,35 EUR
3575 .EUROS    \ affiche      35,75 EUR
```

```
1015 3575 + .EUROS \ affiche 45,90 EUR
```

Dans la définition de **EUROS**, le mot **<#** débute la séquence de définition de format d'affichage. Les deux mots **#** placent les chiffres des unités et des dizaines dans la chaîne de caractère. Le mot **HOLD** place le caractère **,** (virgule) à la suite des deux chiffres de droite, le mot **#S** complète le format d'affichage avec les chiffres non nuls à la suite de **,** . Le mot **#>** ferme la définition de format et dépose sur la pile l'adresse et la longueur de la chaîne contenant les digits du nombre à afficher. Le mot **TYPE** affiche cette chaîne de caractères.

En exécution, une séquence de format d'affichage traite exclusivement des nombres entiers 32 bits signés ou non signés. La concaténation des différents éléments de la chaîne se fait de droite à gauche, c'est à dire en commençant par les chiffres les moins significatifs.

Le traitement d'un nombre par une séquence de format d'affichage est exécutée en fonction de la base numérique courante. La base numérique peut être modifiée entre deux digits.

Voici un exemple plus complexe démontrant la compacité du FORTH. Il s'agit d'écrire un programme convertissant un nombre quelconque de secondes au format HH:MM:SS:

```
: :00 ( ---)
  DECIMAL #          \ insertion digit unité en décimal
  6 BASE !           \ sélection base 6
  #                  \ insertion digit dizaine
  [char] : HOLD      \ insertion caractère :
  DECIMAL ;          \ retour base décimale
: HMS ( n ---)       \ affiche nombre secondes format HH:MM:SS
  <# :00 :00 #S #> TYPE SPACE ;
```

Exemples d'exécution:

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

Explication : le système d'affichage des secondes et des minutes est appelé système sexagésimal. Les **unités** sont exprimées dans la base numérique décimale, les **dizaines** sont exprimées dans la base six. Le mot **:00** gère la conversion des unités et des dizaines dans ces deux bases pour la mise au format des chiffres correspondants aux secondes et aux minutes. Pour les heures, les chiffres sont tous décimaux.

Autre exemple, soit à définir un programme convertissant un nombre entier simple précision décimal en binaire et l'affichant au 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

```

Exemple d'exécution :

```

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

```

Encore un exemple, soit à créer un agenda téléphonique où l'on associe à un patronyme un ou plusieurs numéros de téléphone. On définit un mot par patronyme :

```

: .## ( ---)
    # # [char] . HOLD ;
: .TEL ( d ---)
    CR <# .## .## .## .## # # #> TYPE CR ;
: DUGENOU ( ---)
    0618051254 .TEL ;
dugenou \ display : 06.18.05.12.54

```

Ce répertoire téléphonique, qui peut être compilé depuis un fichier source, est facilement modifiable, et bien que les noms ne soient pas classés, la recherche y est extrêmement rapide.

## Affichage des caractères et chaînes de caractères

L'affichage d'un caractère est réalisé par le mot **EMIT**:

```

65 EMIT           \ affiche A

```

Les caractères affichables sont compris dans l'intervalle 32..255. Les codes compris entre 0 et 31 seront également affichés, sous réserve de certains caractères exécutés comme des codes de contrôle. Voici une définition affichant tout le jeu de caractères de la table ASCII :

```

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

```

```

cr 0 #out !
128 32
DO
    I 3 .R SPACE          \ affiche code du caractère
    4 #out+!
    I EMIT 2 SPACES       \ affiche caractère
    3 #out+!
    #out @ 77 =
    IF
        CR 0 #out !
    THEN
        LOOP ;

```

L'exécution de **JEU-ASCII** affiche les codes ASCII et les caractères dont le code est compris entre 32 et 127. Pour afficher la table équivalente avec les codes ASCII en hexadécimal, taper **HEX JEU-ASCII** :

```

hex jeu-ascii
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

```

Les chaînes de caractères sont affichées de diverses manières. La première, utilisable en compilation seulement, affiche une chaîne de caractères délimitée par le caractère " (guillemet) :

```

: TITRE ." MENU GENERAL" ;
    TITRE      \ affiche      MENU GENERAL

```

La chaîne est séparée du mot **."** par au moins un caractère espace.

Une chaîne de caractères peut aussi être compilée par le mot **s"** et délimitée par le caractère " (guillemet) :

```

: LIGNE1 ( --- adr len)
    S" E..Enregistrement de données" ;

```

L'exécution de **LIGNE1** dépose sur la pile de données l'adresse et la longueur de la chaîne compilée dans la définition. L'affichage est réalisé par le mot **TYPE** :

```

LIGNE1 TYPE      \ affiche E..Enregistrement de données

```

En fin d'affichage d'une chaîne de caractères, le retour à la ligne doit être provoqué s'il est souhaité :

```

CR TITRE CR CR LIGNE1 TYPE CR
\ affiche

```

```
\ MENU GENERAL  
\   
\ E..Enregistrement de données
```

Un ou plusieurs espaces peuvent être ajoutés en début ou fin d'affichage d'une chaîne alphanumérique :

```
SPACE          \ affiche un caractère espace  
10 SPACES      \ affiche 10 caractères espace
```

# 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 MECRISP Forth :

```
: 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. Ça 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.

MECRISP Forth accède aux adresses 32 bits avec les mots **!** Et **@**. Exemple :

```
0 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 14: 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 apprend à déchiffrer les labels. Ici, **GPIO\_OE\_SET** signifie quasiment « **GPIO Output Enable 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 définit 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 15: 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	<b>GPIO25_STATUS</b>	GPIO status
0x0cc	<b>GPIO25_CTRL</b>	GPIO control including function select and overrides.

Figure 16: Registre de contrôle de GPIO25

Dans la colonne de gauche, un décalage **\$0CC**. Ce décalage doit s'appliquer au 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 fère 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'intérêt 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, permet de poser les premières pierres d'un immense édifice.



