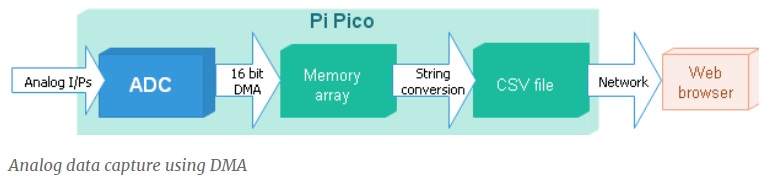
**ENTRADA DEL ADC USANDO DMA Y MICROPYTHON EN LA RASPBERRY PI PICO**

*Embedded systems without the bloat*

<https://iosoft.blog/2021/10/26/pico-adc-dma/>

*October 26, 2021 - DMA, Pico, RP2040*



Esta es la segunda parte de mi proyecto de osciloscopio Pi Pico basado en la web. En la primera parte utilicé un Espressif ESP32 ( <https://iosoft.blog/pico-wireless-web-esp32/> ) para agregar conectividad WiFi al Pico, y ahora estoy escribiendo código para capturar datos analógicos del convertidor analógico a digital (ADC) en el chip, que potencialmente puede proporcionar hasta 500k. muestras/seg.

Las transferencias de alta velocidad como esta normalmente requieren código escrito en C o lenguaje ensamblador, pero he decidido usar *MicroPython*, que es considerablemente más lento, por lo que necesito usar aceleración de hardware para manejar la velocidad de datos, específicamente Acceso Directo a Memoria (DMA).

**'*uctypes*' de MicroPython**

MicroPython no tiene funciones integradas para admitir DMA y no proporciona ninguna forma sencilla de acceder a los registros que controlan los pines ADC, DMA y E/S. Sin embargo, proporciona una forma de definir estos registros, utilizando un nuevo mecanismo llamado "***uctypes***". Esto es vagamente similar a '***ctypes***' en Python estándar, que se usa para definir interfaces de Python para funciones '***foreign***', pero define registros de hardware, usando una sintaxis muy compacta (y algo oscura).

Para dar un ejemplo específico, el controlador DMA tiene múltiples canales y, según la sección 2.5.7 de la hoja de datos del RP2040, cada canal tiene 4 registros, con los siguientes desplazamientos:

0x000 READ\_ADDR

0x004 WRITE\_ADDR

0x008 TRANS\_COUNT

0x00c CTRL\_TRIG

Los primeros tres requieren valores simples de 32 bits, pero el cuarto tiene un campo de bits complejo.

Bit 31: AHB\_ERROR

Bit 30: READ\_ERROR

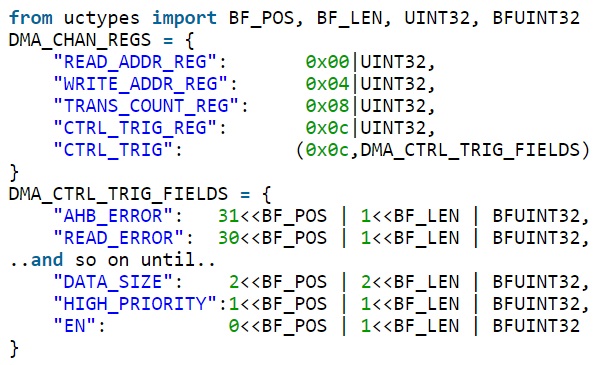
..and so on until..

Bits 3-2: DATA\_SIZE

Bit 1: HIGH\_PRIORITY

Bit 0: EN

Con los ***uctypes*** de *MicroPython*, podemos definir los registros y los campos de bits individuales dentro de esos registros, por ejemplo:



Las entradas ***UINT32***, ***BF\_POS*** y ***BF\_LEN*** pueden parecer extrañas, pero son solo una forma de encapsular el tipo de datos, la posición de los bits y el recuento de bits en una sola variable, y una vez que se ha definido, puede leer o escribir fácilmente cualquier elemento del campo de bits, por ejemplo:

