# DESIGNING A CAMERA MODULE DRIVER USING PROGRAMMABLE I/O ON PI PICO RP2040

A Design Project Report

Presented to the School of Electrical and Computer Engineering of

Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Yibo Yang

M.Eng. Field Advisor: Van Hunter Adams, Bruce Robert Land

Degree Date: December 2022

# Abstract

**Master of Engineering Program**

**School of Electrical and Computer Engineering**

**Cornell University**

**Design Project Report**

**Project Title:**
Designing a Camera Module Driver Using Programmable I/O on Pi Pico RP2040

**Author:**
Yibo Yang

**Abstract:**
This project aims to implement a camera module driver using the Programmable I/O (PIO) on the RP2040 microcontroller from Raspberry Pi. The PIO coprocessors enable high-speed control of the RP2040's I/O ports without any CPU intervention. Furthermore, these coprocessors can communicate data to and from the CPU via Direct Memory Access (DMA) channels. This makes them ideal for implementing high-speed communication protocols with external sensors and devices, like cameras. This report describes the progress made toward that goal. In particular, it describes a Serial Peripheral Interface (SPI) interface to a digital-to-analog converter (DAC) implemented in PIO assembly and fed by DMA channels. This report also describes ongoing work, including integrating an SPI camera module into the system, streaming the input data to a VGA screen, and implementing some other more complex camera interface drivers.

## Executive Summary:

This project aims to implement a camera driver for the RP2040 microcontroller that streams image/video data to a display output without CPU intervention. The driver utilizes the Programmable I/O (PIO) blocks on the RP2040. This report divides into five sections.

The first section motivates the project and introduces the RP2040 microcontroller, the PIO coprocessors, and some Arducam camera modules. The second section describes some challenges associated with this project and some possible solutions. Challenges include high-speed and complex camera interfaces, limited memory on the RP2040, an independent instruction set for PIO development, etc.

The third part provides a detailed description of the development process and solutions for some specific problems. The development process includes three principal milestones: developing a simple SPI driver using PIO assembly, using the SPI driver to drive a DAC, and communicating data to/from these PIO programs using Direct Memory Access (DMA) channels.

The fourth part section summarizes the project's current status and describes the planned course of action for moving the project from its current status to completion. At present, an SPI interface is implemented using the PIO coprocessors to communicate with a Digital-to-Analog Converter (DAC), and two DMA channels communicate data to and from that PIO interface. Future work includes integrating an SPI camera module onto the system, streaming the input data to a VGA screen, and implementing some more complex camera interface drivers.

The fifth part, the sixth part, and the seventh sections include the acknowledgments, references, and appendix respectively.

## 1. Introduction

This project aims to implement a camera driver for the RP2040 that streams data to a display without CPU intervention. After introducing Raspberry Pi Pico, programmable input/output blocks on RP2040, and Arducam camera modules, the motivation and the goal of this project will be addressed.

Raspberry Pi Pico is a microcontroller board based on the Raspberry Pi RP2040 microcontroller chip. The RP2040 chip features dual-core Cortex M0[1], 264k Byte multi-bank high-performance SRAM, with two Programmable (PIO) blocks providing a flexible, user programmable high-speed IO. Its picture is shown in Figure 1. It has been designed to be a low-cost yet flexible development platform for RP2040, with a special key feature—it has 2 Programmable I/O (PIO) blocks, including 8 state machines in total.