# Commande de relais Reed par GPIO

## Piloter un relais Reed

Piloter une LED, c'est bien. Mais piloter de vrais appareils, c'est mieux. Mais dès qu'on souhaite piloter d'autres appareils se posent certains problèmes:

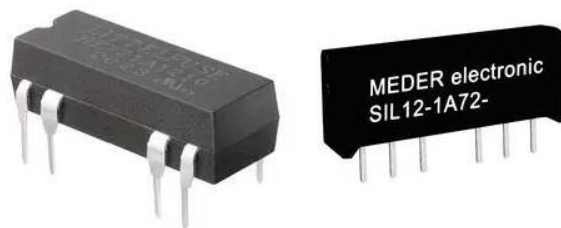
- puissance nécessaire pour activer/désactiver les appareils
- garantir une isolation galvanique entre la carte RP pico et les appareils à piloter;
- disposer d'un interface de pilotage dont les tensions sont compatibles avec la tension de commande et la tension et puissance à commander.

On ne pilote pas une lampe LED 230V comme on commande un moteur électrique de piscine de près de 1000W sous 230V.

Le pilotage de circuits de puissance nécessite des relais adaptés. Or, ces relais ne peuvent pas se commander directement depuis une carte RP pico, sauf à disposer d'une activation électronique. Oubliez les relais de kit ARDUINO. Ces relais ne tiennent pas dans le temps pour la commande de gros circuits. Nous verrons en fin d'article certaines solutions adaptées à des cas précis.

## Le relais REED

Les relais REED sont adaptés au pas des composants pour cartes électroniques:



*Figure 17: deux exemples de relais REED*

Certains de ces relais sont sensibles aux champs magnétiques intenses et peuvent être activés mécaniquement par la proximité d'un aimant permanent.

Principaux intérêt des relais REED:

- petits et légers: Les relais REED sont très petits et légers, ce qui les rend idéaux pour les applications compactes.
- coût abordable: Les relais REED sont relativement abordables, ce qui les rend attrayants pour les applications à petit budget.

- haute fiabilité: Les relais REED sont très fiables et peuvent supporter des millions de cycles de commutation.
- résistance aux vibrations: Les relais REED sont résistants aux vibrations, ce qui les rend idéaux pour les applications mobiles.
- résistance aux chocs: Les relais REED sont résistants aux chocs, ce qui les rend idéaux pour les applications où des chocs peuvent se produire.

Ces relais ne conviennent pas pour la commutation de courants importants. On réservera leur usage pour la commande de relais industriels.

## Pilotage du relais REED par la carte RP pico

La commande du relais REED est triviale. Elle s'effectue exactement comme l'allumage



Figure 18: relais avec pins au pas des cartes standard

d'une LED connectée à un port GPIO.

Ici, un relais REED à 4 pins. Les pins sont numérotés 1 3 5 et 7 en partant de la gauche.

Branchement des connecteurs à la carte RP pico:

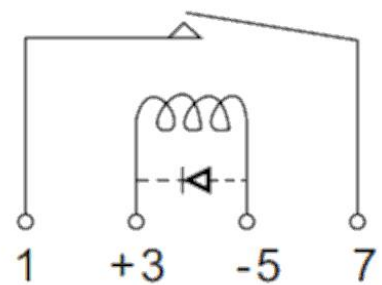
- connecteur +3 vers le GPIO qui active le relais
- connecteur -5 vers GND de la carte RP pico

La majorité du code FORTH reprend celui du précédent article "Initiation aux ports GPIO".

Dans notre exemple, on connecte le GPIO 07 vers le connecteur +3 du relais:

```
07 constant MY_RELAY    \ Reed Relay on GPIO 7

: relay_init ( -- )
  MY_RELAY GPIO_FUNC_SIO gpio_set_function
  MY_RELAY GPIO-OUT gpio_set_dir
;
```



Ensuite, on définit simplement les mots permettant d'activer ou désactiver le relais reed:

```

: relay_state ( state -- )
    MY_RELAY swap gpio_put
;

: relay_on ( -- )
    GPIO_HIGH relay_state
;

: relay_off ( -- )
    GPIO_LOW  relay_state
;

```

## Gestion d'un relais industriel bistable

Voici une petite variante permettant de commander un relais industriel bistable. Un relais bistable se verrouille ou se déverrouille selon la commande reçue. Cette commande est une simple impulsion. Pour notre cas, les impulsions seront transmises par deux relais REED connectés à notre carte RP pico. Exemple de relais bistable:



Figure 19: relais de puissance bistable pour rail DIN

Les relais bistables ont plusieurs avantages:

- ne nécessitent aucune alimentation de la bobine pour conserver l'état ouvert ou fermé;
- conservent l'état ouvert ou fermé en cas de coupure électrique;
- faciles à mettre en œuvre.

Le modèle présenté ici commute un courant pouvant atteindre 16A sous 230V. C'est suffisant pour commander une pompe de piscine par exemple, un système d'éclairage collectif, un radiateur électrique de 2000W....

On définit d'abord les deux ports GPIO qui vont activer deux relais REED distincts:

```

07 constant MY_RELAY_ON    \ Reed Relay on GPIO 7
08 constant MY_RELAY_OFF   \ Reed Relay on GPIO 8

```

Ensuite, on initialise ces deux ports GPIO:

```

: relays_init ( -- )
    MY_RELAY_ON  GPIO_FUNC_SIO gpio_set_function

```

```
MY_RELAY_ON    GPIO-OUT gpio_set_dir
MY_RELAY_OFF   GPIO_FUNC_SIO gpio_set_function
MY_RELAY_OFF   GPIO-OUT gpio_set_dir
;
```

Et pour finir, on définit deux mots permettant d'activer ou désactiver le relais bistable:

```
200 value ACTION_DELAY

: bistable_on ( -- )
    MY_RELAY_ON    GPIO_HIGH gpio_put
    ACTION_DELAY ms
    MY_RELAY_ON    GPIO_LOW  gpio_put
;

: bistable_off ( -- )
    MY_RELAY_OFF   GPIO_HIGH gpio_put
    ACTION_DELAY ms
    MY_RELAY_OFF   GPIO_LOW  gpio_put
;
```

Ici, on met un délai de 300 millisecondes pour gérer la durée d'activation ou désactivation du relais bistable. Cette valeur peut être ajustée.

## Les ports GPIO en entrée

Dans le précédent article nous avons abordé la gestion des ports GPIO en sortie pour commander une LED. Ces mêmes ports GPIO peuvent aussi être gérés en entrée pour tester des signaux. Ici, nous allons simplement tester l'état d'un interrupteur.