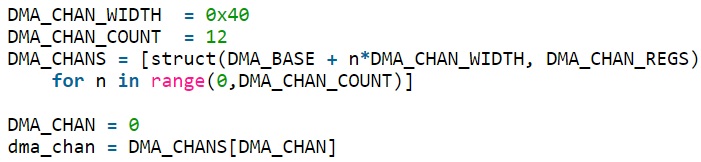
Texto

Descripción generada automáticamente

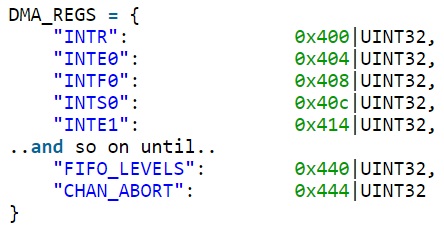
Quizás se pregunte por qué hay dos definiciones para un registro: ***CTRL\_TRIG*** y ***CTRL\_TRIG\_REG***. Aunque es útil poder manipular campos de bits individuales (como en el código anterior), a veces es necesario escribir todo el registro a la vez, por ejemplo, para borrar todos los campos a cero:



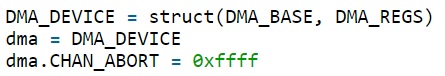
Una complicación adicional es que hay 12 canales DMA, por lo que debemos definir los 12 y luego seleccionar uno de ellos para trabajar:



Para agregar aún más complicaciones, el controlador DMA también tiene un único bloque de registros que no son específicos del canal, por ejemplo:

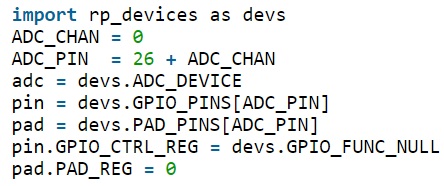


Entonces, para cancelar todas las transacciones DMA en todos los canales:



**Muestra única del ADC**

MicroPython tiene una función para leer el ADC, pero usaremos DMA para capturar múltiples muestras muy rápidamente, por lo que esta función no se puede usar; Necesitamos programar el hardware desde cero. Un primer paso útil es comprobar que podemos producir valores razonables para una única muestra de ADC. En primer lugar, el pin de E/S debe configurarse como entrada analógica, utilizando las definiciones de ***uctype***. Hay 3 canales de entrada analógica, numerados del 0 al 2:



Luego limpiamos el registro de control y estado y el registro de control y estado FIFO; esto sólo es necesario si han sido programados previamente:



Luego habilite el ADC y seleccione el canal a convertir:



Ahora active el ADC para un ciclo de captura y lea el resultado:



Estas dos líneas se pueden repetir para obtener varias muestras.

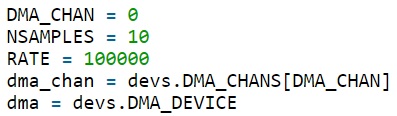
Si el pin de entrada está flotante (no está conectado a nada), entonces el valor devuelto es imposible de predecir, pero generalmente parece estar entre 50 y 80 unidades. El punto importante es que el valor fluctúa entre muestras; Si varias muestras tienen exactamente el mismo valor, entonces hay un problema.

**Múltiples Muestras del ADC**

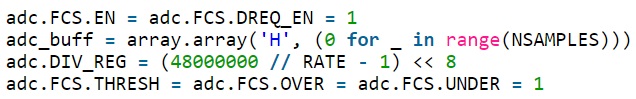
Dado que MicroPython no es lo suficientemente rápido para manejar los datos entrantes, estoy usando DMA, de modo que los valores de ADC se copian directamente en la memoria sin ninguna intervención del software.

Sin embargo, no siempre queremos que el ADC funcione a la velocidad máxima (*500.000 muestras/seg*), por lo que necesitamos alguna forma de activarlo para recuperar la siguiente muestra después de un retraso programable. Los diseñadores del RP2040 se anticiparon a este requisito y lo equiparon con un temporizador programable, controlado por un reloj de 48 MHz. También hay un mecanismo que permite al ADC muestrear automáticamente 2 o 3 entradas por turno; consulte la hoja de datos del RP2040 para obtener más detalles.

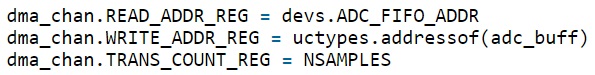
Suponiendo que el ADC se haya configurado como se describe anteriormente, se requiere el código adicional. Primero definimos el canal DMA, el número de muestras y la velocidad (muestras por segundo).



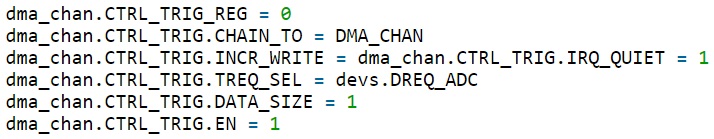
Ahora tenemos que habilitar el ADC FIFO, crear un búfer de 16 bits para contener las muestras y establecer la frecuencia de muestreo:



El controlador DMA está configurado con las direcciones de origen y destino y el recuento de muestras:



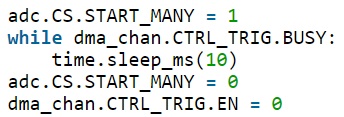
El destino DMA está configurado en incremento automático, con un tamaño de datos de 16 bits; la solicitud de datos proviene del ADC. Luego se habilita DMA, esperando la primera solicitud.