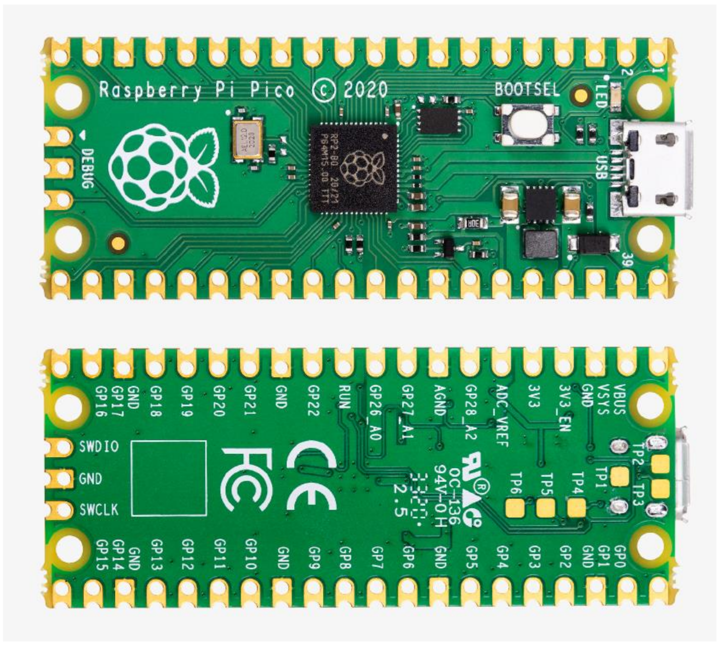


*Figure 1.The Raspberry Pi Pico Rev3 Board*

The programmable input/output block (PIO) is a versatile hardware interface.[1] It can support a variety of IO standards, including 8080 and 6800 parallel bus, I2C, 3-pin I2S, SDIO, SPI, DSPI, QSPI, UART, DPI or VGA (via resistor DAC). PIO is programmable in the same sense as a processor. There are two PIO blocks with four state machines each, that can independently execute sequential programs to manipulate GPIOs and transfer data. The diagram of a PIO block is shown in Figure 2. All four state machines in the same PIO block read from a shared instruction memory that can hold up to 32 instructions. Unlike a general purpose processor, PIO state machines are highly specialized for IO, with a focus on determinism, precise timing, and close integration with fixed-function hardware. Each state machine is equipped with two 32-bit shift registers, two 32-bit scratch registers, four 32-bit bus FIFO in each direction (TX/RX), fractional clock divider (16 integer bits, 8 fractional bits), flexible GPIO mapping, DMA interface, sustained throughput up to 1 word per clock from system DMA, IRQ flag set/clear/status.

PIO state machines are programmed in a software-like manner, which is more user-friendly than a fully configurable logic like an FPGA for embedded software engineers. PIO assembly is compiled along with other software files, presenting a simpler workflow as well as maintaining its high performance and flexibility.

More details about PIO assembly and various functions supported will be referred to in later sections.

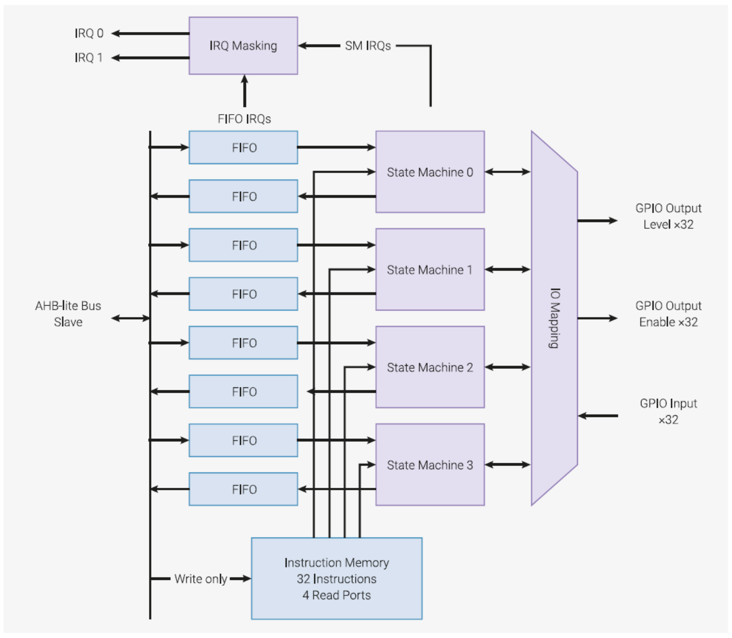


*Figure 2.The diagram of a PIO Block*

Arducam has designed a series of camera modules for Raspberry Pi Pico and third-party RP2040-based boards.[2] There are monochrome cameras that can act as Always-On-Service for machine vision applications, as well as a camera with 5MP color for IoT applications. Some of these cameras can communicate with Raspberry Pi Pico with the SPI (Serial Peripheral Interface) bus. Figure 3 shows the Arducam 2MP SPI camera module. The data rate of the SPI bus can go up to 60 Mbps, which's decent enough for Programmable I/O on the RP2040 to interface with image sensors without using any CPU time, thus saving time for other applications to be processed on the dual-core system.

Initially, this project's goal was to develop a simulator for the PIO state machine since there are no debuggers supporting the development process and all we could only rely on the documents and example programs provided by Raspberry Pi. But near the end of the first semester, we found an open-source PIO state machine emulator that was well implemented by GitJer[3].