### L'interrupteur

J'ai choisi ce type de contacteur, qui est un contacteur de fin de course, à deux états:



Figure 20: contacteur de fin de course

- au repos, le contact est fermé entre Com et NC

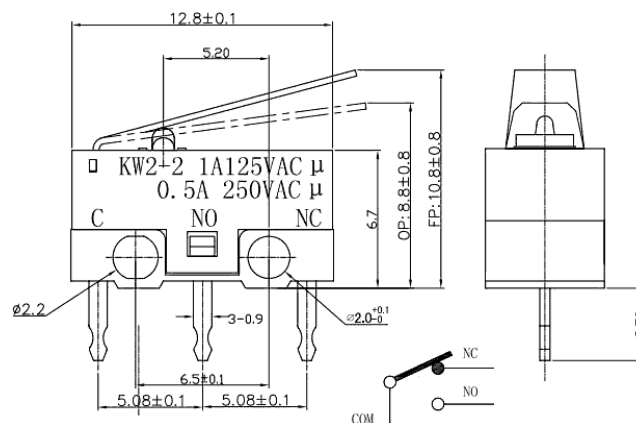


Figure 21: contacteur de fin de course: schéma

- en action, le contact est fermé entre Com et NO

Ce contacteur peut être facilement utilisé sur une plaque d'essai. Il dispose de deux trous permettant un montage facile. Le contacteur est très sensible. Il se déclenche sur une faible pression. Le relâchement de la pression ramène le contacteur à son état initial.

## Gestion par GPIO

Le câblage du contacteur est réalisé de cette manière:

- **pin 36** 3V3 (OUT) <--> entrée **Com** contacteur
- **pin 10** GP7 <--> sortie **NO** contacteur

Pour commencer, on définit le mot `gpio_get`:

```
\ Get state of a single specified GPIO
: gpio_get ( gpio -- state )
  PIN_MASK
  GPIO_IN @ and
;
```

Le mot `gpio_get` récupère l'état du contacteur:

- si state = 0, le contacteur n'est pas actif;
- si state <> 0, le contacteur est actif. La valeur renvoyée correspond au masque binaire du GPIO activé;

Voici comment est initialisé le contacteur:

```
7 constant SWITCH      \ switch to GPIO 07

: init-switch ( -- )
  SWITCH gpio_init
;
```

On peut maintenant tester le contacteur. N'activez pas le contacteur:

```
init-switch
7 gpio_get . \ display: 0
```

Activez le contacteur:

```
7 gpio_get . \ display: 128
```

En binaire, on aurait eu la valeur **10000000**. Le bit à 1 est bien le septième bit. Pour rappel, les bits sont numérotés à partir de 0. Le bit b7 est donc ici à 1. Cette position 7 est en concordance avec le GPIO 07 sur lequel est connecté notre contacteur.

## Exercice pratique avec notre contacteur

Notre cas pratique consiste simplement à allumer ou éteindre la LED qui est montée sur la carte RP pico en lien avec le GPIO 25. Un appui sur le contacteur allume la LED. Si on relâche le contacteur, cette LED reste allumée. Un nouvel appui éteint cette même LED.

On commence par redéfinir l'initialisation des deux ports GPIO que nous utiliserons:

```
7 constant SWITCH      \ switch to GPIO 07
25 constant ONBOARD_LED \ led to GPIO 25
```

```

: gpios.init ( -- )
  SWITCH gpio_init
  ONBOARD_LED gpio_init
  ONBOARD_LED GPIO-OUT gpio_set_dir
;

```

On définit ensuite le mot **led.toggle** qui alterne l'état de la LED rattachée au GPIO 25:

```

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

```

Le mot **switch.wait** attend 20 millisecondes. Cette temporisation est nécessaire pour éviter des rebonds de contact. Puis on bascule l'état de la LED rattachée au GPIO 25. Enfin, la boucle se poursuit tant que le contact est actif. Le relâchement du contacteur provoque la sortie de boucle:

```

: switch.wait ( -- )
  20 ms \ anti-bounce delay
  ONBOARD_LED led.toggle
  begin
    100 ms
    SWITCH gpio_get
  0= until
;

```

Cette dernière définition initialise les ports GPIO.

Puis on exécute une boucle qui ne sera interrompue que par appui sur une touche du clavier.

Dans la boucle, on attend que le contacteur soit actif. Si c'est le cas, on exécute le mot **switch.wait**:

```

: main-loop ( -- )
  gpios.init
  begin
    SWITCH gpio_get if
      switch.wait
    then
  key? until
;

```

Les appuis successifs sur le contacteur alternent l'état de la LED rattachée au GPIO 25 exactement comme le ferait un télérupteur domestique pour commander un éclairage depuis plusieurs points distincts.

# Le multitâche avec MECRISP Forth

FORTH est un langage multi-tâche. Il est multi-tâche dès son origine. Déjà sous MS-DOS – qui n'était pas multi-tâche – FORTH savait exécuter plusieurs tâches distinctes. Et c'est toujours le cas avec MECRISP Forth. C'est ce que nous allons voir en détail dans ce chapitre.

## Définition du multi-tâche

Le système FORTH est bien plus qu'un simple interpréteur/compilateur. C'est un système complet. FORTH intègre un système de gestion de tâches par jeton. Ce mécanisme est totalement transparent pour l'utilisateur. Bien que très efficace, il a quelques menus inconvénients :

- une tâche peut être préemptive. En clair, elle peut empêcher le passage de jeton et bloquer le système ;
- du fait d'un passage de jeton, les tâches ne peuvent pas être synchrones

Mais dans la très grande majorité des situations, la gestion multi-tâche est efficace.

Reprenons le code du clignotement de la LED implantée sur la carte RP pico rattachée au GPIO 25 :

```
25 constant ONBOARD_LED

: led.init ( -- )
  ONBOARD_LED GPIO_FUNC_SIO gpio_set_function
  ONBOARD_LED GPIO-OUT gpio_set_dir
;

: blink ( -- )
  led.init
  begin
    ONBOARD_LED PIN_MASK GPIO_OUT_XOR !
    300 ms
  again
;
```

Si on exécute **blink**, la LED implantée sur la carte RP pico va bien clignoter. Mais c'est tout ce qu'on peut faire. Car la partie de code gérant le clignotement est située entre **begin** et **again**, en rouge dans le code ci-dessus.

