PIO du RP2040 en pratique

expérimentation des E/S programmables du RP2040

Martin Ossmann (Allemagne)

La RPi Pico est une carte de développement appréciée qui offre une puissance de traitement élevée pour une faible consommation d'énergie. Et elle représente un excellent rapport qualité-prix à seulement 4 € l'unité. Le microcontrôleur RP2040 utilisé sur la carte possède une caractéristique unique (jusqu'à présent) : deux blocs PIO-I/O avec huit machines à états. À l'aide d'exemples concrets, nous examinons de plus près les avantages qu'ils peuvent apporter à une application.

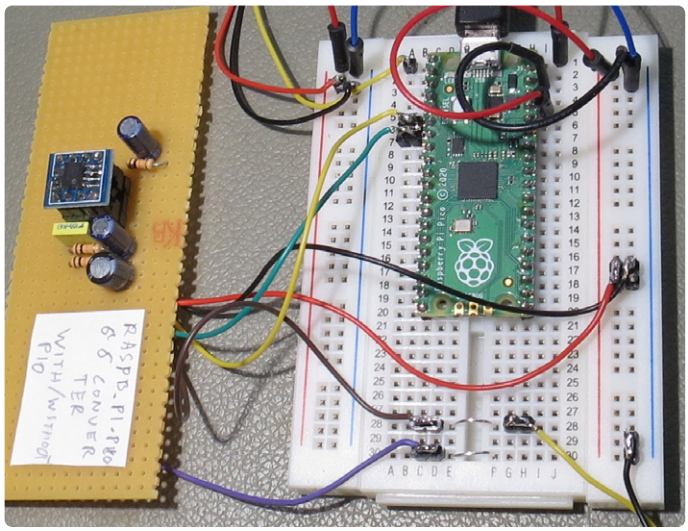


Figure 1. La carte RPi Pico montée sur une platine d'expérimentation et le circuit imprimé Delta_Sigma.

Si vous avez déjà une certaine expérience de la programmation des microcontrôleurs, vous apprécierez sans doute l'utilité d'une unité d'E/S programmable intégrée pour réduire la charge sur les cœurs de traitement principaux. La vraie question est de savoir si cette unité est inutilement compliquée à utiliser et donc n'en vaut pas la peine. Vous aurez certainement une meilleure appréciation de ce qu'elle peut faire une fois que vous aurez travaillé sur les exemples pratiques décrits ici, en utilisant les E/S programmables intégrées au processeur RP2040. La **figure 1** montre ma configuration matérielle expérimentale avec laquelle ces exemples ont été développés.

L'unité PIO

La flexibilité de l'unité d'entrées/sorties programmables vous permet d'utiliser les broches d'E/S pour mettre en œuvre d'autres types de protocoles de communication en plus des standards existants tels que SPI et I²C. Elle se compose de deux blocs PIO, chacun avec quatre processeurs d'E/S. Chaque processeur d'E/S s'intègre à l'architecture de la puce comme le montre la **figure 2**.

Au total, il y a deux PIO dans le RP2040 et chacun contient quatre machines à états (SM) et une mémoire d'instructions de 32 mots pour commander les machines à états. Chaque SM possède deux registres X et Y de 32 bits, un registre à décalage d'entrée de 32 bits (ISR) et un registre à décalage de sortie de 32 bits (OSR). Les noyaux PIO sont chacun connectés à l'unité centrale RP2040 à l'aide de deux FIFO de 32 bits de large et de quatre mots de profondeur. L'unité FIFO TX est