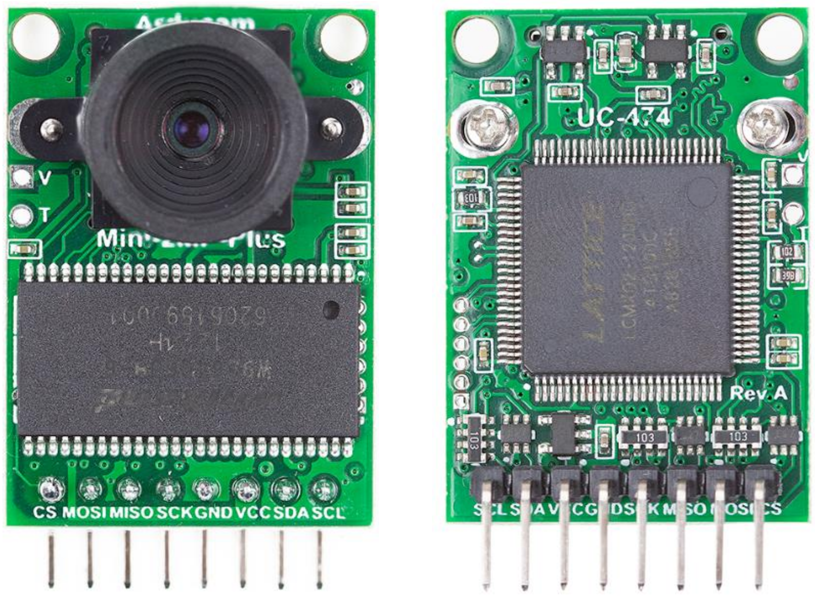


*Figure 3.Arducam 2MP SPI Camera Module*

Therefore, after discussion with my advisors, we shifted the project's direction to developing some specific applications with the programmable input/output blocks. Since a camera interface is rather a complex interface for a microcontroller and has a demand for high speed and memory, we found it very challenging to implement a camera driver on the Pi Pico board with PIO blocks. If a camera system with a real-time display can be developed on such a low-cost microcontroller without using any CPU cycles, it can serve as a good example project for further image applications to be developed upon, such as some Always-On-Service applications, real-time image processing systems, tiny machine learning applications, etc.

As for my personal interest, such a project that pushes a microcontroller to its limit helps me apply the knowledge I learned about embedded systems (operating systems, computer architecture, etc.) into practice. I also expect to learn more about some powerful features of a microcontroller from this project, like DMA and multi-threading supported by the dual-core system.

## 2. Design challenges and possible solutions

Although we expect a big picture of what could possibly be achieved, some real challenges will be encountered and should be analyzed for each step of implementation to be practical and achievable.

At the beginning of this project (after we shifted the goal to implement a camera module driver), a few major problems and expected solutions were discussed below:

1. PIO assembly:

The development of the PIO assembly can be difficult. We are working with a feature that few microprocessors have—hardware blocks that are programmable in the same sense as a processor. The PIO assembly provides nine 16-bit instructions and various functions that maximize the PIO coprocessors' programmability, but increase the development difficulty as well. We thought we could use the PIO state machine emulator we found in the first semester, but it requires Hackaday membership to include Pi Pico libraries. The example given by Raspberry Pi was good enough to implement an SPI interface that functions correctly, so we simply didn't use any debuggers or simulators for the PIO development.

2. Limited memory:

Limited memory can be a problem for image processing applications. Streaming input of large images taken from the camera module can be demanding of memories. It's important how we choose the camera modules with different resolutions, frame rates, and memory capacities. We also have to use our memory on RP2040 efficiently. A buffer that can hold one frame is probably needed between the streaming input and the streaming output, which could be the bottleneck for this system. Without the buffer, perfect synchronization is required: each pixel's data needs to be generated from the camera at the same rate that the VGA requires the pixel's data. In this way, we can get rid of the buffer but it could be hard to achieve that synchronization.

3. No CPU time:

It's hard to imagine a microcontroller working without its CPUs executing instructions. But with full utilization of the PIO blocks and the hardware supported on RP2040, there's still great flexibility for interface implementations.

