Le grand livre

de MECRISP Forth

version 1.1 - 20 décembre 2023

Auteur

- Marc PETREMANN

Table des matières

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

Installer MECRISP sur la carte RP pico

Préparation de la carte RP PICO

Déballez 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

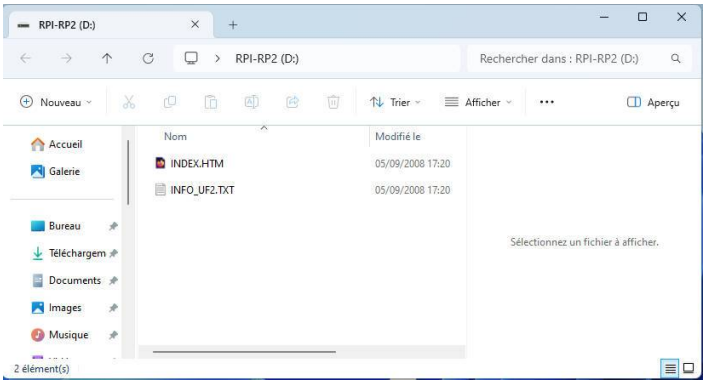


Figure 2: espace de stockage dans la carte RP pico

une fenêtre dans l'explorateur de fichiers:

Téléchargement et installation de MECRISP sur RP PICO

Vous pouvez téléchager 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**.

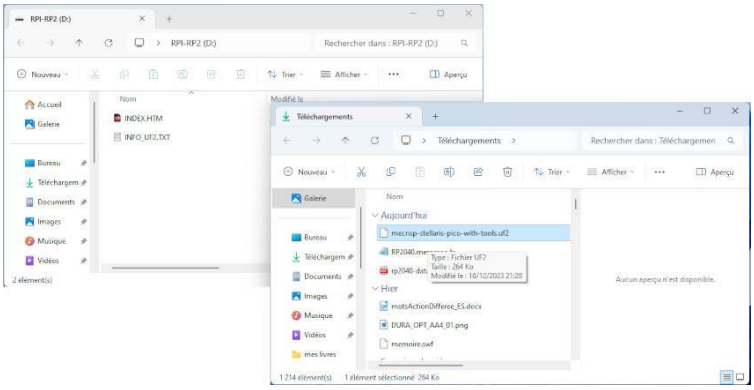


Figure 3: copie du fichier d'extension UF2
vers l'space de la carte RP pico

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 MECRSIP 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

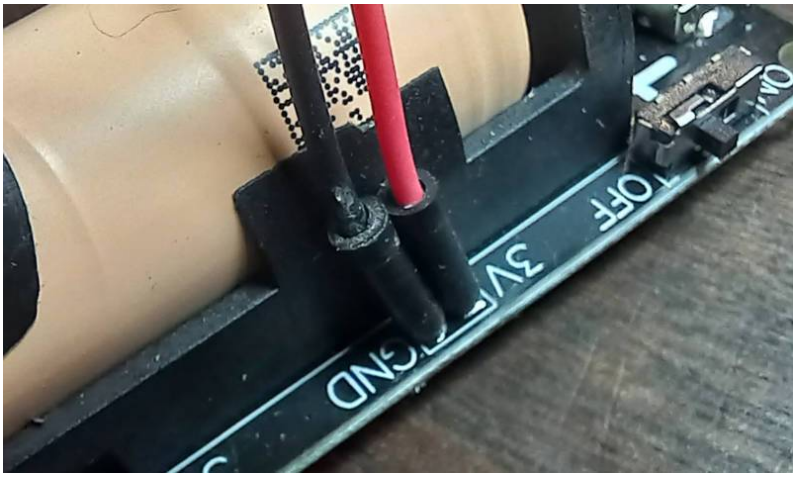


Figure 5: soudage de fils à une sortie 3V3

- 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:

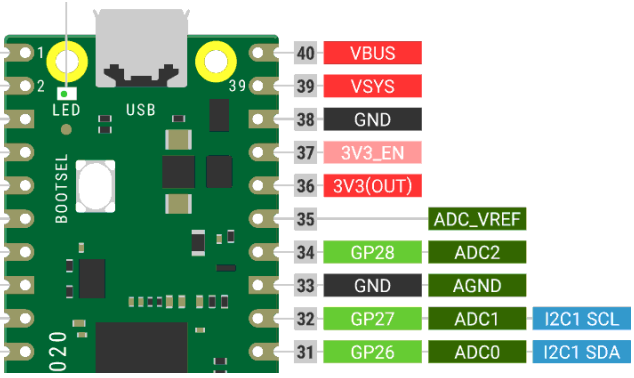


Figure 6: pins 38 et 39 vers alimentation 3V3

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.

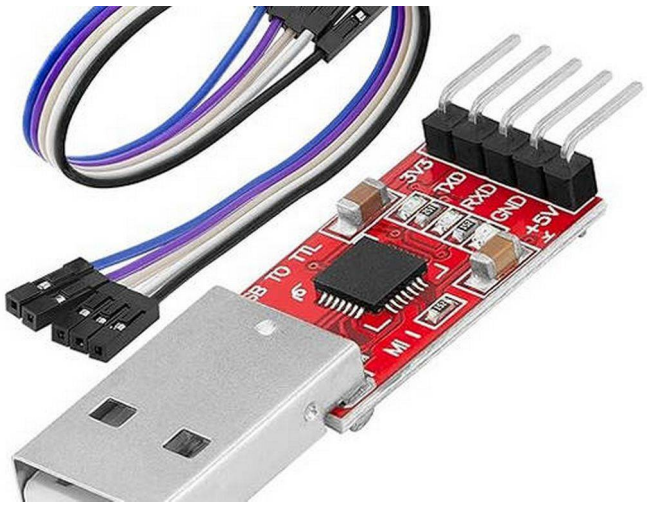


Figure 7: adaptateur TTL <-> USB

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)

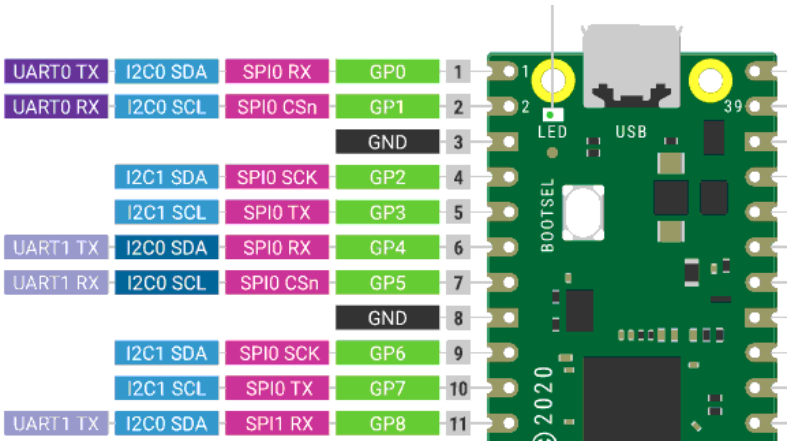


Figure 8: pins 1 à 3 vers adaptateur USB / TTL

- RXD --> pin 1 (UART0 TX)