Antes de comenzar el muestreo, es importante limpiar el ADC FIFO leyendo las muestras existentes; si se omite este paso, los datos que obtenga serán una combinación de datos antiguos y nuevos, lo que puede resultar muy confuso.



Ahora podemos configurar el bit ***START\_MANY*** y el ADC comenzará a generar muestras, que se cargarán en su FIFO y luego se transferirán mediante DMA al búfer RAM. Una vez que el búfer esté lleno (es decir, se haya alcanzado el recuento de transferencias DMA y se borre su bit BUSY), las transferencias DMA se detendrán, pero el ADC seguirá intentando colocar muestras en el FIFO hasta que se borre el bit ***START\_MANY***.



Ahora podemos imprimir los resultados, convertidos en una lectura de voltaje:



Al igual que con la prueba de un solo valor, los valores mostrados deberían mostrar algo de vacilación; Si la entrada es flotante, es posible que veas algo como:

['0,045', '0,045', '0,047', '0,046', '0,045', '0,046', '0,045', '0,046', '0,046', '0,041']

**Ejecutando el Código**

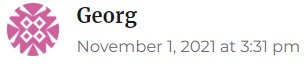
Si no está familiarizado con el proceso de cargar MicroPython en Pico o cargar archivos en el sistema de archivos MicroPython, le sugiero que lea mi publicación anterior (*https://iosoft.blog/pico-wireless-web-esp32/*).

Los archivos fuente están disponibles en Github aquí (*https://github.com/jbentham/pico*); debe cargar el archivo de biblioteca ***rp\_devices.py*** en el sistema de archivos MicroPython y luego ejecutar ***rp\_adc\_test.py***; Normalmente ejecuto esto usando ***Thonny***, ya que simplifica el proceso de edición, ejecución y depuración del código.

En la siguiente parte combino el muestreo ADC y la interfaz de red para crear un osciloscopio en red con una interfaz de navegador *(https://iosoft.blog/oscilloscope-display-pi-pico/*).

Copyright (c) Jeremy P Bentham 2021. Please credit this blog if you use the information or software in it.

**8 Pensamientos sobre "Entrada del ADC usando DMA y Micropython en la Raspberry Pi Pico”**



Muy buen artículo. ¡Gracias por eso!

¿Sería un problema leer los 4 canales al mismo tiempo y esto tendría un efecto en la velocidad máxima de 500k muestras/seg?



No lo he probado todavía, pero según la hoja de datos debería ser posible escanear 4 canales con el ADC funcionando a su máxima velocidad.



Hola, gracias por publicar esta serie de artículos. ¡Estoy tratando de aprender sobre DMA en Pi Pico y tu artículo me proporcionó la entrada perfecta :-)!

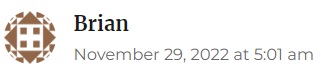
Por cierto, cuando ejecuté el ejemplo de ADC, noté que las primeras ocho muestras de la parte ***Leer Muchos*** son valores anteriores a la ejecución del programa (cambié el voltaje en el pin antes de ejecutar el programa, pero solo los valores noveno y décimo correspondientes a la tensión actual, a pesar de haber leído previamente el FIFO). ¿Alguna idea de cómo vaciar realmente el FIFO y obtener todos los valores correctos?

¡Gracias!

Benjamín



Lamentablemente, no he tenido tiempo de investigar por qué ocurre esto y simplemente utilizo la solución obvia de descartar las primeras 8 muestras, lo que realmente no soluciona el problema.



¡Gracias por el gran artículo! Estoy tratando de hacer algo muy similar en MicroPython, donde tengo un PIO de entrada sincronizada de 8b (en funcionamiento) que puede superar las lecturas estándar del RX FIFO doble. Estoy cerca, sólo quiero una ráfaga de 20 bytes. Quiero probar DMA en MicroPython antes de transferir todo el programa a C, pero no sé cómo configurar DMA para esta condición. ¿Tiene alguna sugerencia sobre cómo configurar los registros para DMA?



Desafortunadamente, no tengo experiencia en programación de Pico DMA en Python, por lo que no puedo ofrecer ninguna ayuda, pero ¿has mirado la función de “***código nativo***” de Micropython? No lo he usado, pero supuestamente ofrece una mejora significativa en la velocidad.