4. Image processing algorithm:

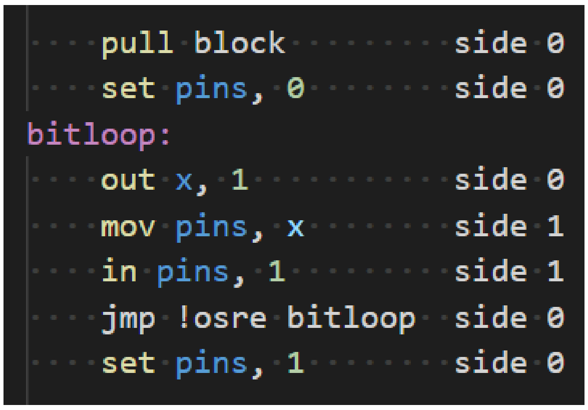
Say we free the CPU from driving the camera and streaming the data to the display output, what work should we assign to the dual-core Cortex M0 system? Can some kind of image processing algorithm be done? It depends on how the data flows from camera input to display output, and how CPUs interact with this process. This is part of the future work that needs to be done. We need to finish the basic camera system and analyze the bottleneck in this system before going on to this step.

## 3. Detailed development process

This section will mainly introduce my work for this project in chronological order: implementing a simple SPI driver using PIO assembly, driving a DAC to output triangle wave using the SPI driver, and using chained DMA to free the CPUs. At the end of this section, some of my own workflow and environment setup will be discussed.

### 1. PIO-SPI driver

I started from an example provided by Raspberry Pi[4]. The PIO assembly programs implement full-duplex SPI, with an SCK period of 4 clock cycles. Both settings of CPHA and CPOL are provided separately in four programs. I adapted one of them and developed an SPI driver with CPHA = 1 and CPOL = 0, with the SCK frequency programmable in the PIO initialization. The PIO assembly is shown in Figure 4 and explained in detail as follows.



```
    pull block          side 0
    set pins, 0         side 0
bitloop:
    out x, 1            side 0
    mov pins, x         side 1
    in pins, 1          side 1
    jmp !osre bitloop   side 0
    set pins, 1         side 0
```

*Figure 4.PIO Assembly Program of the SPI Driver*

The SPI driver includes 4 pins: SCK, MOSI, MISO, and $\overline{CS}$. The specific GPIO pins are initialized by the CPU. Specifically in PIO assembly, instruction "set" corresponds to $\overline{CS}$, "mov" corresponds to MOSI, "in" corresponds to MISO, and function "sideset" corresponds to SCK. This program wraps at the end, that is, after executing the last instruction, the program counter automatically points back to the first instruction, which is very compatible with common I/O interfaces.

It starts from a "pull block", which means that the program stalls when TX FIFO is empty since we only want the state machine to work at a pace that we want. When TX FIFO receives a 16-bit data (user-specified), this instruction loads the data from the TX FIFO into the OSR (output shift register). Then the "set" instruction set the $\overline{CS}$ pin to (active) low, and we enter the "bitloop", where each bit of the data is transmitted serialized via MOSI.

In the bitloop, "out" moves 1 bit from OSR to scratch X register. "mov" sets the corresponding MOSI pin depending on the value in scratch X register, and triggers a rising edge of the SCK. The SCK holds for one more cycle when "in" shifts 1 bit from MISO pin into the ISR (input shift register), and then a falling edge of SCK is triggered by "side 0" of "jmp", which also directs the program counter back to the start of the bitloop if OSR is not empty, which indicates that the data we initially put into the TX FIFO has not been fully transmitted. When all bits are transmitted/OSR is empty, we exit the bitloop and "set" the $\overline{CS}$ pin high.

The bitloop consists of 4 instructions, which is 4 cycles since PIO state machine is a single cycle processor. Therefore, the SCK period is 4 cycles as well, indicating that the SPI frequency is 1/4 of the PIO clock's frequency. The frequency of the PIO clock is configured during setup, which will be discussed in the next section.

An example waveform of $\overline{CS}$ signal and MOSI signal for one SPI transaction is shown in Figure 5.