La question est : « comment exécuter un autre code pendant que la LED clignote ? ». C'est une excellente question, je vous remercie de l'avoir posée.

A moins de réaliser un code complexe, il n'y a aucune solution élégante et facile à maintenir. Ou à modifier.



Le multi-tâche est une solution permettant de décrire quelle partie de code est défini dans une tâche, puis exécuter cette tâche en laissant disponible le reste du système FORTH. Pour définir une tâche, on utilise le mot de création **task:** comme ceci :

```
task: blinktask
```

Je dirai que le plus dur est fait ! Enfin, presque... Car il faut maintenant indiquer la partie de code à exécuter dans cette tâche. C'est ce qu'on va faire dans la définition de **blink** :

```
: blink& ( -- )
  led.init
  blinktask activate
  begin
    ONBOARD_LED PIN_MASK GPIO_OUT_XOR !
    300 ms
  again
;
```

Notez au passage que le mot **blink** est devenu **blink&** dans notre code. Ce renommage n'a rien d'obligatoire. Le rajout du caractère **&** après **blink** est surtout une habitude héritée du monde UNIX où les parties de code multi-tâches étaient marquées de cette manière.

## Activer et désactiver des tâches

Pour activer le moteur multi-tâche, on exécute :

```
multitask
```

Puis on exécute une seule fois la tâche à insérer dans la boucle de gestion multi-tâche :

```
blink&
```

Dès cet instant, la LED implantée sur la carte RP pico clignote. Jusque là, rien d'étonnant. Mais ce qui peut surprendre le programmeur débutant en FORTH, c'est d'avoir toujours la main pour taper d'autres commandes FORTH !

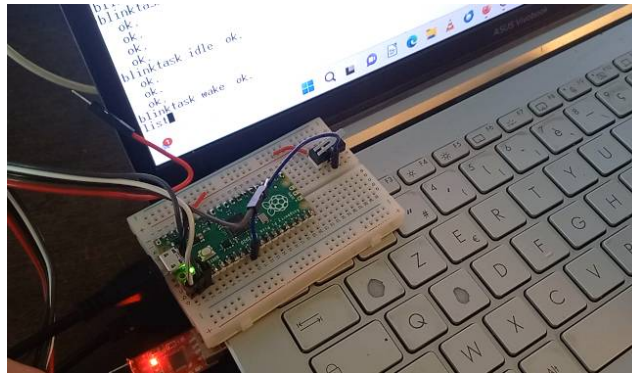


Figure 22: multi-tâche actif avec MECRISP  
Forth

Ça surprend toujours la première fois ! Nous avons pourtant une boucle **begin..again** active. Théoriquement on ne peut en sortir, sauf à débrancher la carte RP pico. Mais on doit se rendre à l'évidence, on a bien la main sur l'interpréteur FORTH.

Pour interrompre cette tâche, on exécute :

```
blinktask idle
```

Et pour la ré-activer, on exécute :

```
blinktask wake
```

Allez-y, essayez !

Vous commencez maintenant à comprendre toute la puissance du langage FORTH. Car nous venons tout simplement d'ouvrir la porte à des possibilités de programmation extraordinaires.

Pour connaître le nombre de tâches actives, on exécute **tasks**.

Pour revenir à une gestion de tâche unique, exécuter **singletask**. Ceci interrompt toutes les tâches autre que la tâche gérant l'interpréteur FORTH.

En théorie, il est possible d'exécuter plusieurs tâches en même temps. Mais si une tâche particulière nécessite une réactivité quasi instantanée, cette réactivité sera limitée par le passage de jeton des autres tâches. Si vous effectuez du traitement de signal à très haute fréquence, on évitera le multi-tâche.

Toutes les variables définies dans FORTH peuvent être lues ou modifiées par une tâche quelconque. Plusieurs tâches distinctes peuvent appeler un même mot. Les paramètres posés sur la pile par une tâche restent spécifiques à cette tâche.

La tâche système principale ne peut être interrompue.

# Contenu détaillé des mots MECRISP Forth

MECRISP Forth ne met à disposition implicitement que le vocabulaire FORTH.

Vous trouverez ici la liste de tous les mots FORTH définis dans MECRISP Stellaris. Certains mots sont présentés avec un lien coloré :

[align](#) est un mot FORTH ordinaire ;

**CONSTANT** est mot de définition ;

**begin** marque une structure de contrôle ;

**LED** est un mot défini par **constant**, **variable** ou **value** ;

Ces mots sont affichés par ordre alphabétique.