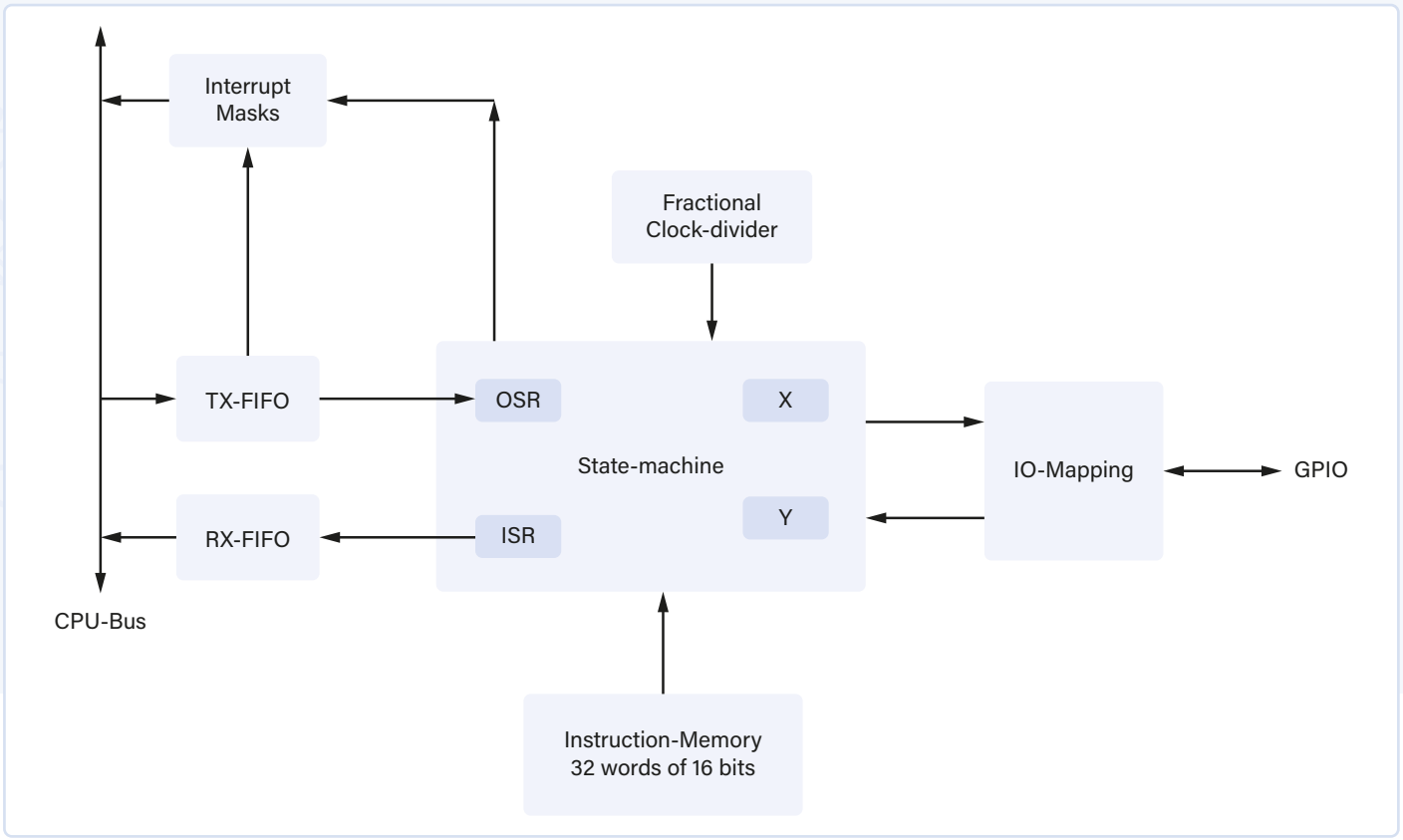


Figure 2. Schéma fonctionnel d'un des deux processeurs d'E/S [4].

utilisée pour l'émission des données et l'unité FIFO RX pour la réception. Si les données ne circulent que dans un sens, les deux FIFO peuvent être configurées pour fonctionner comme une seule FIFO de huit mots de profondeur. Chaque processeur PIO possède un diviseur sur 16 bits entiers plus huit bits fractionnaires pour la génération de l'horloge (voir ci-dessous). Cela permet de régler la fréquence d'horloge de chaque processeur entre 2 et 125 MHz.

Les processeurs PIO ne sont capables de comprendre que neuf instructions. Ces instructions ont une largeur de 16 bits et sont stockées dans une mémoire d'instructions de 32 × 16 bits qui commande les

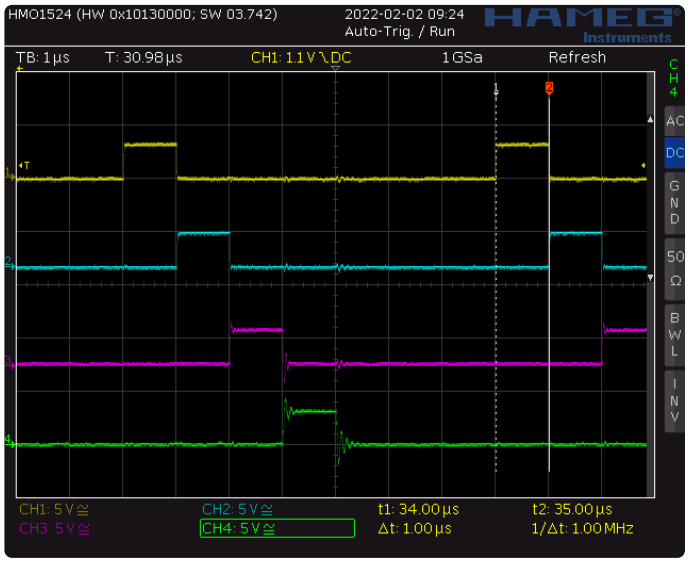


Figure 3. Formes d'onde du premier exemple.

quatre SM de chaque bloc PIO. Les programmes individuels des PIO sont suffisamment courts pour tenir dans cette petite mémoire. Les instructions sont stockées ici par l'unité centrale RP2040. Ces petits programmes sont écrits grâce à un programme assembleur PIO dédié mais assez basique. Chaque instruction est exécutée en un cycle d'horloge.

Un seul processeur PIO ne peut adresser que quelques broches GPIO, et il utilise pour cela des adresses courtes. Le mappage des E/S détermine quelles broches sont alors spécifiquement adressées. Il existe plusieurs constantes programmables qui déterminent pour la catégorie concernée (*in, out, sideset* et *set*) quelle est la première broche adressée. Cela permet de définir les broches d'E/S d'un processeur PIO de manière très souple. Plusieurs processeurs peuvent aussi accéder à la même broche.

Instruments de travail

L'environnement de développement utilisé ici est VS Code. L'installation et l'utilisation de cet outil sont bien décrites dans [1]. Vous y trouverez également des informations sur l'utilisation du générateur de projet utilisé pour créer un projet RPi avec tous les fichiers nécessaires (code source, fichiers de configuration, *Make-Setup*, etc.)