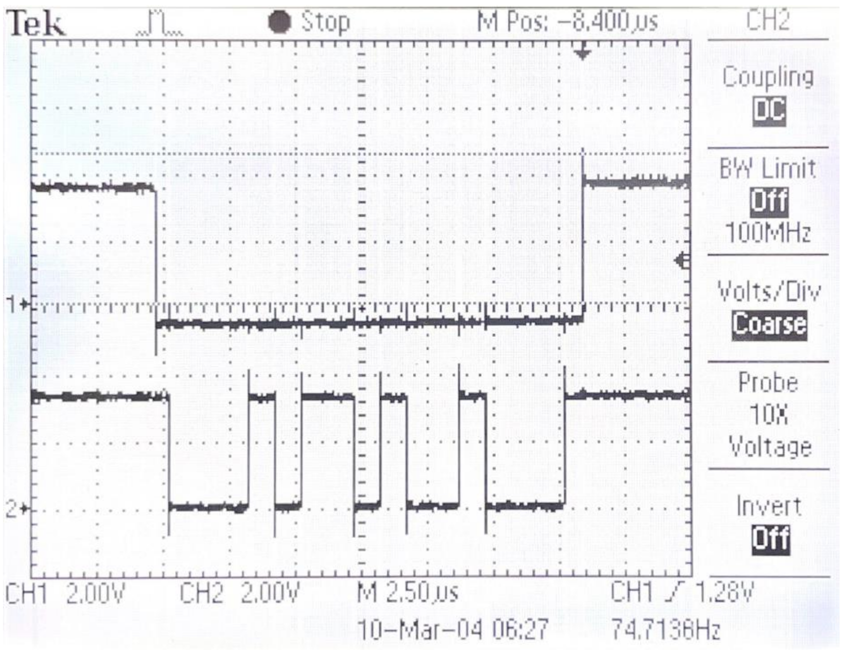


*Figure 5.Waveform of $\overline{CS}$ and MOSI for One SPI Transaction*

## 2. DAC

Next, I used the SPI interface introduced above to drive a 12-bit dual voltage output digital-to-analog converter—MCP4822. It can be driven by an SPI interface with a maximum frequency of 20MHz. Figure 6 shows the pins of this module.

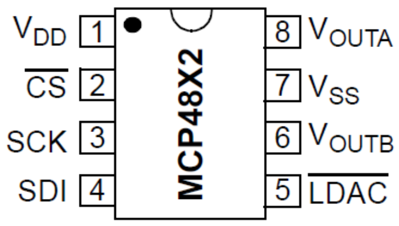
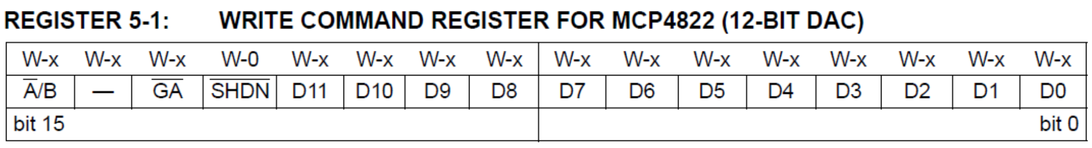


*Figure 6.Pinmap of MCP4822*[5]

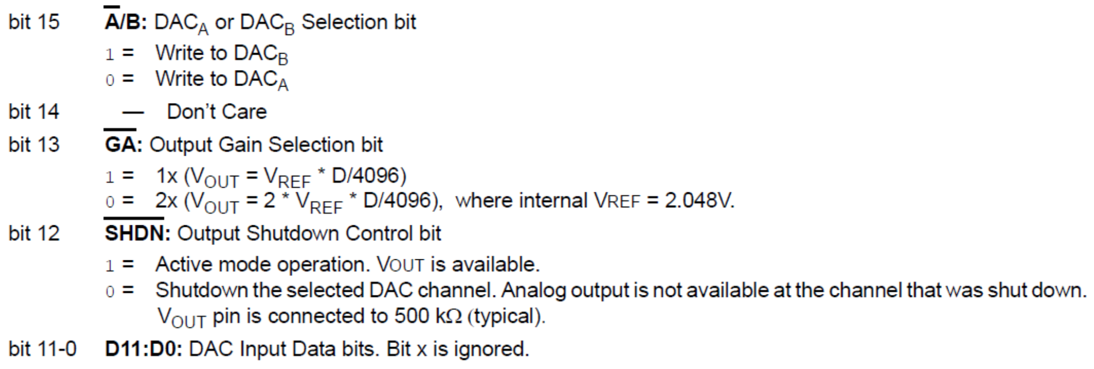
The driver program still uses the CPU to write a 16-bit data into the TX FIFO of a PIO state machine, and lets the PIO state machine to output the data. Specifically, the user defines a 16-bit I/O read/write pointer that points to the physical address of the TX FIFO, and then writes a 16-bit data to the FIFO. The 16-bit data has to comply with the write command of the DAC, that the most significant 4 bits configure the DAC, and the rest 12 bits specify the output voltage.



