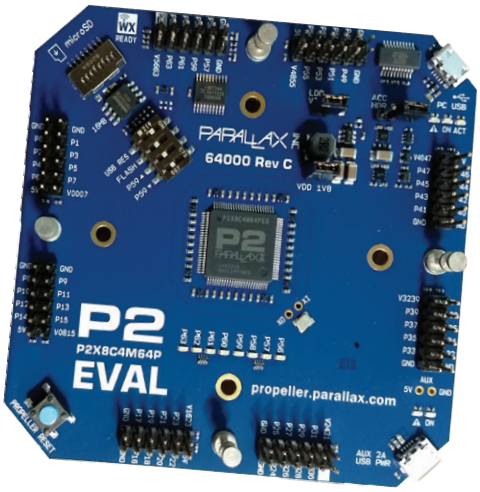
Kennismaking met de Parallax Propeller 2 (deel 4)

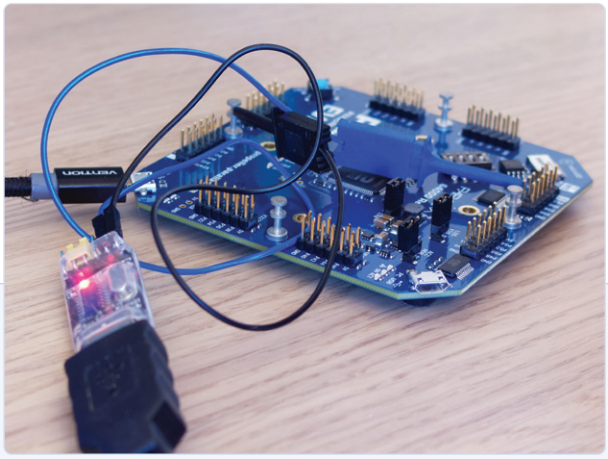
tekens versturen



Mathias Claußen (Elektor)

Deze keer in onze serie over de Parallax Propeller 2 gaan we in op het versturen van strings met de SPIN2 taal.

Het doel van deze serie artikelen is om je kennis te laten maken met de Parallax Propeller 2. In dit artikel zullen we onze mogelijkheden uitbreiden om tekens te versturen, zoals je dat op je Arduino systemen doet met `print`. In deze eerste opdracht versturen we op de klassieke manier wat debug-informatie. Om samen te vatten waar we zijn: onze code is in staat om een enkel karakter te verzenden met 115200 baud, 8 databits, geen pariteit en een stop bit. De volledige code staat in **listing 1** en kan worden gedownload



Figuur 1. Propeller 2 connected to a USB serial converter.

Listing 1. Complete code.

```
'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
```

Listing 2. Text char by char.

```
'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
```

op [1]. Als we nu "Code alive" willen verzenden, zoals getoond in **listing 2**, zal dit resulteren in een heleboel aanroepen naar onze tx()-functie, voor elk karakter één. Zodra we herhalende coderegels zien, denken we natuurlijk direct aan het veralgemenen van de taak. Gekopieerde delen van codefragmenten in een project kun je beter vermijden, vooral als je later iets moet toevoegen of de code moet repareren. We zullen een functie bouwen die een string als argument neemt, deze in losse karakters knipt en deze één voor één naar onze tx()-functie stuurt. Dit klinkt eenvoudig, en als je genoeg gecodeerd hebt, zou het dat in theorie ook moeten zijn. Maar als je SPIN2 -of SPIN in het algemeen- voor het eerst gebruikt , betekent dit wat meer lees- en knutselwerk.

Tekenreeksen versturen

Omdat strings eigenlijk bestaan uit een reeks opeenvolgende geheugenplaatsen die alle karakters na elkaar bevat, afgesloten door een NULL-karakter("\0"), zal geprobeerd worden om het startadres van dit geheugen door te geven aan een functie. Als we C/C++ gebruiken, doen we dat met de &-operator in combinatie met het eerste element. In SPIN2 wordt dit qua syntaxis een beetje anders aangepakt, maar het werkt grotendeels hetzelfde. Wat anders is, is de manier waarop we de functie en het argument zelf declareren. Het type variabele dat aan onze functie wordt doorgegeven is niet gespecificeerd. Als we geen type specificeren, zal SPIN2 aannemen dat het een LONG type is (32 bits). In sommige gevallen kan het handig zijn om deze automatische aanname in SPIN2 te gebruiken voor variabelen die aan een functie worden doorgegeven, maar het kan ook enkele ongewenste neveneffecten hebben. In ons geval is het het geheugenadres voor onze string en zal het prima werken. Wat anders is, is de manier waarop we lokale variabelen declareren. Het is duidelijk te zien dat alle variabelen, hier c, gedeclareerd moeten worden in de kop van de functie. Dit wordt

gedaan met een '|' na het haakje aan het einde van die regel. Als we meer dan één variabele nodig hebben, zetten we ze achter elkaar in dezelfde regel en scheiden ze met een komma. **Listing 3** toont hoe de functiekop eruit zal zien.
Binnen de functie (zie **listing 4**) moeten we de gegevens zorgvuldig lezen, byte voor byte. We hebben de `c := byte[s++]` opdracht die wat uitleg behoeft. Onze variabele c heeft momenteel geen type, of beter, een standaardtype. Zonder de `byte[]` te gebruiken zouden we 32 bits (`uint32_t` variable) in één keer lezen; door nu `byte[s++]` te gebruiken, lezen we één byte op de locatie van 's', op het geheugenadres waar de string begint. De combinatie van `s++` met de `byte[]` zal ervoor zorgen dat het adres slechts met één byte wordt verhoogd. Als we dit niet doen, zullen alle operaties 32 bits breed zijn. Omdat er nu voor gezorgd is dat we byte voor byte lezen, gaan we kijken naar het repeat-while statement. Zolang c ongelijk (<>) is aan binair nul zal alles binnen de repeat-while loop worden herhaald.
Binnen de lus, getoond in **listing 5**, hebben we onze tx()-functie die één byte zal versturen waarna in de volgende regel het volgende byte wordt opgehaald en opgeslagen in onze variabele c. Daarna gaat de lus verder en begint te controleren of c nog steeds ongelijk binair nul is.

Dit betekent dat we nu onze `prints()` in orde hebben en in staat zijn om strings via UART naar een aangesloten PC of logic analyzer te sturen. Het volgende is het lezen van de status van I/O pinnen en deze uitvoeren via de UART. Lees de status van een I/O pin. Hoe moeilijk kan het zijn? Wel, in theorie is het heel gemakkelijk. Soms zijn het de leuke extra's die het verschil maken en laten zien welke mogelijkheden de Smart Pins van de Propeller 2 te bieden hebben.

200479-D-03

Listing 3. Function head.

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

Listing 4. prints function.

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

Listing 5. REPEAT WHILE.

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

Vragen of opmerkingen?
Heb je technische vragen of opmerkingen over dit artikel? E-mail de auteur via mathias.claussen@elektor.com of neem contact op met Elektor via editor@elektor.com.

Een bijdrage van:
Tekst en foto's: **Mathias Claußen**
Redactie: **Jens Nickel, C.J. Abate**
Layout: **Giel Dols**
Vertaling: **Jelle Aarnoudse**

WEBLINK

[1] **Artikel web pagina**: www.elektormagazine.nl/magazine/elektor-169/59326