-inf	-roll	<a href="#">-rot</a>	<a href="#">.</a>	<a href="#">.</a>	<a href="#">!</a>	<a href="#">?do</a>
<a href="#">?dup</a>	<a href="#">?of</a>	<a href="#">.</a>	<a href="#">."</a>	<a href="#">.digit</a>	<a href="#">.s</a>	(create)
(dp)	(find)	(latest)	(pause)	[	[']	[char]
@	*	*/	*/mod	/	/mod	#
#S	+	+	+inf	<a href="#">+loop</a>	<	<#
<>	<builds	=	>	>< ,	>=	>in
0-foldable	0<	0<>	0=	0to1-atan	0to1sqrt	1-
1+	1over1nof2	2-	2-foldable	2-rot	2!	2@
2/	2+	2>r	<a href="#">2constant</a>	<a href="#">2drop</a>	<a href="#">2dup</a>	2nip
2r@	2r>	2rdrop	2rot	2swap	2tuck	2variable
4-foldable	5-foldable	6-foldable	7-foldable	<a href="#">abs</a>	<a href="#">accept</a>	<a href="#">acos</a>
adcs	add	addr.	addrinflash?	addrinram?	adds	adds&subs
<a href="#">again</a>	ahead	<a href="#">align</a>	<a href="#">aligned</a>	<a href="#">allot</a>	<a href="#">and</a>	<a href="#">ands</a>
<a href="#">asin</a>	asrs	asrsr	<a href="#">atan</a>	atan-coef	atan-table	b
<a href="#">base</a>	base-ivl-atan	bcc	bcs	<a href="#">begin</a>	<a href="#">beg</a>	bge
bgt	bhi	bhs	<a href="#">bic</a>	<a href="#">bic!</a>	bics	<a href="#">binary</a>
<a href="#">bit@</a>	<a href="#">bl</a>	ble	blo	bls	blt	blx
<a href="#">bne</a>	boot-task	bpl	buffer:	bvc	bvs	bx
<a href="#">c!</a>	c"	<a href="#">c@</a>	<a href="#">c+</a>	call,	<a href="#">case</a>	catch
<a href="#">cbis!</a>	<a href="#">cbit@</a>	<a href="#">cell+</a>	<a href="#">cells</a>	cexpect	cflash!	<a href="#">char</a>
check+label						
cjump,	<a href="#">clz</a>	cmp	compare	<a href="#">compileonly</a>	<a href="#">compiletoflash</a>	
<a href="#">compiletoram</a>		<a href="#">compiletoram?</a>		connect-flash	const.	<a href="#">constant</a>
<a href="#">cos</a>	cos-coef	count	<a href="#">cr</a>	<a href="#">create</a>	ctype	current-source
<a href="#">cxor!</a>	<a href="#">cycles</a>	d-	d.	d/	d/mod	d+
d<>	d=	d>	d0<	d0=	d2*	d2/
<a href="#">decimal</a>	<a href="#">deg-90to90</a>	deg0to360	<a href="#">deg2rad</a>	delay-cycles	<a href="#">depth</a>	dabs
destination-r0						
<a href="#">dictionarynext</a>		<a href="#">dictionarystart</a>	<a href="#">digit</a>	<a href="#">dint</a>	disasm	disasm-\$
disasm-fetch		disasm-step	disasm-string	dnegate	<a href="#">do</a>	<a href="#">does&gt;</a>
double-operand	<a href="#">drop</a>	dshl	dshr	du<	du>	<a href="#">dump</a>
dump16	<a href="#">dup</a>	<a href="#">eint</a>	eint?	<a href="#">else</a>	<a href="#">emit</a>	<a href="#">emit?</a>
<a href="#">endof</a>	enter-xip	eors	erase-range	erase#	ersteszeichen	<a href="#">evaluate</a>
<a href="#">even</a>	<a href="#">execute</a>	<a href="#">exit</a>	exit-xip	exp	exp-1to1	exp-coef
f.n	f*	f/	f#	f#S	f>s	<a href="#">false</a>
<a href="#">find</a>	<a href="#">flashvar-here</a>	<a href="#">floor</a>	flush-cache	forgetram	h,	<a href="#">h!</a>
h.s	<a href="#">h@</a>	<a href="#">h+</a>	half-q1-cos-rad	half-q1-sin-rad		halign
handler	<a href="#">hbit!</a>	<a href="#">hbit!</a>	<a href="#">hbit@</a>	<a href="#">here</a>	<a href="#">hex</a>	<a href="#">hex.</a>
						hflash!

<a href="#">hold</a>	hold<	hook-emit	hook-emit?	hook-find	hook-key	hook-key?	hook-pause
hook-quit	<a href="#">hxor!</a>	<a href="#">i</a>	<a href="#">idle</a>	<a href="#">if</a>	image>spi-offset		imm3.
imm3<<1.	imm3<<2.	imm5.	imm5<<1.	imm5<<2.	imm7<<2.	imm8.	imm8<<1.
imm8<<2.	<a href="#">immediate</a>	<a href="#">inline</a>	inline,	insert	interpret	<a href="#">ipsr</a>	irq-
ADC_FIFO							
irq-CLOCKS	irq-collection		irq-DMA_0	irq-DMA_1	irq-fault	irq-I2C0	irq-I2C1
irq-IO_BANK0		irq-IO_QSPI	irq-pendsv	irq-PIO0_0	irq-PIO0_1	irq-PIO1_0	irq-PIO1_1
irq-PWM_WRAP		irq-RTC	irq-SIO_PROC0		irq-SIO_PROC1		irq-SPI0
irq-SPI1	irq-svcall	irq-systick	irq-TIMER_0	irq-TIMER_1	irq-TIMER_2	irq-TIMER_3	irq-UART0
irq-UART1	irq-USBCTRL	irq-XIP	<a href="#">j</a>	jump-destination	jump,		jumps
<a href="#">k</a>	<a href="#">key</a>	<a href="#">key?</a>	l-:	l+:	label-	label--	label---
label-f1	label-f2	label-f3	label-f4	label-f5	label-f6	label-f7	label-f8
ldmia	ldr	ldr=	ldrb	ldrh	ldrsh	ldrsh	leave
<a href="#">list</a>	literal,	ln	ln10overln2	lnof2	load&store	load#	<a href="#">log10</a>
log10of2	log2	log2-1to2	<a href="#">loop</a>	<a href="#">lshift</a>	lsls	lslsr	lsrs
lsrsr	m*	m/mod	<a href="#">max</a>	memstamp	<a href="#">min</a>	<a href="#">mod</a>	mov
mov&add	<a href="#">move</a>	movs	movs&cmp	<a href="#">ms</a>	mults	multitask	mvns
name.	<a href="#">negate</a>	<a href="#">new</a>	next-task	<a href="#">nip</a>	<a href="#">nop</a>	<a href="#">not</a>	number
numbertable	nvariable	<a href="#">of</a>	opcode?	operandenparser		<a href="#">or</a>	orrs
<a href="#">over</a>	<a href="#">parse</a>	<a href="#">pause</a>	<a href="#">pi/2</a>	<a href="#">pi/4</a>	pick	pop	postpone
<a href="#">pow10</a>	<a href="#">pow2</a>	prepairetask		previous	program-range		push
push&pop	q1-sin-rad	q1toq4-sin	<a href="#">query</a>	quit	r@	<a href="#">r&gt;</a>	<a href="#">rad2deg</a>
<a href="#">rdepth</a>	<a href="#">rdrop</a>	<a href="#">recurse</a>	reg.	reg16.	reg16split.		register.
<a href="#">registerlist.</a>		registerliteral,		<a href="#">registerparser</a>		registerparser16	
remember+jump		remove	<a href="#">repeat</a>	<a href="#">reset</a>	restart	ret,	rev
rev16	revsh	<a href="#">rol</a>	roll	rom-code	rom-data	<a href="#">ror</a>	rors
<a href="#">rot</a>	<a href="#">rp!</a>	<a href="#">rp@</a>	rpick	rrotate	<a href="#">rshift</a>	<a href="#">s"</a>	<a href="#">s&gt;d</a>
s>f	<a href="#">save</a>	save-task	save#	sbscs	<a href="#">see</a>	seec	<a href="#">serial-</a>
<a href="#">emit</a>							
serial-emit?		serial-key	serial-key?	setflags	<a href="#">setsource</a>	sev	shifts-imm
<a href="#">shl</a>	<a href="#">shr</a>	sign	signedloads	sin	sin-coef	single-operand	
singletask	skipstring	<a href="#">smudge</a>	<a href="#">source</a>	<a href="#">sp!</a>	<a href="#">sp@</a>	<a href="#">space</a>	<a href="#">spaces</a>
sqrtr	<a href="#">stackspace</a>	<a href="#">state</a>	stmia	stmia&ldmia	<a href="#">stop</a>	str	strb
strh	string,	subs	<a href="#">swap</a>	sxtb	sxth	symbolwert	<a href="#">tan</a>
task-in-list?		task-state	<a href="#">task:</a>	<a href="#">tasks</a>	<a href="#">then</a>	throw	<a href="#">tib</a>
<a href="#">TIMEHR</a>	<a href="#">TIMEHW</a>	<a href="#">TIMELR</a>	<a href="#">TIMELW</a>	<a href="#">TIMERAWH</a>	<a href="#">TIMERAWL</a>	token	<a href="#">true</a>
tst	<a href="#">tuck</a>	<a href="#">type</a>	<a href="#">u.</a>	<a href="#">u.2</a>	<a href="#">u.4</a>	<a href="#">u.8</a>	
u.ns	u.s	u*/	u*/mod	<a href="#">u/mod</a>	<a href="#">u&lt;</a>	u<=	<a href="#">u&gt;</a>
u>=	<a href="#">ud.</a>	<a href="#">ud*</a>	ud/mod	udm*	<a href="#">um*</a>	<a href="#">um/mod</a>	umax
umin	unhandled	<a href="#">unloop</a>	<a href="#">until</a>	<a href="#">unused</a>	up	<a href="#">us</a>	uxtb
uxth	<a href="#">variable</a>	<a href="#">vorneabschneiden</a>		<a href="#">wake</a>	<a href="#">wfi</a>	<a href="#">while</a>	<a href="#">words</a>
<a href="#">xor</a>	<a href="#">xor!</a>	zero-operand					