**REGISTER 5-1: WRITE COMMAND REGISTER FOR MCP4822 (12-BIT DAC)**

| W-x | W-x | W-x | W-0 | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x | W-x |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\overline{A}$/B | — | $\overline{GA}$ | $\overline{SHDN}$ | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| bit 15 | | | | | | | | | | | | | | | bit 0 |

*Figure 7.Write Command Register for MCP4822*



bit 15 **$\overline{A}$/B:** DAC$_A$ or DAC$_B$ Selection bit
1 = Write to DAC$_B$
0 = Write to DAC$_A$

bit 14 — Don't Care

bit 13 **$\overline{GA}$:** Output Gain Selection bit
1 = 1x ($V_{OUT} = V_{REF}$ * D/4096)
0 = 2x ($V_{OUT}$ = 2 * $V_{REF}$ * D/4096), where internal VREF = 2.048V.

bit 12 **$\overline{SHDN}$:** Output Shutdown Control bit
1 = Active mode operation. VOUT is available.
0 = Shutdown the selected DAC channel. Analog output is not available at the channel that was shut down. $V_{OUT}$ pin is connected to 500 kΩ (typical).

bit 11-0 **D11:D0:** DAC Input Data bits. Bit x is ignored.

*Figure 8.Details for Each Bit of the Write Command*

Figure 9 shows a triangle wave output by the DAC, driven by PIO implemented SPI interface, with the CPU specifying the write command of the DAC.

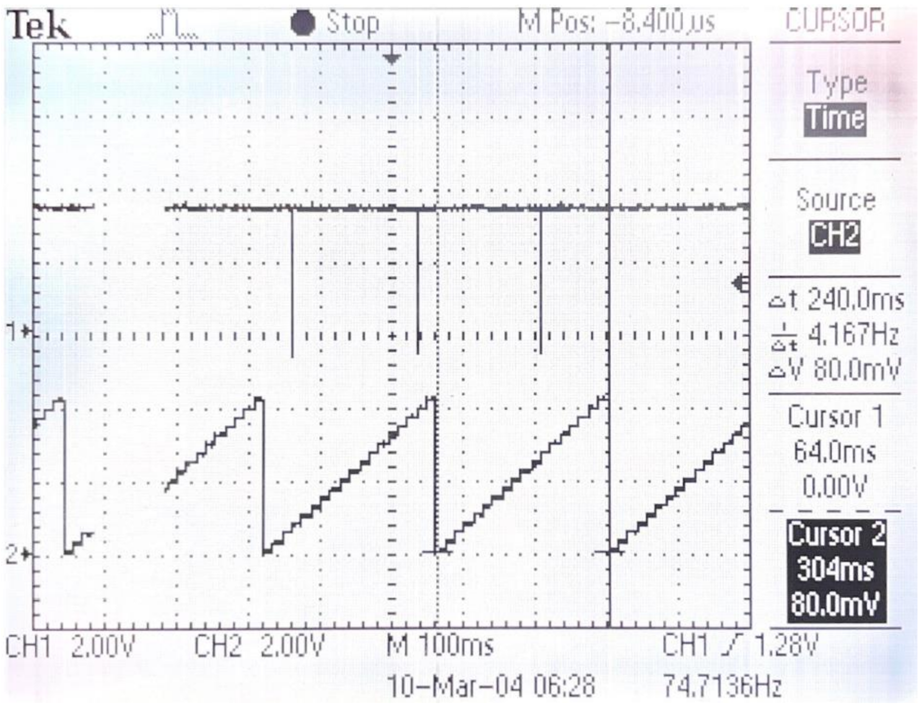


*Figure 9.A Triangle Wave Generated by MCP4822*

## 3. DMA

In addition to the purpose of using no CPU cycles, another reason for using DMA for data streaming is because of the speed. It can perform one read access and one write access, up to 32 bits in size for every clock cycle. The throughput is significantly higher than a Cortex-M0 processor. The DMA architecture overview is shown in Figure 10. Each DMA channel is configured via its own control and status registers

(CSRs), including read address, write address, transfer count, and control register. The software can program and monitor the status of a channel with its CSR.

