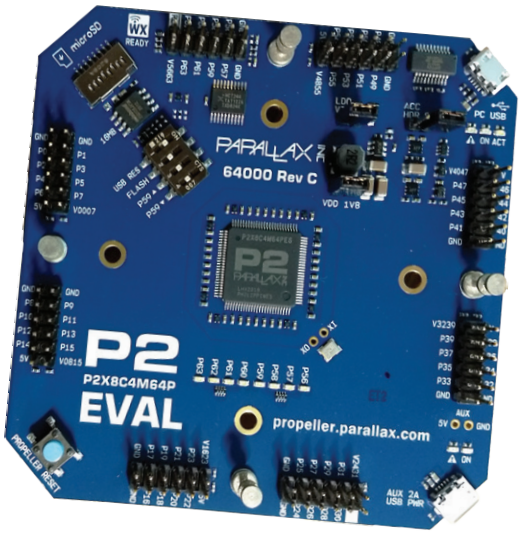
Propeller 2 de Parallax (4)

Envoi de chaînes de caractères



Mathias Claussen (Elektor)

Notre série d'articles sur le Propeller 2 de Parallax se poursuit ! Cette fois, nous nous penchons sur certains problèmes que vous pourriez rencontrer lors de l'envoi de chaînes de caractères avec le langage Spin2…

L'objectif de cette série d'articles est de présenter le microcontrôleur Propeller 2 de Parallax. Dans cet article, nous élargirons les possibilités d'envoyer des chaînes de caractères, comme vous le faites avec la fonction `print` sur vos systèmes Arduino. L'idée est de pouvoir émettre de manière très classique des chaînes de caractères, ce qui peut être utile

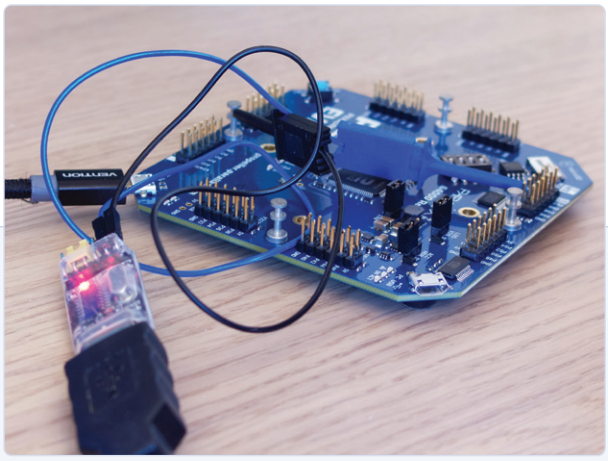


Figure 1. Propeller 2 connecté à un convertisseur USB-série.

Listage 1. Code complet.

```
'We run in the internal RC giving us 25 MHz clockspeed

PUB main()
   pinwrite(56, 1 )
   serial_start()
  repeat
     waitms( 250 )
     pinwrite(56, 1 )
     tx("0")
    waitms( 250 )
     pinwrite(56, 0 )
     tx("1")

 PUB serial_start()
     WRPIN( 57, %01_11110_0  )             'set async tx mode for txpin
     WXPIN( 57, ((217<<16) + (8-1 )) )     'set baud rate to sysclock/115200 and word size to 8 bits
   org
     dirh    #57
   end

PUB tx(val)
    WYPIN(57,val) 'load output word
   org
    WAITX   #1             'wait 2+1 clocks before polling busy
     wait
     RDPIN   val,#57  WC    'get busy flag into C
     IF_C   JMP   #wait     'loop until C = 0
  end
```

Listage 2. Envoi d'un texte, caractère par caractère.

```
'We run in the internal RC giving us 25MHz clockspeed
PUB main()
   pinwrite(56, 1 )
   serial_start()
  repeat
     waitms( 250 )
     pinwrite(56, 1 )
     tx("C")
     tx("o")
     tx("d")
     tx("e")
     tx(" ")
     tx("a")
     tx("l")
     tx("i")
     tx("v")
     tx("e")
    waitms( 250 )
     pinwrite(56, 0 )
     tx("1")

 PUB serial_start()
   WRPIN( 57, %01_11110_0  )               'set async tx mode for txpin
   WXPIN( 57, ((217<<16) + (8-1 )) )       'set baud rate to sysclock/115200 and word size to 8 bits

  org
     dirh    #57
   end

PUB tx(val)
       WYPIN(57,val) 'load output word
   org
     WAITX   #1             'wait 2+1 clocks before polling busy
      wait
     RDPIN   val,#57  WC    'get busy flag into C
     IF_C   JMP   #wait     'loop until C = 0
  end
```

pour le débogage. Pour récapituler où nous en sommes : notre code est capable d'envoyer un seul caractère avec les réglages suivants : 115 200 bauds, 8 bits de données, pas de parité, un bit d'arrêt. Le code complet apparaît dans le **listage 1** et il est possible de le télécharger à l'adresse suivante [1].