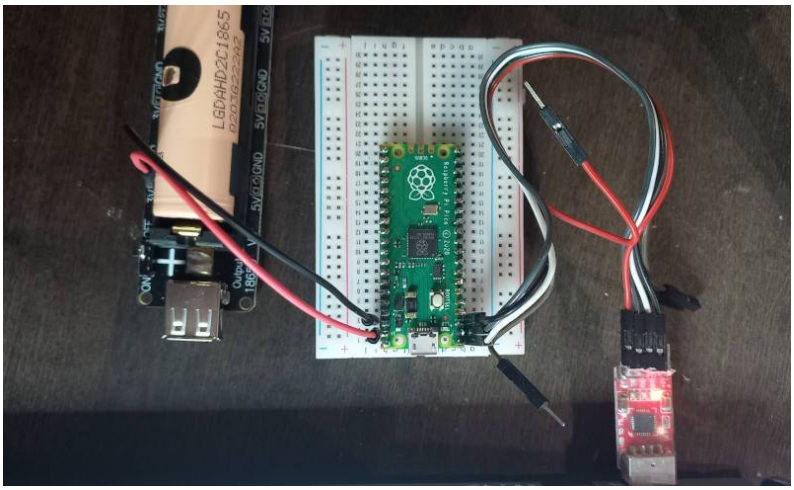


Figure 9: branchement carte RP pico au PC

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

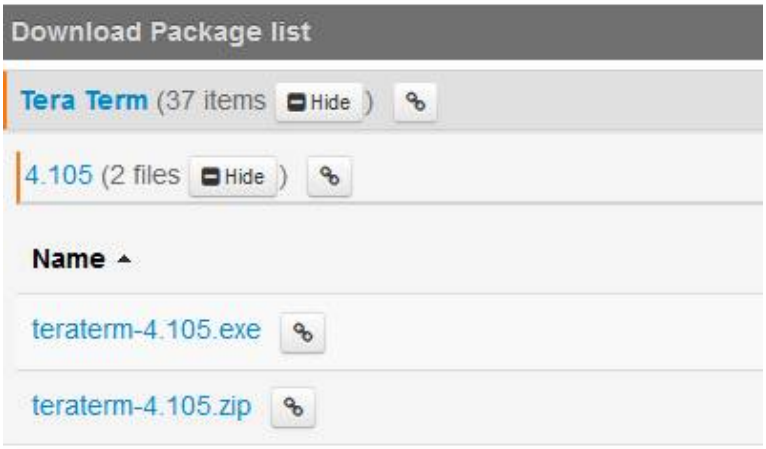


Figure 10: invite de téléchargement de TERA TERM

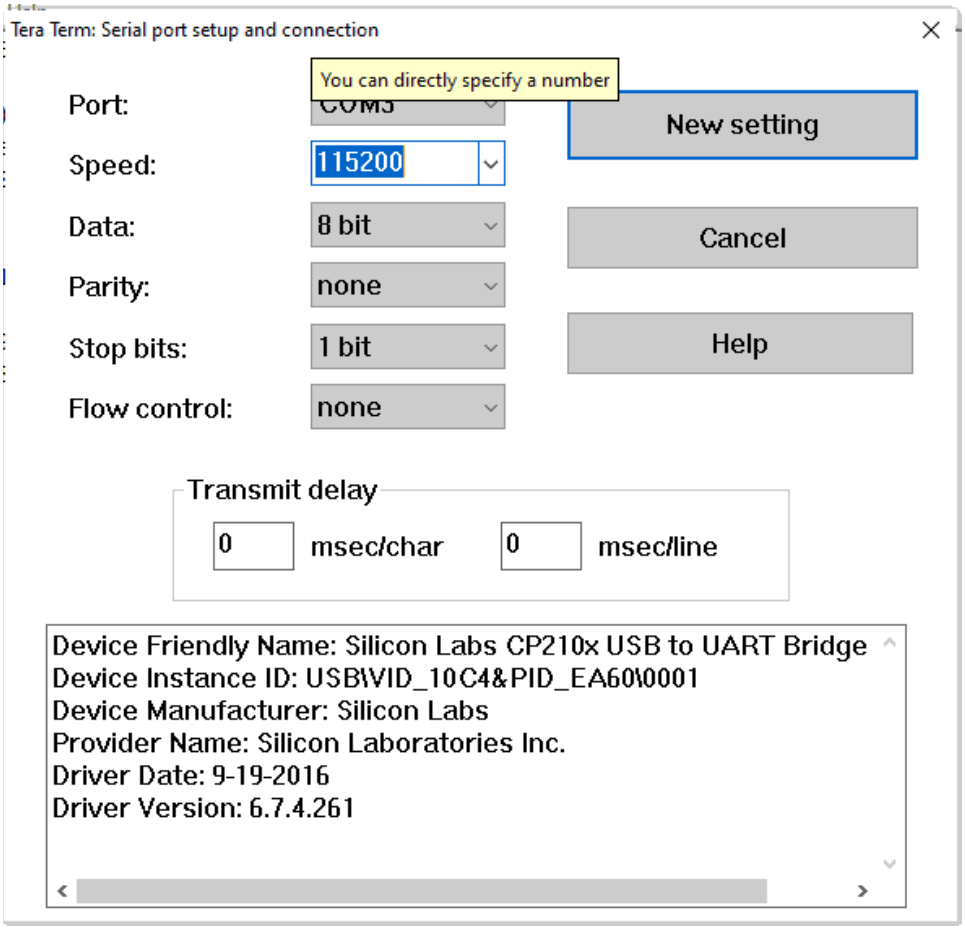
Allez sur la page téléchargement, récupérez le fichier exe ou zip:

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

Paramétrage de Tera Term

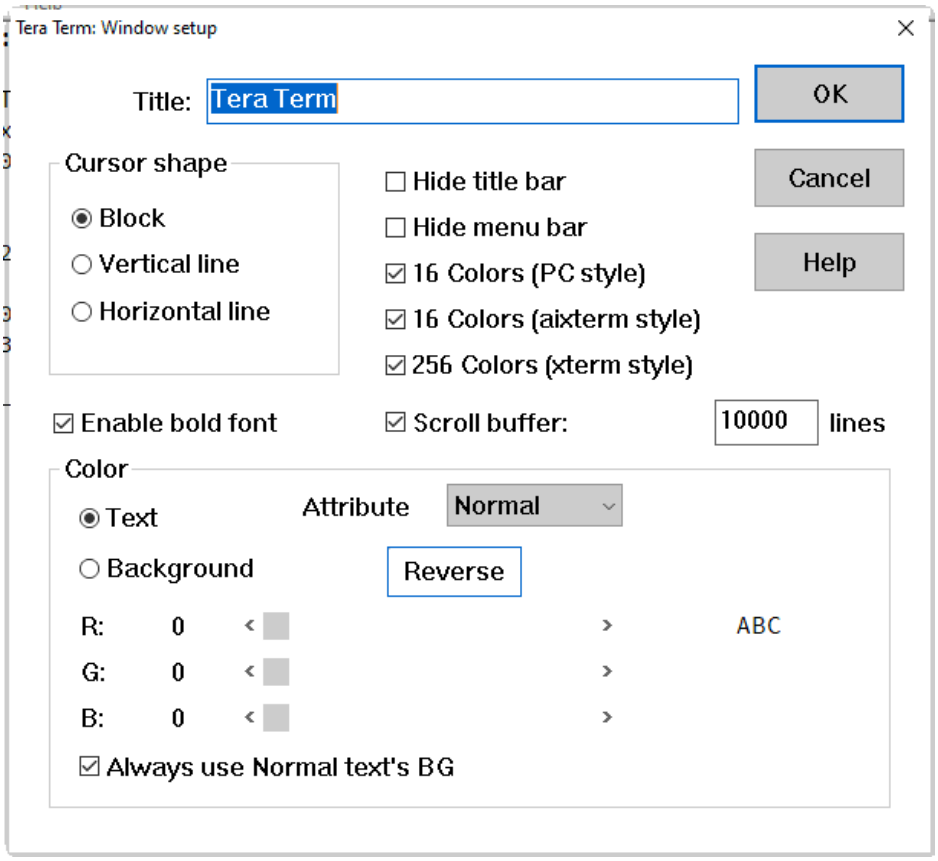
Pour communiquer avec MECRISP Forth, il faut régler certains paramètres:

- cliquez sur Configuration -> port série

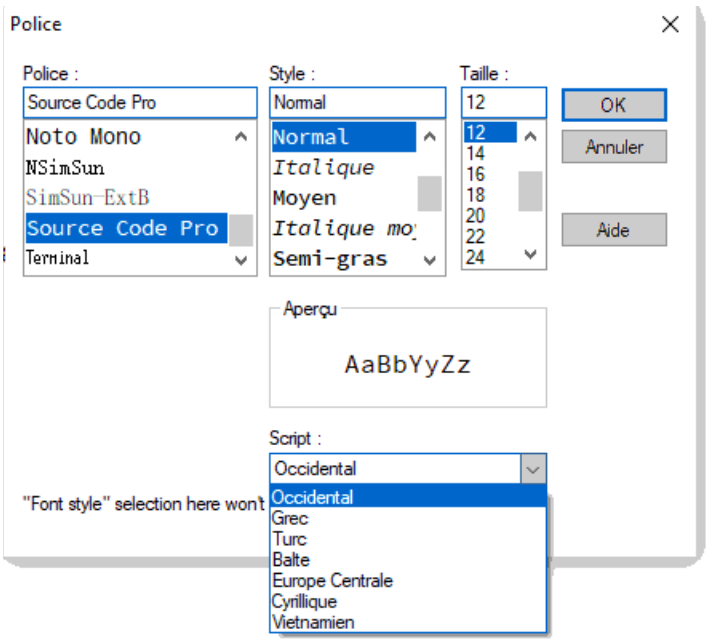


Pour un affichage confortable:

- 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**.

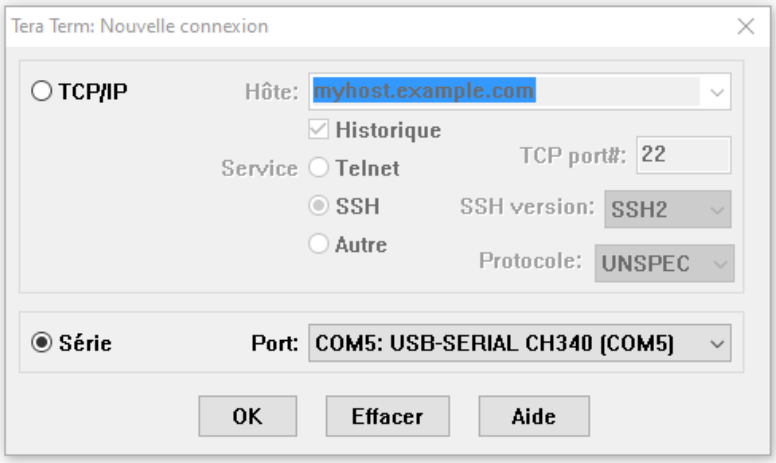
Utilisation de Tera Term

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:



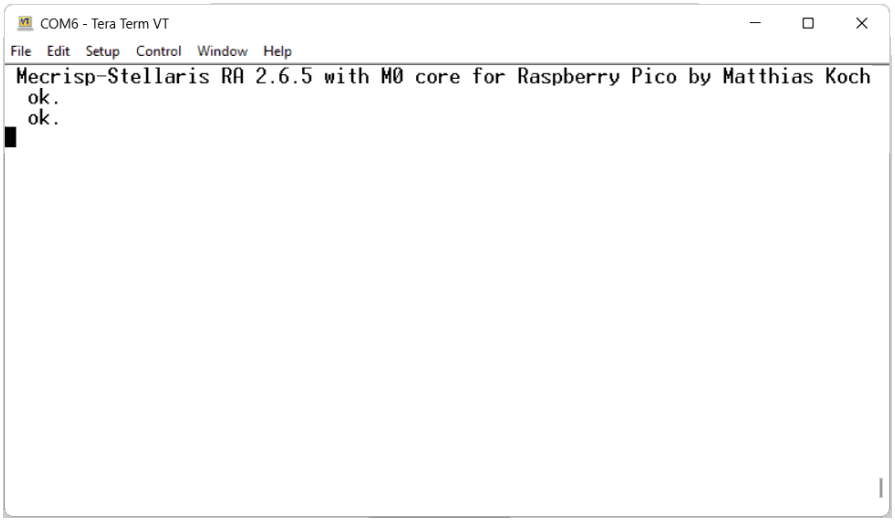


Figure 11: fenêtre du terminal Tera Term communiquant avec MECRISP Forth

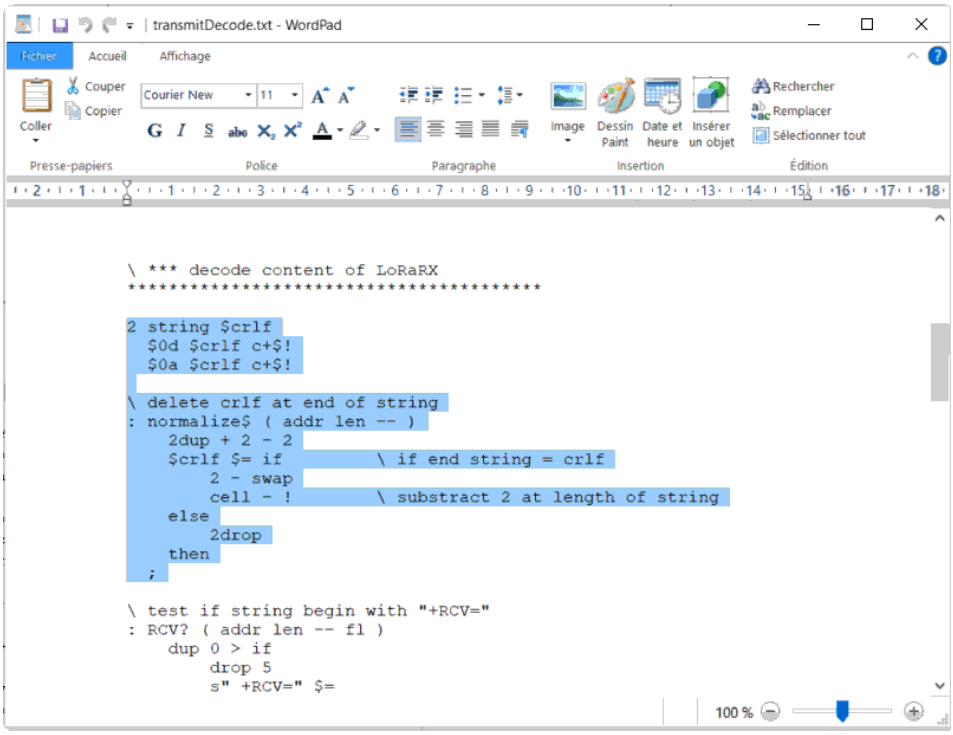
Si tout s'est bien passé, vous devez voir ceci:

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

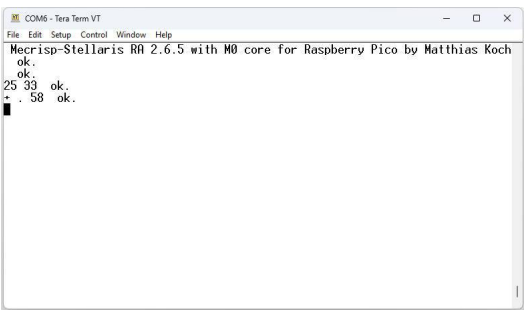


Figure 12: première opération avec MECRISP

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'intéret 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 seuls combinaisons de représentation hexadécimale seront comprises ente `0` et `F`. Ici, les lettres A à F **sont des chiffres** hexadécimaux !

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

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

Taille des nombres sur la pile de données FORTH

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 `$00000000`, la plus grande sera `$FFFFFFFF`. Toute tentative d'empiler une valeur de taille supérieure se solde par un écrétage de cette valeur :

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

Empilons la plus grande valeur possible au format hexadicimal 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 réprésenter un nombre entier, le bit de poids fort est utilisé comme signe :

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

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

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

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

```
%00000000000000000000000000000001
%11111111111111111111111111111111
%                              10
```

Etape suivante :

```
%00000000000000000000000000000001
%11111111111111111111111111111111
%                             10
%                            100
```

Arrivé à la fin, on aura comme résultat :

```
 %00000000000000000000000000000001
 %11111111111111111111111111111111
%100000000000000000000000000000000
```

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

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

Accès mémoire et opérations logiques

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

```
hex
0 variable score
score 10 dump   \ display :
200203A0 :  00 00 00 00 08 78 6B 0C   9D 53 C8 D9 C8 07 50 6E  | .....xk.  .S....Pn |
200203B0 :  68 7B B2 80 09 FB AE BC   9C AC 71 D3 16 9C F7 77  | h{......  ..q....w |
```