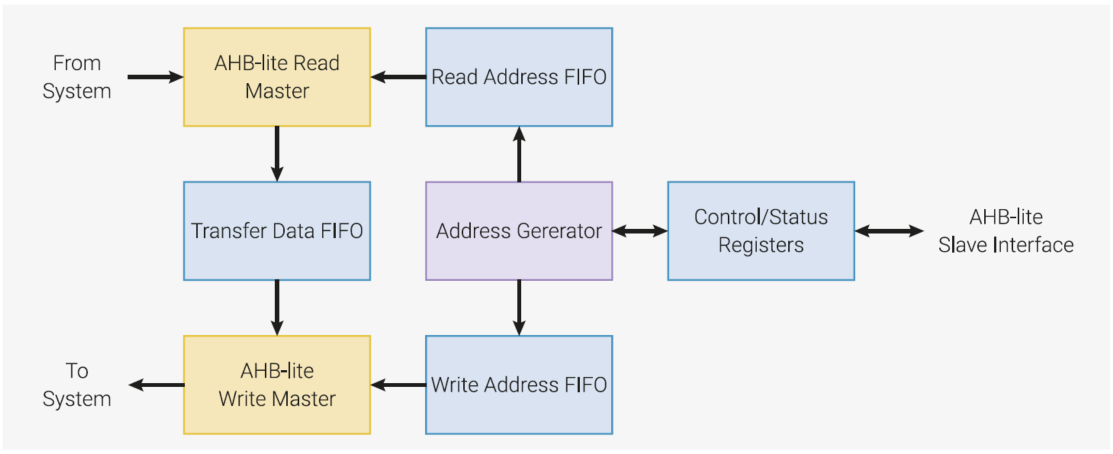


*Figure 10.DMA Architecture Overview*

In an application that requires a repetitive transaction of the same block of data, there are two approaches to reconfigure the data channel: interrupt handler and chained DMA channels. Although both are implemented, the first one still consumes CPU cycles in the interrupt handler and it only serves as an incremental step in the project. Instead, the second one frees the CPU once channel configurations are done. Below are the details for the second implementation:

DMA channel can be set to trigger another channel to immediately start after the first one's transaction is done. By setting the other channel to reconfigure the first DMA channel's CSR, we can have chained DMA channels. Specifically, I set the first channel to transfer data from a predefined wavetable to the TX FIFO. The wavetable stores all the 16-bit commands for the DAC to generate one triangle wave. After the data channel's transaction is done, it automatically triggers the second DMA channel to start. The second DMA channel simply transfers the address of the pointer of the wavetable (it is twisted but this is how it works) to the read address trigger register of the first channel, which triggers the first channel to start over from the beginning of the wavetable. In this way, the two DMA channels continuously trigger each other and keep streaming the wavetable to the TX FIFO, and the PIO state machine works in parallel to output the 16-bit commands in SPI protocol. The waveform generated by the DAC is similar to the CPU one, only that the transaction speed of this implementation is paced by how fast the PIO state machine will empty the TX FIFO, since the data request (DREQ) signal of the data channel is set to the TX FIFO.

## 4. Environment setup

In this section, I will share some ideas and thoughts about the development workflow for Raspberry Pi Pico and the environment setup. Although some notes may not be portable on other platforms, I find it convenient personally.

- My development environment is Windows Subsystem for Linux (WSL), and I use VS Code for programming and compiling. I used to set up the toolchain on

Windows, but I found it somewhat complicated. On the other hand, development under Linux is better documented by Raspberry Pi, I guess it would also be a more popular choice among developers.

- I soldered the pin headers on the bootsel button's side and plugged the board onto a whiteboard to wire it up with other modules (such as DAC). In this way, we can see the names of each pin so that we don't have to check a GPIO map, and we don't need the bootsel button anymore—as long as you link the library 'pico_bootsel_via_double_reset'[6] in the CMakeList.txt before compiling. And then you add a small button to connect the RUN pin to the ground. It can work as a reset button if you tap it once, and if you tap it twice the board will enter the USB ROM bootloader (BOOTSEL mode). All you need to do for fast altering code and re-flashing is to compile the code in the terminal, copy the generated uf2 file, double tap the reset button, paste the file into the popped window, and the program is well loaded.
- After setting up the whiteboard, you can just plug in a USB-to-serial converter for 'printing debugging'. A layout of my Pi Pico on a whiteboard is shown in Figure 11. The button at the left of the board is the reset button; the three colored lines are SCK, CS, and MOSI (MISO is not used yet); the pinheads at the top right of the board is for UART; the module below Pi Pico is MCP4822.