Vous devez également spécifier la sortie via l'USB ou l'UART de la fonction `printf()`. Ici, vous configurez également les bibliothèques que vous devez utiliser. Dans cet article, par exemple, nous utilisons l'outil PIO. Vous pouvez également trouver de l'aide pour l'installation de VS Code dans [2]. Il y a aussi une description détaillée des outils supplémentaires (*Make, Git*, etc.) qui doivent être installés.

Tous les exemples de logiciels peuvent être téléchargés à partir de [3].

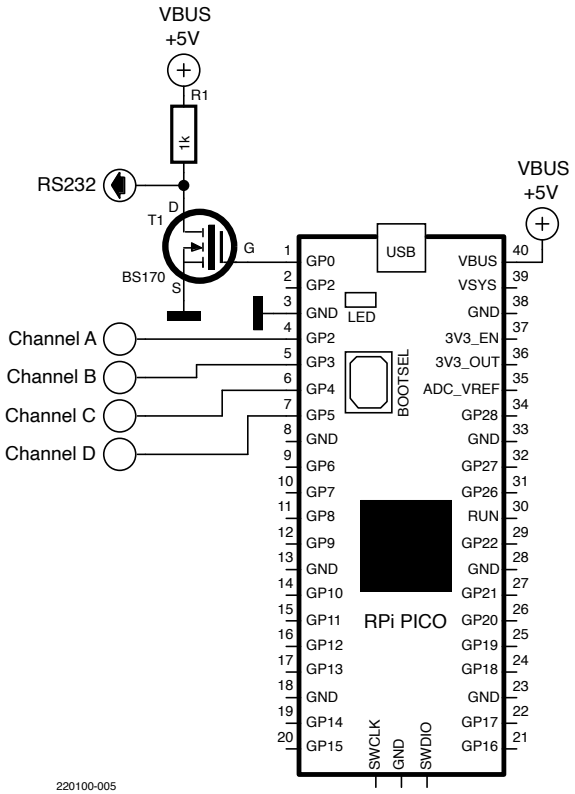


Figure 4. Connexion d'un oscilloscope à quatre canaux au RPi Pico pour visualiser les formes d'onde de sortie du premier exemple.

Commençons

Nous voulons que ce premier programme émette une impulsion de 1 µs sur quatre broches de sortie, l'une après l'autre, comme le montre la **figure 3**. Un oscilloscope à quatre canaux est connecté aux quatre broches de sortie pour visualiser les signaux de sortie résultants. La **figure 4** montre la connexion de l'oscilloscope aux broches d'E/S du RPi Pico.

Le listage suivant contient le premier exemple de programme PIO. Après trois lignes de directives assembleur, la première instruction exécutable se trouve à la ligne 4. Cette ligne et les suivantes génèrent ensuite les quatre impulsions.

Listage 1

```
001 .program pioTest
002 .side_set 2 opt
003 .wrap_target
004   nop           side 0b01 // GP2=1, GP3=0
005   nop           side 0b10 // GP2=0, GP3=1
006   set pins,0b01 side 0b00 // GP2=0, GP3=0,
        GP4=1, GP5=0
007   set pins,0b10       //    GP4=0,      GP5=1
008   set pins,0b00  [2]  //    GP4=0,      GP5=0
009 .wrap
```

Les lignes 4 et 5 contiennent les instructions `nop`, qui, comme vous vous en doutez, ne font rien, mais la ligne 4 est prolongée par le texte `side 0b01`. Cela indique que sur cette ligne, quelque chose d'autre est « fait à côté ». Dans ce cas, la broche *sideset 0* est mise à « 1 » et la broche *sideset 1* est remise à « 0 ». Chaque instruction PIO peut être étendue de cette manière. Cela vous permet d'influer facilement sur beaucoup plus de broches en même temps. Il est important d'identifier quelles sont les broches *sideset* en utilisant des commandes de configuration écrites en C. La définition est faite en C dans le **listage 2** en utilisant la fonction `sm_config_set_sideset_pins()` à la ligne 5. Cela indique que les broches *sideset* commencent à GP2.

Listage 2

```
001 #define set_base 4   // pins start at GP4
002 #define set_count 2  // number of pins is 2
003 sm_config_set_set_pins (&c,set_base,set_count);
004
005 #define side_set_base 2 // side-pins start at GP2
006 sm_config_set_sideset_pins(&c, side_set_base);
007
008 float div = 125.0;
009 sm_config_set_clkdiv(&c, div);
```

L'instruction `nop` de la ligne 4 du **listage 1** génère un front montant via `side` sur la broche *side 1* (= GP3). La ligne 6 contient une instruction `set` qui active les broches 0 et 1 des broches *set*. L'impulsion atteint alors la broche GP4. Les broches *sideset 0* (GP2) et 1 (GP3) sont réinitialisées via la commande `side`. Dans la routine de configuration du **listage 2**, ligne 3, on déclare que les broches set commencent à partir de la broche GP4. L'instruction set peut être utilisée pour charger des registres ou des broches d'E/S avec des valeurs constantes.

Le modificateur `[2]` après l'instruction `set` de la ligne 8 du **listage 1** fait que l'instruction est suivie d'un délai de deux périodes d'horloge. Avec cette spécification utilisant des crochets, chaque instruction peut être retardée de 1 à 31 périodes d'horloge.

Il ne vous reste plus qu'à utiliser une horloge appropriée pour que chaque impulsion soit exactement de 1 µs. Pour ce faire, vous devez diviser l'horloge système de 125 MHz par 125. Avec la fréquence d'horloge résultante de 1 MHz, chaque cycle dure 1 µs comme désiré. Ceci est défini dans la routine de configuration du **listage 2** aux lignes 8 et 9.