# Les registres du processeur RP2040

## Les registres SIO

Les registres SIO démarrent à l'adresse de base **\$d0000000**. Exemple de définition :

```
$d0000000 constant SIO_BASE
```

Pour définir un registre, on exploite ensuite le décalage en le rajoutant simplement à cette adresse de base :

```
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	Identifiant du cœur du processeur
\$004	GPIO_IN	Valeur d'entrée pour les broches GPIO
\$008	GPIO_HI_IN	Valeur d'entrée pour les broches QSPI
\$010	GPIO_OUT	Valeur de sortie GPIO
\$014	GPIO_OUT_SET	Valeur de sélection de sortie GPIO
\$018	GPIO_OUT_CLR	Valeur de dé-sélection de sortie GPIO
\$01C	GPIO_OUT_XOR	Valeur d'inversion par XOR de sortie GPIO
\$020	GPIO_OE	Activation de la sortie GPIO
\$024	GPIO_OE_SET	Ensemble d'activation de sortie GPIO
\$028	GPIO_OE_CLR	Ensemble de dés-activation de sortie GPIO
\$02C	GPIO_OE_XOR	Ensemble d'inversion par XOR de sortie GPIO
\$030	GPIO_HI_OUT	Valeur de sortie QSPI
\$034	GPIO_HI_OUT_SET	Valeur de sélection de sortie QSPI
\$038	GPIO_HI_OUT_CLR	Valeur de dé-sélection de sortie QSPI
\$03C	GPIO_HI_OUT_XOR	Ensemble d'inversion par XOR de sortie QSPI
\$040	GPIO_HI_OE	Activation de la sortie QSPI
\$044	GPIO_HI_OE_SET	Ensemble d'activation de sortie QSPI
\$048	GPIO_HI_OE_CLR	Ensemble de dés-activation de sortie QSPI
\$04C	GPIO_HI_OE_XOR	Ensemble d'inversion par XOR de sortie QSPI
\$050	FIFO_ST	Registre d'état pour les FIFO intercœurs (boîtes aux lettres).
\$054	FIFO_WR	Accès en écriture au TX FIFO de ce cœur

Décalage	Nom	Info
\$058	FIFO_RD	Accès en lecture au RX FIFO de ce cœur
\$05C	SPINLOCK_ST	État du verrouillage rotatif
\$060	DIV_UDIVIDEND	Dividende diviseur non signé
\$064	DIV_UDIVISOR	Diviseur diviseur non signé
\$068	DIV_SDIVIDEND	Diviseur signé dividende
\$06C	DIV_SDIVISOR	Diviseur signé diviseur
\$070	DIV_QUOTIENT	Quotient de résultat du diviseur
\$074	DIV_REMAINDER	Reste du résultat du diviseur
\$078	DIV_CSR	Registre de contrôle et d'état pour diviseur.
\$080	INTERP0_ACCUM0	Accès en lecture/écriture à l'accumulateur 0
\$084	INTERP0_ACCUM1	Accès en lecture/écriture à l'accumulateur 1
\$088	INTERP0_BASE0	Accès en lecture/écriture au registre BASE0.
\$08C	INTERP0_BASE1	Accès en lecture/écriture au registre BASE1.
\$090	INTERP0_BASE2	Accès en lecture/écriture au registre BASE2.
\$094	INTERP0_POP_LANE0	Lit le résultat LANE0 et écrit simultanément les résultats des voies dans les deux accumulateurs (POP).
\$098	INTERP0_POP_LANE1	Lit le résultat LANE1 et écrit simultanément les résultats des voies dans les deux accumulateurs (POP).
\$09C	INTERP0_POP_FULL	Lit le résultat FULL et écrit simultanément les résultats des voies dans les deux accumulateurs (POP).
\$0A0	INTERP0_PEEK_LANE0	Lit le résultat LANE0, sans altérer aucun état interne (PEEK).
\$0A4	INTERP0_PEEK_LANE1	Lit le résultat LANE1, sans altérer aucun état interne (PEEK).
\$0A8	INTERP0_PEEK_FULL	Lire le résultat COMPLET, sans altérer aucun état interne (PEEK).
\$0AC	INTERP0_CTRL_LANE0	Registre de contrôle pour la voie 0
\$0B0	INTERP0_CTRL_LANE1	Registre de contrôle pour la voie 1
\$0B4	INTERP0_ACCUM0_AD D	Les valeurs écrites ici sont ajoutées atomiquement à ACCUM0
\$0B8	INTERP0_ACCUM1_AD D	Les valeurs écrites ici sont ajoutées atomiquement à ACCUM1
\$0BC	INTERP0_BASE_1AND0	Lors de l'écriture, les 16 bits inférieurs vont simultanément à BASE0, les bits supérieurs à BASE1.
\$0C0	INTERP1_ACCUM0	Accès en lecture/écriture à l'accumulateur 0
\$0C4	INTERP1_ACCUM1	Accès en lecture/écriture à l'accumulateur 1
\$0C8	INTERP1_BASE0	Accès en lecture/écriture au registre BASE0.

Décalage	Nom	Info
\$0CC	INTERP1_BASE1	Accès en lecture/écriture au registre BASE1.
\$0D0	INTERP1_BASE2	Accès en lecture/écriture au registre BASE2.
\$0D4	INTERP1_POP_LANE0	Lit le résultat LANE0 et écrit simultanément les résultats des voies dans les deux accumulateurs (POP).
\$0D8	INTERP1_POP_LANE1	Lit le résultat LANE1 et écrit simultanément les résultats des voies dans les deux accumulateurs (POP).
\$0DC	INTERP1_POP_FULL	Lit le résultat FULL et écrit simultanément les résultats des voies dans les deux accumulateurs (POP).
\$0E0	INTERP1_PEEK_LANE0	Lit le résultat LANE0, sans altérer aucun état interne (PEEK).
\$0E4	INTERP1_PEEK_LANE1	Lit le résultat LANE1, sans altérer aucun état interne (PEEK).
\$0E8	INTERP1_PEEK_FULL	Lire le résultat FULL, sans altérer aucun état interne (PEEK).
\$0EC	INTERP1_CTRL_LANE0	Registre de contrôle pour voie 0