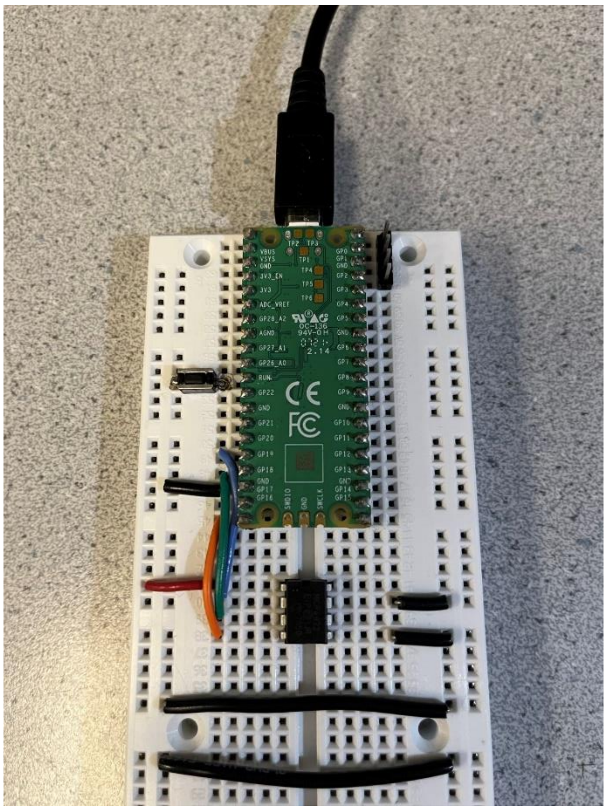


*Figure 11.Whiteboard layout*

## 4. Conclusion and Future work

In conclusion, the current work of this project mainly includes:

- A 2.5MHz SPI interface is generated by PIO assembly and tested with a 12-bit

DAC module, generating an analog triangle wave with 32 levels;

- Use chained DMA channels to free the CPU, the data channel is responsible for pumping a wavetable to the PIO state machine to generate SPI signals, and the configuration channel is responsible for resetting the first channel to repeat the transaction;
- A current VGA example[7] is available to be integrated with the SPI driver, which supports 8 colors and uses DMA and PIO state machines for data transfer.

There is still plenty of future work that needs to be done, including:

- Use the current SPI interface to drive the Arducam 2MP SPI camera module, and then use DMA to stream the input video data to the VGA output for real-time display. An expected system overview is shown in Figure 12;
- Other high-speed camera interfaces could be considered for implementation using programmable I/O blocks;
- The dual-core system could also conduct some other processes, or even interact with the camera system to achieve some more complicated applications like real-time image processing;

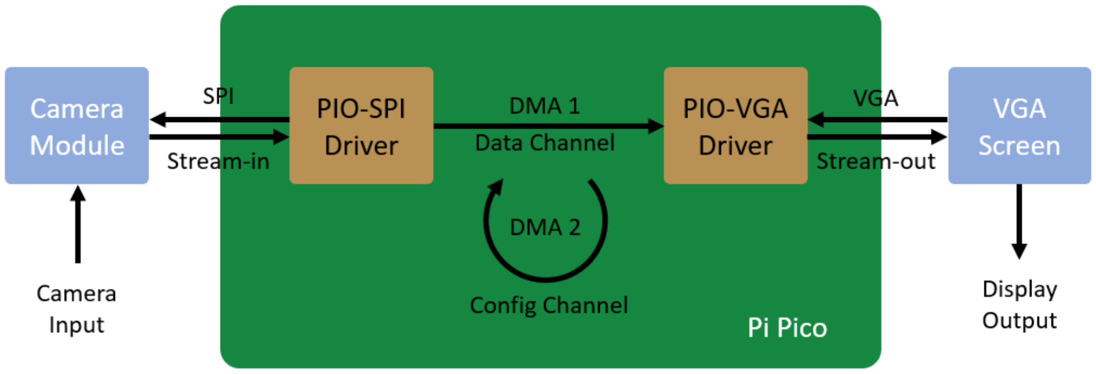


*Figure 12.The Overview of a Fully Implemented Camera System*

## 5. Acknowledgments

I would like to express my sincere gratitude to my project advisors, Dr. Van Hunter Adams and Prof. Bruce Robert Land for their patient guidance along this year and the genuine encouragement for my work and progress on the project.

## 6. Reference

[1] RP2040 Datasheet:
https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf
[2] Arducam Camera Modules:
https://www.arducam.com/raspberry-pi-pico-camera-modules/
[3] Emulator of a Raspberry Pi Pico PIO state machine
https://github.com/GitJer/Som_RPI-Pico_stuff/tree/main/state_machine_emulator
[4] PIO-SPI example provided by Raspberry Pi

https://github.com/raspberrypi/pico-examples/tree/master/pio/spi
[5] MCP4822 Datasheet
https://ww1.microchip.com/downloads/en/DeviceDoc/20002249B.pdf
[6] Library for bootsel via double reset
https://raspberrypi.github.io/pico-sdk-doxygen/group__pico__bootsel__via__double__reset.html
[7] A current PIO-VGA example by Hunter Adams
https://vanhunteradams.com/Pico/VGA/VGA.html

## 7. Appendix

All the programs are in https://github.com/thomasyyb/PiPico.