Les directives assembleur `.wrap_target` et `wrap` dans le **listage 1** font que les instructions entre elles sont répétées sans fin dans une boucle au moment de l'exécution. Le saut vers le début de la boucle (`.wrap_target`) ne prend pas de temps. Vous pouvez programmer des boucles « sans cycle ajouté ». Dans notre programme, il faut sept cycles d'horloge (cinq instructions + deux cycles de retard) pour parcourir une fois la boucle.

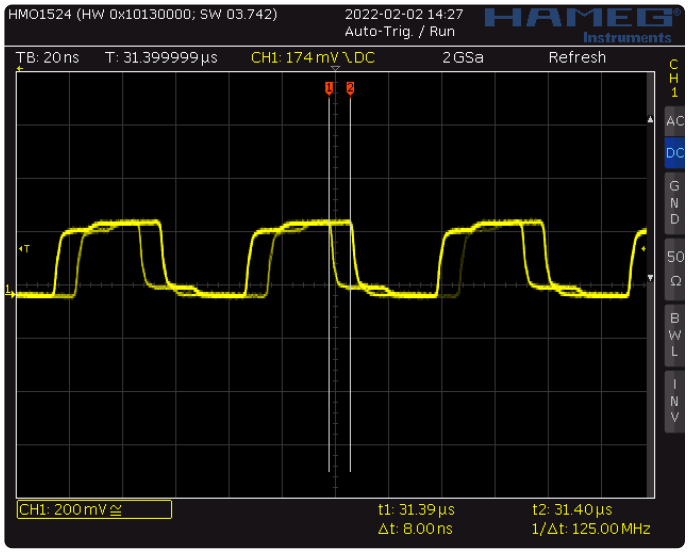


Figure 5. Le signal d'horloge généré présente une gigue.

Génération d'horloge : un diviseur fractionnaire

Comme déjà mentionné, l'horloge du processeur PIO est dérivée de l'horloge système de 125 MHz à l'aide d'un « diviseur fractionnaire ». Ce diviseur de 16 bits peut compter jusqu'à 216 = 65 536 et dispose de huit bits après la virgule. La précision est donc de 1/256. Le « diviseur fractionnaire » est mis en œuvre de la même manière que dans les générateurs de signaux DDS. La fraction derrière la virgule est incrémentée en continu et un front d'horloge est déclenché à chaque report. Le listage suivant démontre le réglage d'une fréquence d'horloge fractionnelle.

Listage 3

```
001 float clockFrequency=28e6;
002 float div=125e6/clockFrequency;
003 sm_config_set_clkdiv(&c,div);
```

L'horloge est élaborée de telle façon que sa période fluctue autour d'une période d'horloge système ce qui produit une gigue dans la forme d'onde. Dans l'exemple, nous voulons une fréquence d'horloge de 28 MHz, mais comme 28 n'est pas divisible par 125, le facteur de division de l'horloge n'est pas un nombre entier mais comporte un certain résidu. Le facteur de division de l'horloge est 125/28 = 4,4642857…

Le **listage 3** montre comment on configure ce diviseur. À titre de démonstration, le **listage 4** ci-dessous montre comment produire un simple signal carré avec le programme PIO.

Listage 4

```
001 .program theProgram
002 .side_set 1 opt
003     nop    side 0
004     nop    side 1
```

L'oscilloscope montre très clairement la gigue de l'horloge dans la forme d'onde de la **figure 5**. Avec des valeurs de diviseur d'horloge plus élevées, la gigue devient proportionnellement plus petite et beaucoup moins importante, de sorte qu'elle peut souvent être négligée. Avec le « diviseur fractionnaire », vous pouvez alors réaliser des vitesses de transmission telles que la typique 115 200 bits/s avec une précision suffisante.

Transmission de données série : un émetteur UART

L'exemple suivant est un peu plus pratique : il s'agit d'envoyer des données en série en utilisant une interface RS232. L'interface est connectée à la broche GP4 afin de laisser libre le GP0 de l'UART intégré à des fins de débogage. Comme l'interface est inversée, on doit générer un signal inversé (état de repos = haut). Lorsque le caractère ASCII est envoyé, la forme de l'impulsion sur la broche GP4 ressemble alors à celle de la **figure 6**.

En plus du signal TXD, l'UART doit émettre un 1 sur une ligne *Busy* tout en envoyant les huit bits de données. GP5 (broche *sideset 0*) peut servir de ligne *Busy*. Le programme PIO associé figure dans le **listage 5** ci-dessous.

Listage 5

```
001 .program theProgram
002 .side_set 1 opt
003     pull              [1]
004     set pins,0 side 1 [2] // startBit=0
005     set x,7               // loop for 7+1 times
006 bit:
007     out pins,1        [2] // LSB first
008     jmp x-- bit
009     set pins,1 side 0 [2] // stopBit=1
```

L'instruction `pull` de la ligne 3 prend un mot de 32 bits dans la FIFO TX et le stocke dans l'OSR. Si aucun mot n'est présent dans la FIFO TX, l'instruction attend simplement un mot et l'insère directement dans l'OSR. La broche de sortie est mise à 0 pour servir de bit de départ par l'instruction `set` de la ligne 4.