\$0F0	INTERP1_CTRL_LANE1	Registre de contrôle pour voie 1
\$0F4	INTERP1_ACCUM0_AD D	Les valeurs écrites ici sont ajoutées atomiquement à ACCUM0
\$0F8	INTERP1_ACCUM1_AD D	Les valeurs écrites ici sont ajoutées atomiquement à ACCUM1
\$0FC	INTERP1_BASE_1AND0	Lors de l'écriture, les 16 bits inférieurs vont simultanément à BASE0, les bits supérieurs à BASE1.
\$100	SPINLOCK0	Registre rotatif 0
\$104	SPINLOCK1	Registre rotatif 1
\$108	SPINLOCK2	Registre rotatif 2
\$10C	SPINLOCK3	Registre rotatif 3
\$110	SPINLOCK4	Registre rotatif 4
\$114	SPINLOCK5	Registre rotatif 5
\$118	SPINLOCK6	Registre rotatif 6
\$11C	SPINLOCK7	Registre rotatif 7
\$120	SPINLOCK8	Registre rotatif 8
\$124	SPINLOCK9	Registre rotatif 9
\$128	SPINLOCK10	Registre rotatif 10
\$12C	SPINLOCK11	Registre rotatif 11
\$130	SPINLOCK12	Registre rotatif 12
\$134	SPINLOCK13	Registre rotatif 13
\$138	SPINLOCK14	Registre rotatif 14



Décalage	Nom	Info
\$13C	SPINLOCK15	Registre rotatif 15
\$140	SPINLOCK16	Registre rotatif 16
\$144	SPINLOCK17	Registre rotatif 17
\$148	SPINLOCK18	Registre rotatif 18
\$14C	SPINLOCK19	Registre rotatif 19
\$150	SPINLOCK20	Registre rotatif 20
\$154	SPINLOCK21	Registre rotatif 21
\$158	SPINLOCK22	Registre rotatif 22
\$15C	SPINLOCK23	Registre rotatif 23
\$160	SPINLOCK24	Registre rotatif 24
\$164	SPINLOCK25	Registre rotatif 25
\$168	SPINLOCK26	Registre rotatif 26
\$16C	SPINLOCK27	Registre rotatif 27
\$170	SPINLOCK28	Registre rotatif 28
\$174	SPINLOCK29	Registre rotatif 29
\$178	SPINLOCK30	Registre rotatif 30
\$17C	SPINLOCK31	Registre rotatif 31

## Les registres PWM

Les registres PWM démarrent à l'adresse de base **\$40050000** . Exemple de définition :

```
$40050000  constant PWM_BASE
```

Décalage	Nom	Info
\$00	CH0_CSR	Registre de contrôle et d'état
\$04	CH0_DIV	INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$08	CH0_CTR	Accès direct au compteur PWM
\$0C	CH0_CC	Compte les valeurs de comparaison
\$10	CH0_TOP	Valeur du compteur
\$14	CH1_CSR	Registre de contrôle et d'état
\$18	CH1_DIV	INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système

Décalage	Nom	Info
		divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$1C	CH1_CTR	Accès direct au compteur PWM
\$20	CH1_CC	Compte les valeurs de comparaison
\$24	CH1_TOP	Valeur du compteur
\$28	CH2_CSR	Control and status register
\$2C	CH2_DIV	INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$30	CH2_CTR	Accès direct au compteur PWM
\$34	CH2_CC	Compte les valeurs de comparaison
\$38	CH2_TOP	Valeur du compteur
\$3C	CH3_CSR	Registre de contrôle et d'état
\$40	CH3_DIV	0x40 INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$44	CH3_CTR	Accès direct au compteur PWM
\$48	CH3_CC	Compte les valeurs de comparaison
\$4C	CH3_TOP	Valeur du compteur
\$50	CH4_CSR	Registre de contrôle et d'état
\$54	CH4_DIV	INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$58	CH4_CTR	Accès direct au compteur PWM
\$5C	CH4_CC	Compte les valeurs de comparaison
\$60	CH4_TOP	Valeur du compteur
\$64	CH5_CSR	Registre de contrôle et d'état
\$68	CH5_DIV	INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$6C	CH5_CTR	Accès direct au compteur PWM
\$70	CH5_CC	Compte les valeurs de comparaison
\$74	CH5_TOP	Valeur du compteur
\$78	CH6_CSR	Registre de contrôle et d'état

Décalage	Nom	Info
\$7C	CH6_DIV	INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$80	CH6_CTR	Accès direct au compteur PWM
\$84	CH6_CC	Compte les valeurs de comparaison
\$88	CH6_TOP	Valeur du compteur
\$8C	CH7_CSR	Registre de contrôle et d'état
\$90	CH7_DIV	INT et FRAC forment un nombre fractionnaire à virgule fixe. Le taux de comptage est la fréquence de l'horloge système divisée par ce nombre. La division fractionnaire utilise un simple sigma-delta du 1er ordre.
\$94	CH7_CTR	Accès direct au compteur PWM
\$98	CH7_CC	Compte les valeurs de comparaison
\$9C	CH7_TOP	Valeur du compteur
\$A0	EN	Ce registre alias les bits CSR_EN pour tous les canaux. L'écriture dans ce registre permet d'activer ou de désactiver plusieurs canaux simultanément, afin qu'ils puissent fonctionner en parfaite synchronisation. Pour chaque canal, il n'y a qu'un seul bit de registre EN physique, accessible via ici ou CHx_CSR.
\$A4	INTR	Interruptions brutes
\$A8	INTE	Activation d'interruption
\$AC	INTF	Force d'interruption
\$B0	INTS	Statut d'interruption après masquage et forçage

## Les registres timer

Les registres TIMER démarrent à l'adresse de base **\$40054000**. Exemple de définition :