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

```
decimal
1900 score !
Score 16 dump   \ partial display :
200203A0 :  6C 07 00 00 08 78 6B 0C   9D 53 C8 D9 C8 07 50 6E
200203B0 :  68 7B B2 80 09 FB AE BC   9C AC 71 D3 16 9C F7 77
```

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

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

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

```
1 ONBOARD_LED lshift GPIO_OE !
```

Ce code active GPIO25 en sortie. Le mot **ONBOARD_LED** est une constante, dont le contenu contient 25. On aurait aussi bien pu écrire ceci :

```
1 25 lshift GPIO_OE !
```

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

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

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

```
1 17 lshift GPIO_OE !
```

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

```
1 25 lshift
1 17 lshift or  GPIO_OE !
```

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

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

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

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

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" ;  fonctionne pas @todo à corriger
: 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 -- )   fonctionne pas @todo à corriger
    create
        c,
    does>
        dup 1+ swap c@ 0 do
            dup i + c@ emit
        loop
        drop space
  ;
2 morse: mA     char . c,   char - c,
4 morse: mB     char - c,   char . c,   char . c,   char . c,
4 morse: mC     char - c,   char . c,   char - c,   char . c,
mA mB mC    \ display   .- -... -.-.
```

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....

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: 3,19999999953433871269226074218750
3,3 f.       \ affiche: 3,29999999813735485076904296875000
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 :

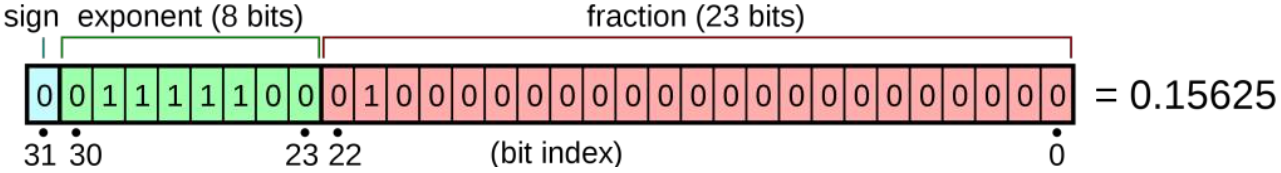


Figure 13: structure d'un nombre réel sur 32 bits

La portée des nombres pouvant être ainsi utilisés est dans l'intervalle [1.18E-38 → 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: 0,09999999860301613807678222656250
```

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° est exact, il n'en est pas de même pour celui de 45° qui devrait être 0.70710678...

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

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

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 ;

key est un mot d'exécution différée ;

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

`registers` marque un vocabulaire.

Ces mots sont affichés par ordre alphabétique.

-inf	-roll	-rot	,	;	:	!	?do
?dup	?of	.	."	.digit	.s	'	(create)
(dp)	(find)	(latest)	(pause)	[	[']	[char]	]
@	*	*/	*/mod	/	/mod	#	#>
#S	+	+!	+inf	+loop	<	<#	<=
<>	<builds	=	>	><,	>=	>in	>r
0-foldable	0<	0<>	0=	0to1-atan	0to1sqrt	1-	1-foldable
1+	1overlnof2	2-	2-foldable	2-rot	2!	2@	2*
2/	2+	2>r	2constant	2drop	2dup	2nip	2over
2r@	2r>	2rdrop	2rot	2swap	2tuck	2variable	3-foldable
4-foldable	5-foldable	6-foldable	7-foldable	abs	accept	acos	activate
adcs	add	addr.	addrinflash?		addrinram?	adds	adds&subs
again	ahead	align	aligned	allot	and	ands	arshift
asin	asrs	asrsr	atan	atan-coef	atan-table	b	background
base	base-ivl-atan		bcc	bcs	begin	beq	bge
bgt	bhi	bhs	bic	bic!	bics	binary	bis!
bit@	bl	ble	blo	bls	blt	blx	bmi
bne	boot-task	bpl	buffer:	bvc	bvs	bx	c,
c!	c"	c@	c+!	call,	case	catch	cbic!
cbis!	cbit@	cell+	cells	cexpect	cflash!	char	
check+label							
cjump,	clz	cmp	compare	compileonly		compiletoflash	
compiletoram		compiletoram?		connect-flash		const.	constant
cos	cos-coef	count	cr	create	ctype	current-source	
cxor!	cycles	d-	d.	d/	d/mod	d+	d<
d<>	d=	d>	d0<	d0=	d2*	d2/	dabs
decimal	deg-90to90	deg0to360	deg2rad	delay-cycles		depth	
destination-r0							
dictionarynext		dictionarystart		digit	dint	disasm	disasm-$
disasm-fetch		disasm-step	disasm-string		dnegate	do	does>
double-operand		drop	dshl	dshr	du<	du>	dump
dump16	dup	eint	eint?	else	emit	emit?	endcase
endof	enter-xip	eors	erase-range	erase#	ersteszeichen		evaluate
even	execute	exit	exit-xip	exp	exp-1to1	exp-coef	f.