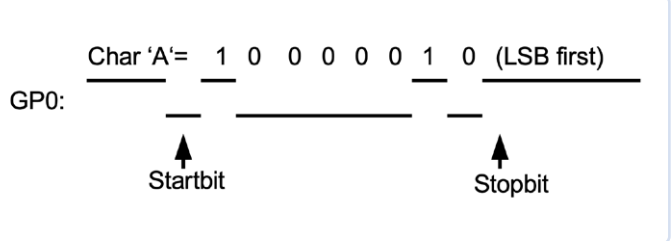


Figure 6. Séquence d'impulsions pour envoyer le caractère « A ».

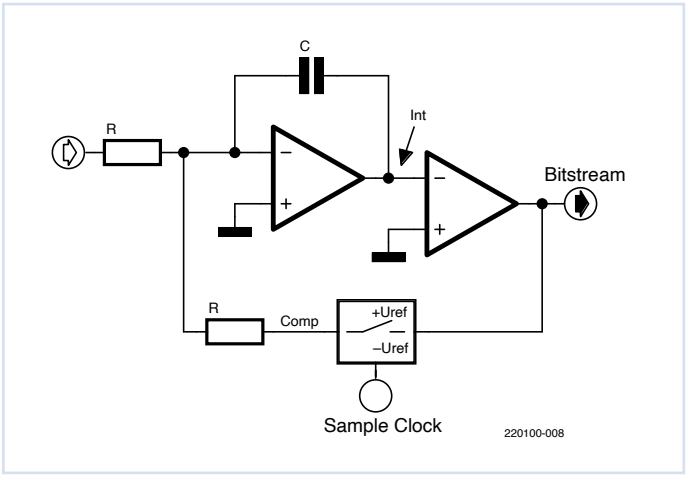


Figure 7. Diagramme fonctionnel d'un convertisseur analogique-numérique Delta-Sigma.

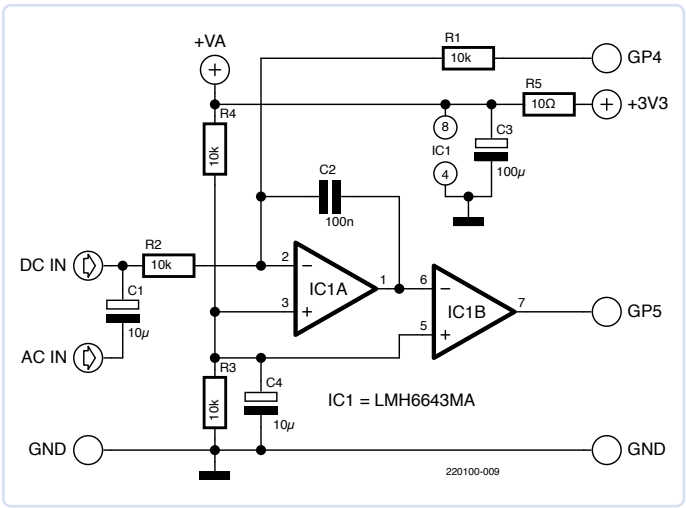


Figure 8. Circuit pratique CA/N Delta-Sigma selon la figure 7.

Les huit bits les moins significatifs du mot dans l'OSR sont maintenant émis dans une boucle. Pour cela, il est nécessaire d'initialiser le compteur de boucle. Le registre X sert ici de compteur de boucle. L'instruction `set` de la ligne 5 l'initialise avec la valeur 7.

Le label `bit` de la ligne 6 marque le début de la boucle. La première et seule instruction de la boucle est l'instruction `out` de la ligne 7. Elle émet le LSB de l'OSR sur la broche out 0 et décale les bits de l'OSR d'une position vers la droite. En plus des broches *set* et *sideset*, les broches out constituent le troisième mode d'utilisation des broches GPIO. La broche GP4 est également définie comme une broche out.

Le saut conditionnel de la ligne 8 avec l'option `x--` décrémente le registre X de 1 et saute ensuite au label `bit` (début de la boucle) lorsque `x` est supérieur à 0. De cette façon, la boucle est parcourue huit fois. La dernière instruction de la routine de transmission TX est l'instruction `set` à la ligne 9, qui crée le bit d'arrêt en mettant la broche *set* 0 à 1.

Avec les deux extensions `side`, le signal *Busy* est généré par la broche *sideset 0* (GP5).

Les extensions de délai entre crochets font que chaque bit a exactement quatre périodes d'horloge. Le débit en bauds est donc égal au débit de l'horloge divisé par quatre. En fixant `clkDiv` à 125 000 000/(4 x 115 200), on obtient bien la vitesse de transmission souhaitée de 115 200 bit/s.

On utilise la fonction `pio_sm_put_blocking(...)` pour transférer des caractères avec un programme C vers le PIO-UART, comme le montre la fonction `PIOprint()` dans le listage suivant. Ici, elle transmet un caractère à la FIFO TX. Si l'unité est pleine, elle se bloque et attend qu'il y ait de la place.

Listage 6

```
001 void PIOprint(const char *s) {
002    while (*s)
003       pio_sm_put_blocking(pio, sm, *s++);
004    }
```

La routine TX-UART montre bien qu'il est possible d'obtenir une fonctionnalité intéressante avec seulement six instructions, les extensions d'instructions `side` et `[delay]` jouant un rôle important.