```
$40054000 constant TIMER_BASE
```

Décalage	Nom	Info
\$00	TIMEHW	Écrit sur les bits 63:32 de time, écrivez toujours timelw avant timehw
\$04	TIMELW	Écrit sur les bits 31 : 0 des écritures de temps ne sont pas copiés dans le temps tant que timehw n'est pas écrit
\$08	TIMEHR	Lit à partir des bits 63:32 du temps, toujours lire timelr avant timehr

Décalage	Nom	Info
\$0C	TIMELR	Lit à partir des bits 31:0 du temps
\$10	ALARM0	Arme l'alarme 0 et configure l'heure à laquelle elle se déclenchera. Une fois armée, l'alarme se déclenche lorsque <code>TIMER_ALARM0 == TIMELR</code> . L'alarme se désarmera d'elle-même une fois déclenchée et pourra être désarmée plus tôt à l'aide du registre d'état ARMED.
\$14	ALARM1	Arme l'alarme 1 et configure l'heure à laquelle elle se déclenchera. Une fois armée, l'alarme se déclenche lorsque <code>TIMER_ALARM1 == TIMELR</code> . L'alarme se désarmera d'elle-même une fois déclenchée et pourra être désarmée plus tôt à l'aide du registre d'état ARMED.
\$18	ALARM2	Arme l'alarme 2 et configure l'heure à laquelle elle se déclenchera. Une fois armée, l'alarme se déclenche lorsque <code>TIMER_ALARM2 == TIMELR</code> . L'alarme se désarmera d'elle-même une fois déclenchée et pourra être désarmée plus tôt à l'aide du registre d'état ARMED.
\$1C	ALARM3	Arme l'alarme 3 et configure l'heure à laquelle elle se déclenchera. Une fois armée, l'alarme se déclenche lorsque <code>TIMER_ALARM3 == TIMELR</code> . L'alarme se désarmera d'elle-même une fois déclenchée et pourra être désarmée plus tôt à l'aide du registre d'état ARMED.
\$20	ARMED	Indique l'état armé/désarmé de chaque alarme. Une écriture dans le registre <code>ALARMx</code> correspondant arme l'alarme. Les alarmes se désactivent automatiquement lors du tir, mais si vous en écrivez une ici, elles se désarmeront immédiatement sans attendre le déclenchement.
\$24	TIMERAWH	Lecture brute à partir des bits 63:32 du temps (pas d'effets secondaires)
\$28	TIMERAWL	Lecture brute à partir des bits 31:0 du temps (pas d'effets secondaires)
\$2C	DBGPAUSE	Définissez les bits à un niveau élevé pour activer la pause lorsque les ports de débogage correspondants sont actifs
\$30	PAUSE	Réglez haut pour mettre le minuteur en pause
\$34	INTR	Interruptions brutes
\$38	INTE	Activation d'interruption
\$3C	INTF	Force l'interruption
\$40	INTS	Statut d'interruption après masquage et forçage

# Ressources

## ***MECRISP Forth***

Site en deux langues (français, anglais) avec plein d'exemples

<https://mecrisp.arduino-forth.com/>

## ***Mecrisp download***

Le site de référence pour récupérer la version qui convient à vos cartes électroniques

<https://sourceforge.net/projects/mecrisp/>

## ***Mecrisp Stellaris Unofficial UserDoc***

Documentation en ligne très complète

<https://mecrisp-stellaris-folkdoc.sourceforge.io/index.html>

## Index lexical

activate.....	69	GPIO_OUT.....	73	TIMEHR.....	78
and.....	23	GPIO_OUT_CLR.....	73	TIMEHW.....	78
BASE.....	46	GPIO_OUT_SET.....	73	TIMELR.....	79
binary.....	18	GPIO_OUT_XOR.....	73	TIMELW.....	78
BINARY.....	46	hex.....	18	TIMERAWH.....	79
c!.....	42	HEX.....	46	TIMERAWL.....	79
c@.....	42	HOLD.....	47	u.....	21
compiletoflash.....	28	idle.....	70	variable.....	42
compiletooram.....	28	mémoire.....	42	wake.....	70
constant.....	42	multitask.....	69	.....	39
decimal.....	18	pile de retour.....	41	.....	39
DECIMAL.....	46	S".....	50	.....	50
drop.....	41	save.....	28	.....	38
dump.....	37	see.....	37	@.....	42
dup.....	41	shift.....	22	#.....	47
EMIT.....	49	sin.....	44	#>.....	47
f.....	44	SIO_BASE.....	73	#S.....	47
GPIO_OE.....	73	SPACE.....	51	<#.....	47
GPIO_OE_CLR.....	73	task:.....	69		
GPIO_OE_SET.....	73	Tera Term.....	11		