f.n	f*	f/	f#	f#S	f>s	false	fill
find	flashvar-here		floor	flush-cache	forgetram	h,	h!
h.s	h@	h+!	half-q1-cos-rad		half-q1-sin-rad		halign
handler	hbic!	hbis!	hbit@	here	hex	hex.	hflash!
hold	hold<	hook-emit	hook-emit?	hook-find	hook-key	hook-key?	hook-pause
hook-quit	hxor!	i	idle	if	image>spi-offset		imm3.
imm3<<1.	imm3<<2.	imm5.	imm5<<1.	imm5<<2.	imm7<<2.	imm8.	imm8<<1.
imm8<<2.	immediate	inline	inline,	insert	interpret	ipsr	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	j	jump-destination		jump,	jumps
k	key	key?	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	ldrsb	ldrsh	leave
list	literal,	ln	ln10overln2	lnof2	load&store	load#	log10
log10of2	log2	log2-1to2	loop	lshift	lsls	lslsr	lsrs
lsrsr	m*	m/mod	max	memstamp	min	mod	mov
mov&add	move	movs	movs&cmp	ms	muls	multitask	mvns
name.	negate	new	next-task	nip	nop	not	number
numbertable	nvariable	of	opcode?	operandenparser		or	orrs
over	parse	pause	pi/2	pi/4	pick	pop	postpone
pow10	pow2	preparetask		previous	program-range		push
push&pop	q1-sin-rad	q1toq4-sin	query	quit	r@	r>	rad2deg
rdepth	rdrop	recurse	reg.	reg16.	reg16split.		register.
registerlist.		registerliteral,		registerparser		registerparser16	
remember+jump		remove	repeat	reset	restart	ret,	rev
rev16	revsh	rol	roll	rom-code	rom-data	ror	rors
rot	rp!	rp@	rpick	rrotate	rshift	s"	s>d
s>f	save	save-task	save#	sbcs	see	seec	serial-emit
serial-emit?		serial-key	serial-key?	setflags	setsource	sev	shifts-imm
shl	shr	sign	signedloads	sin	sin-coef	single-operand	
singletask	skipstring	smudge	source	sp!	sp@	space	spaces
sqrt	stackspace	state	stmia	stmia&ldmia	stop	str	strb
strh	string,	subs	swap	sxtb	sxth	symbolwert	tan
task-in-list?		task-state	task:	tasks	then	throw	tib
TIMEHR	TIMEHW	TIMELR	TIMELW	TIMERAWH	TIMERAWL	token	true
tst	tuck	type	u.	u.2	u.4	u.8	
u.ns	u.s	u*/	u*/mod	u/mod	u<	u<=	u>
u>=	ud.	ud*	ud/mod	udm*	um*	um/mod	umax
umin	unhandled	unloop	until	unused	up	us	uxtb
uxth	variable	vorneabschneiden		wake	wfi	while	words
xor	xor!	zero-operand					

Index lexical