Un CA/N Delta-Sigma

Dans la section suivante, on va implémenter un convertisseur A/D delta-sigma en utilisant un PIO. On peut voir le principe d'un tel convertisseur à la **figure 7**. On essaie de compenser la tension d'entrée avec une source de tension commutable. Un intégrateur à l'entrée intègre le signal d'erreur suivi d'un comparateur qui détermine le signe de la tension de compensation pour le prochain intervalle de temps. Une séquence de bits est créée à la sortie, qui suit la moyenne de la tension d'entrée. La **figure 8** montre un circuit pratique selon ce principe.

Ce circuit se connecte à GP4 et GP5 du RPi Pico. La séquence de bits est déterminée via PIO et huit bits sont stockés ensemble dans un octet dans la FIFO RX. Le microcontrôleur prend ensuite les octets de cette unité et les traite. On peut voir le programme PIO associé dans le listage suivant.

Listage 7

```
001 .program hello
002 Loop1:
003    set   x,7
004 inLoop:
005    in    pins,1
006    mov   osr,~isr
007    out   pins, 1
008    jmp   x-- go1   [20]
009    push  noblock
010    jmp   Loop1     [23]
011 go1:
012    jmp   inLoop    [26]
```

La boucle extérieure portant l'étiquette `Loop1` se répète sans fin. La boucle intérieure effectue toujours huit itérations. Chaque itération commence par l'instruction qui suit l'étiquette `inLoop`. L'instruction `in` de la ligne 5 prend un bit de la broche *in* et l'écrit dans l'ISR. A la ligne 6, l'ISR est copié inversé dans l'OSR. À la ligne 7, le bit le moins significatif est émis sur la broche out, ce qui génère l'étape de compensation suivante dans l'intégrateur.

Le bit lu est le bit de sortie suivant du convertisseur delta-sigma. Huit bits sont collectés dans l'ISR. L'instruction `jmp` de la ligne 8 commande la boucle interne. Lorsqu'elle a terminé 8 boucles, l'instruction `push` de la ligne 9 est exécutée. Cette instruction stocke l'ISR dans la FIFO RX que le MCU peut ensuite décharger. Les extensions de délai entre crochets garantissent que chaque étape delta-sigma dure 50 impulsions d'horloge. Le diviseur d'horloge a été réglé sur 25. La fréquence d'échantillonnage du convertisseur delta-sigma est donc de 125 MHz / (25 × 50) = 100 kHz.

Pour la démonstration, une onde sinusoïdale de 50 Hz a été convertie par le convertisseur delta-sigma. La **figure 9** montre le spectre résultant. La distance entre le signal utile et le bruit est d'environ 60 dB. Ce n'est pas mal du tout pour un circuit aussi simple !

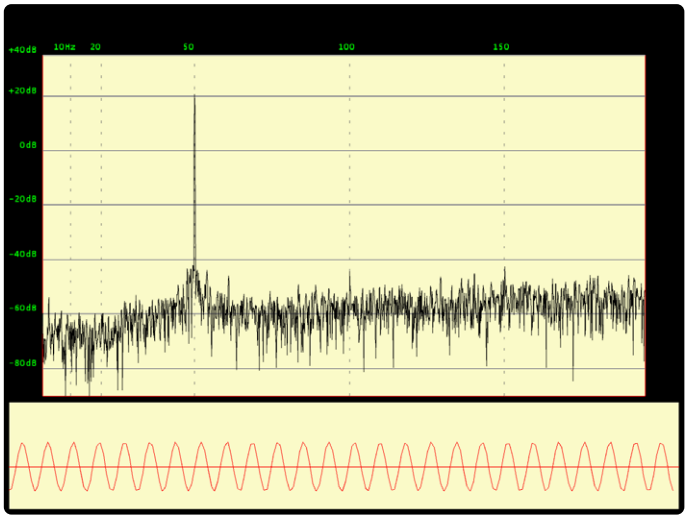


Figure 9. Contenu spectral d'une onde sinusoïdale de 50 Hz après conversion Delta-Sigma.

Construire un générateur de signal

Dans cet exemple, nous construisons un CD/A-R2R simple en utilisant des résistances arrangées en échelle, conformément à la **figure 10**. Nous le connectons aux broches d'E/S du RPi Pico pour produire des signaux de sortie analogiques. Le générateur devra produire un signal de sortie périodique continu. Pour ce faire, les valeurs de données doivent être transférées de manière répétitive d'une table en mémoire vers le convertisseur N/A. Cela fonctionne particulièrement bien en utilisant le DMA (**D**irect **M**emory **A**ccess). Le contrôleur DMA du RPi Pico transfère les données à la FIFO TX. Le **listage 8** ci-dessous montre le programme PIO, composé d'une seule instruction exécutable.

Listage 8

```
001 .program pio_serialiser
002
003 .wrap_target
004     out pins,8 // use autopull
005 .wrap
```

L'instruction `out` amène les huit bits les moins significatifs de l'OSR aux huit broches de sortie GP0 à GP7. L'OSR est alors automatiquement rechargé à partir de la FIFO TX car la fonction *auto-pull* PIO a été activée dans la routine de configuration. Les huit bits suivants sont alors émis avec l'instruction suivante. La boucle de sortie dure donc exactement une impulsion d'horloge PIO. Les données doivent alors passer de la mémoire principale à la FIFO TX le plus rapidement possible. Pour ce faire, deux contrôleurs DMA du RP2040 sont connectés en série en utilisant un DMA de données et un DMA de commande.