Hola,

Muchas gracias por tu guion. He descubierto que el guión funciona muy bien para grabaciones cortas (3 o 4), pero tiene problemas para una grabación grande. Para grabaciones grandes, tenemos que grabar segmentos de 3 a 4 y transmitirlos al almacenamiento interno o wifi. He probado ambos, sin embargo, después de cada bit de 3 segundos, el retraso introducido por IO (escritura de almacenamiento o transmisión wifi) pierde entre 2 y 3 segundos de audio. Esto es un poco mejor para el almacenamiento interno, pero incluso ese almacenamiento también es muy limitado. ¿Hay algún trabajo alrededor? Creo que se debe a la lentitud de Python y voy a probar C++. Pensé que valía la pena intentar publicarlo aquí.

He probado el siguiente código

TOTSECS = 12 # TOTAL SECONDS OF RECORDING REQUIRED

MEMSECS = 3 # SECONDS OF AUDIO WHICH CAN BE HELD IN MEMORY WITHOUT OVERFLOW

RATE = 8000

NSAMPLES = RATE\*MEMSECS

for i in range(int(TOTSECS/MEMSECS)): # This part triggers the DMA multiple times for fetch audio

# for j in range(NSAMPLES):

# adc\_buff[j] = 0

dma\_chan.READ\_ADDR\_REG = devs.ADC\_FIFO\_ADDR

dma\_chan.WRITE\_ADDR\_REG = uctypes.addressof(adc\_buff)

dma\_chan.TRANS\_COUNT\_REG = NSAMPLES

dma\_chan.CTRL\_TRIG\_REG = 0

dma\_chan.CTRL\_TRIG.CHAIN\_TO = DMA\_CHAN

dma\_chan.CTRL\_TRIG.INCR\_WRITE = dma\_chan.CTRL\_TRIG.IRQ\_QUIET = 1

dma\_chan.CTRL\_TRIG.TREQ\_SEL = devs.DREQ\_ADC

dma\_chan.CTRL\_TRIG.DATA\_SIZE = 1

dma\_chan.CTRL\_TRIG.EN = 1

while adc.FCS.LEVEL:

x = adc.FIFO\_REG

adc.CS.START\_MANY = 1

while dma\_chan.CTRL\_TRIG.BUSY:

time.sleep\_ms(10)

adc.CS.START\_MANY = 0

dma\_chan.CTRL\_TRIG.EN = 0

# print(adc\_buff)

# SENDS DATA OVER WIFI, THE DATA IS CACHED IN RAM AT THE SERVER

try:

res = urequests.post(url=’http://192.168.1.3:5001/login’, data=adc\_buff, headers={‘Content-Type’: ‘application/octet-stream’})

except Exception as e:

print(e)

pass

# FINALLY AT THE END, DATA IS WRITTEN TO DISK IN SERVER IN WAV FILE

res = urequests.get(url=’http://192.168.1.3:5001/success’)



Enviar los datos usando HTTP implica mucha negociación de ida y vuelta con el servidor para crear un enlace TCP y encapsular los datos, por lo que no me sorprende que esto haga un gran agujero en el flujo de datos. Lo primero que probaría es una interfaz de sólo escritura que envíe los datos como datagramas a un servidor UDP; Si eso no es lo suficientemente confiable para su aplicación, puede enviar cada datagrama dos veces, con detección de duplicados.

Si desea seguir con TCP, entonces podría valer la pena probar Websockets (o simplemente un simple servidor TCP) para crear una conexión TCP persistente, lo que evitaría parte de la negociación cada vez que se envía un bloque de datos.

Es posible que traducir su código actual a C no solucione el problema, ya que sospecho que el tiempo de CPU está siendo absorbido principalmente por el código lwIP C, no por su Python. Dentro de unas semanas analizaré toda la cuestión del rendimiento de datos como parte de mi proyecto PicoWi.

**ANTERIOR**

Pi Pico wireless Web server using ESP32 and MicroPython (*https://iosoft.blog/2021/09/26/pico-wireless-web-esp32/*)

**PRÓXIMO**

Web display for Pi Pico oscilloscope ([*https://iosoft.blog/2021/11/10/oscilloscope-display-pi-pico/*](https://iosoft.blog/2021/11/10/oscilloscope-display-pi-pico/)*)*

Lean2 - Blog at WordPress.com ([*https://wordpress.com/?ref=footer\_blog*](https://wordpress.com/?ref=footer_blog))