Si nous voulons maintenant envoyer la chaîne « Code alive » (cf. **listage 2**), il faut effectuer plusieurs appels à la fonction `tx()`, un pour chaque caractère. Si nous voyons des lignes de code qui se répètent, il va être intéressant de réfléchir à généraliser la tâche. Copier des parties de fragments de code partout dans le programme n'est jamais une bonne idée, surtout si vous devez ajouter quelque chose ou corriger le code ultérieurement. Nous construirons une fonction qui prend une chaîne de caractères comme argument, la découpe en caractères séparés et les envoie un par un à notre fonction `tx()`. Cela semble facile, et si vous avez une certaine expérience du codage, cela devrait théoriquement le rester. Si vous utilisez Spin2 ou Spin pour la première fois, il y aura sans doute un peu plus de lecture et de tâtonnements.

Passer une chaîne de caractères

Comme les chaînes de caractères occupent un espace continu de mémoire contenant tous les caractères consécutifs ainsi qu'une terminaison formée d'un zéro binaire (« \0 »), notre action consistera simplement à transmettre l'adresse de départ de cette mémoire à une fonction. Si nous utilisons le langage C/C++, l'opérateur `&` fera l'affaire, en conjonction avec le premier élément. Avec Spin2, la syntaxe est un peu différente, mais le fonctionnement est pratiquement le même. Ce qui diffère, c'est la façon dont nous déclarons la fonction et l'argument lui-même. Le type de variable passé à notre fonction n'est pas spécifié. Si nous n'indiquons aucun type, le langage Spin2 considèrera qu'il s'agit d'un type long (32 bits). Il est parfois pratique d'utiliser ce réglage par défaut avec Spin2 pour les variables passées à une fonction, mais au prix d'un certain nombre d'effets secondaires indésirables. Dans notre cas, il s'agit de l'adresse mémoire de notre chaîne de caractères et cela conviendra parfaitement à notre objectif. Ce qui va être différent, c'est la manière de déclarer les variables locales. Nous voyons clairement que toutes les variables, ici `c`, doivent être déclarées en tête de la fonction.

Nous utilisons donc un caractère `|` après le crochet fermant de cette ligne. S'il nous faut plusieurs variables, nous les placerons sur cette ligne en les séparant par des virgules (`,`). Le **listage 3** montre à quoi ressemblera l'entête de la fonction.

Listage 3. Entête de la fonction.

```
PUB prints( s ) | c
```

À l'intérieur de la fonction (voir **listage 4**), nous devons procéder à une lecture attentive des données, octet par octet. L'instruction `c := byte[s++]` nécessite quelques mots d'explication. Notre variable `c` n'a actuellement aucun type, ou plutôt, elle possède un type par défaut. En l'absence de la mention `byte[]`, nous lirions 32 bits (variable de type `uint32_t`) en une seule fois ; avec `byte[s++]`, nous lisons un octet à l'emplacement de 's', à l'adresse mémoire où commence la chaîne. Le `s++` ici, combiné avec `byte[]`, garantit que l'adresse ne sera incrémentée que d'un octet. Si nous ne le faisons pas, toutes les opérations appliqueront un format de 32 bits. Comme nous savons maintenant que nous lisons octet par octet, examinons l'instruction `repeat while`. Tant que `c` n'est pas égal (`<>`) au zéro binaire, tout ce qui se trouve à l'intérieur de l'instruction `repeat while` fait partie de la boucle.

Listage 4. Fonction `prints`.

```
PUB prints( s ) | c
   c := byte[s++]
   REPEAT WHILE (c <> 0)
     tx(c)
     c := byte[s++]
```

À l'intérieur de la boucle, illustrée par le **listage 5**, la fonction `tx()` émet un octet. Une autre ligne récupère l'octet suivant et le place dans notre variable `c`. Ensuite la boucle continue et commence à vérifier si `c` est encore différent du zéro binaire.

Listage 5. Fonction `tx()` appelée octet par octet.

```
  REPEAT WHILE (c <> 0)
     tx(c)
     c := byte[s++]
```

Notre fonction `prints()` est donc maintenant en place et nous sommes capables d'envoyer des chaînes de caractères, par le biais d'un UART, à un PC ou un analyseur logique. La suite consiste à lire l'état des broches d'E/S et à transmettre l'information à l'aide de l'UART. Est-il si difficile de lire l'état d'une broche d'E/S ? Eh bien, en théorie, c'est très facile. Parfois, ce sont les petits extras qui font la différence et qui montrent l'étendue des fonctions qu'offrent les broches intelligentes.

(200479-D-04)

Des questions, des commentaires ?
Envoyez un courriel à l'auteur (mathias.claussen@elektor.com) ou contactez Elektor (redaction@elektor.fr).

Contributeurs
Conception et texte : **Mathias Claußen**
Rédaction : **Jens Nickel, C. J. Abate**
Mise en page : **Giel Dols**
Traduction : **Pascal Godart**

PRODUITS

- **Convertisseur USB>TTL CH340**
  www.elektor.fr/ch340-usb-to-ttl-converter-uart-module-ch340g-3-3-v-5-5-v

LIEN

[1] **Page de l'article** : www.elektor.fr/200479-D-04