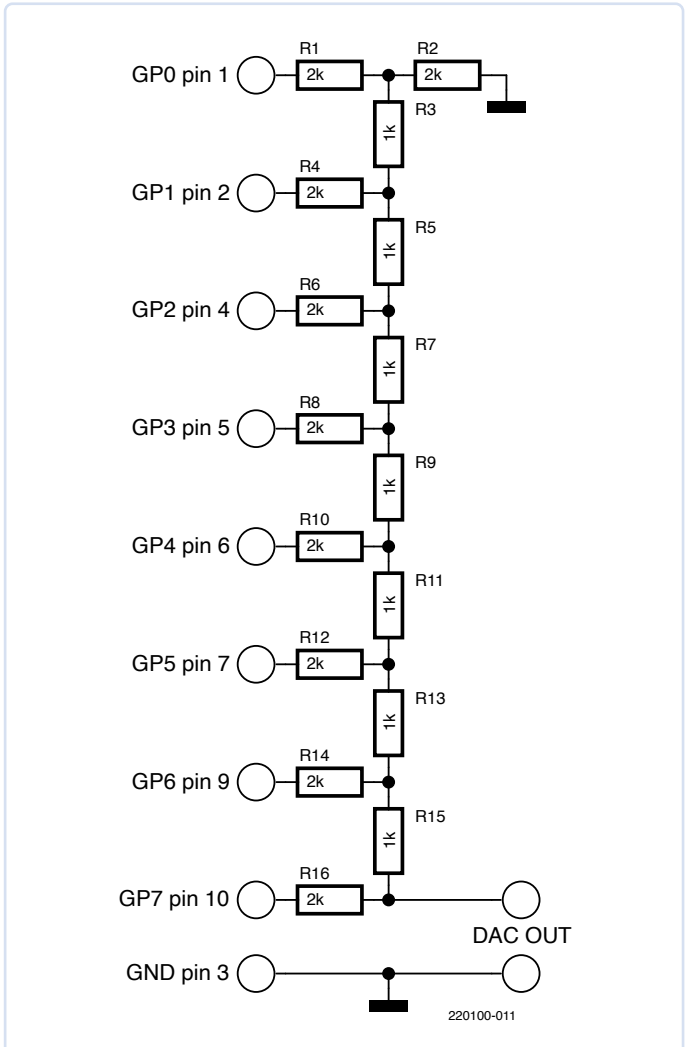


Figure 10. Circuit CD/A-R2R.

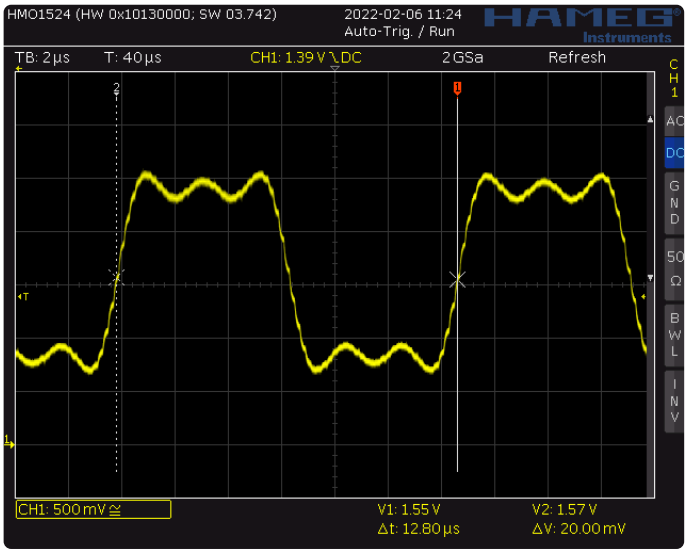


Figure 11. La fréquence de base plus deux harmoniques donnent une onde carrée.

Le contrôleur DMA de commande veille à ce que le contrôleur DMA de données soit redémarré rapidement. Avec cette technique, un nouvel octet peut être émis tous les quatre cycles d'horloge système (125 MHz / 4). Toutefois, le simple convertisseur N/A illustré à la **figure 10** n'est pas adapté à une vitesse de sortie aussi élevée.

Dans la **figure 11**, *clkDiv* a été réglé sur 25. L'horloge du DAC qui en résulte est de 125 MHz / 25 = 5 MHz. Comme une période du signal de sortie dure 64 échantillons, la fréquence générée est calculée comme suit : 5 MHz / 64 = 78,125 kHz. On a utilisé l'approximation de Fourier d'une onde carrée composée d'une onde fondamentale et de deux harmoniques pour produire la forme d'onde de sortie.

Initialisation du registre

L'instruction set peut être utilisée pour initialiser les registres X et Y avec des valeurs fixes. Comme les commandes sont codées sur 16 bits, la commande set ne peut traiter que des constantes comprises entre 0 et 31. Pour initialiser X ou Y avec des constantes plus grandes, vous pouvez utiliser une astuce.

En appelant la routine `pio_sm_exec(pio, sm, instruction)` depuis un programme C, vous interrompez la séquence d'instructions du bloc PIO `pio` et de la machine à état `sm` et insérez la commande `instruction`. On peut utiliser la séquence de fonctions suivante pour initialiser le registre X avec la valeur 500 000.

Listage 9

```
001 pio_sm_put_blocking(pio, sm, 500000);
002 pio_sm_exec(pio, sm,
      pio_encode_pull(false, false));
003 pio_sm_exec(pio, sm,
      pio_encode_mov(pio_x, pio_osr));
```

Tout d'abord, vous placez 500 000 dans la FIFO TX, puis vous exécutez l'instruction `pull` sur le contrôleur PIO, qui amène cette valeur de la FIFO dans l'OSR. Enfin, l'instruction `mov X,OSR` place la valeur dans le registre X comme souhaité. Vous pouvez maintenant lancer le PIO proprement dit. Le programme du **listage 10**, qui fait clignoter une LED, sert d'exemple.

Listage 10

```
001 .program blink
002 .side_set 1 opt
003    mov y,x   side 1
004 yLoop1:
005    jmp y--   yLoop1
006    mov y,x   side 0
007 yLoop2:
008    jmp y--   yLoop2
```

MicroPython

Le RPi Pico est une plateforme bien adaptée au développement de programmes écrits en MicroPython. Il existe une interface Python correspondante pour la programmation PIO. Celle-ci n'est pas programmée à l'aide de l'assembleur PIO, mais en utilisant la syntaxe Python appropriée, qui rappelle fortement un langage de programmation assembleur. Un avantage de l'utilisation de MicroPython est que vous pouvez sauvegarder des projets complets dans un seul fichier. L'exemple de programme associé se trouve dans le **listage MicroPython**.

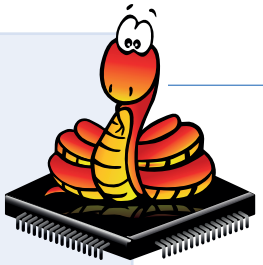
Le programme PIO proprement dit est codé comme la fonction Python `pio_prog()`. Il est constitué du code contenu dans la boucle entre `wrap_target()` et `wrap()`. Le programme bascule la broche *sideset 0* – qui est ici la broche de la LED GP25. Le temps d'activation/désactivation est régi par une boucle d'attente Y. La valeur initiale du registre Y est dans le registre X. La valeur du X est affectée de manière externe via la FIFO TX. Ceci est exécuté par la fonction `pull(noblock)`. Si un mot se trouve dans la FIFO TX, il est transféré dans le registre OSR. Le paramètre `noblock` fait en sorte que le contenu du registre X soit placé dans le registre OSR lorsque l'unité FIFO est vide pour que la séquence de commande continue.

Conclusion

Ici se termine l'aperçu de la programmation PIO du MCU RP2040. Il a été démontré que des interfaces puissantes peuvent être programmées à l'aide de quelques instructions simples. La facilité de programmation PIO de ces MCU les rend plus polyvalents, de sorte que vous n'aurez peut-être même pas besoin d'envisager l'utilisation de FPGA pour étendre les capacités du processeur.

Cet article ne peut donner qu'une vue d'ensemble de la programmation PIO. Il y a tellement plus à explorer. Si vous voulez un bon guide pratique, consultez la littérature [1][2][4][5][6][7], qui contient également de nombreux autres exemples.

220100-04 — VF : Denis Lafourcade



Listage MicroPython

```
001 from machine import Pin
002 from rp2 import PIO, StateMachine, asm_pio
003 from time import sleep
004
005 @asm_pio( sideset_init=(PIO.OUT_LOW))
006 def pio_prog():
007      wrap_target()
008      pull(noblock)
009      out(x, 32)
010
011      mov(y, x) .side(1)
012      label("Loop1")
013      jmp(y_dec, "Loop1")
014
015      mov(y, x) .side(0)
016      label("Loop2")
017      jmp(y_dec, "Loop2")
018      wrap()
019
020 class PIOAPP:
021      def __init__(self, sm_id, pinA, periodLength, count_freq):
023         self._sm = StateMachine(sm_id, pio_prog, freq=count_freq, sideset_base=Pin(pinA)
025         self._sm.active(1) # start PIO execution
026
027      def set(self, value): self._sm.put(value) # put val into TX-FIFO
029
030 pio1 = PIOAPP(0, pinA=25, periodLength=1, count_freq=1_000_000)
032 print("ready1")
033 while True:
034      pio1.set(200000) # 200 ms On/Off time
035      sleep(1) # for one second
036      pio1.set(100000) # 100 ms On/Off time
037      sleep(1) # for one second
038      print("idle")
038      print("idle")
```

Des questions, des commentaires ?

Pour toute question technique relative à cet article, veuillez contacter l'auteur (ossmann@fh-aachen.de) ou contacter Elektor (redaction@elektor.fr).

Produits

- **Raspberry Pi Pico RP2040 (SKU 19562)**
  https://elektor.com/19562
- **Dogan Ibrahim, *« Dogan Ibrahim, Raspberry Pi Pico Essentials »* (SKU 19673)**
  https://elektor.fr/19673

LIENS

[1] Démarrer avec Raspberry Pi Pico : https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf
[2] Manuel d'installation de la chaîne d'outils par Shawn Hymel : https://tinyurl.com/yde5dd8a
[3] Téléchargement du logiciel : http://www.elektormagazine.fr/220100-04
[4] Fiche technique du RP2040 : https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf
[5] Raspberry Pi Pico SDK : https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf
[6] Documentation du SDK : https://raspberrypi.github.io/pico-sdk-doxygen/index.html
[7] RP2040 MicroPython : https://docs.micropython.org/en/latest/library/rp2.html