

PROGRAMMING THE PI PICO RP2040 I/O PROCESSOR

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

Parth Sarthi Sharma

MEng Field Advisor: Prof. Hunter Adams

MEng Outside Advisor: Prof. Bruce Land

Degree Date: December 2021

Abstract

Master of Engineering Program
School of Electrical and Computer Engineering
Cornell University
Design Project Report

Project Title:

Programming the Pi Pico RP2040 I/O Processor

Authors:

Parth Sarthi Sharma

Abstract:

In early 2021, the Raspberry Pi foundation launched a new microcontroller-the RP2040, which is a dual core, ARM Cortex M0 with an innovative input/output processor that can be programmed to produce custom waveforms and serial protocols. The system is programmable in Python or C, but the I/O processor is programmed in a custom assembly language (not ARM assembler). Prof. Hunter Adams and Prof. Bruce Land are considering replacing the existing PIC32 microcontrollers with RP2040 for the course ECE-4760 starting Fall 2022. I used a combination of dual core processor, DMA, and programmable I/O to create many programs and applications like SPI using PIO system and even created a TFT display library running independently on a PIO state machine that will be used by students taking the course starting next year. I also worked on some interesting applications such as Conway's game of life, Google Dino game, Fractals and so on. On top of that, I also tested the performance differences in single core vs dual core applications. Lastly, I tested out various programs on the VGA screen using the VGA library created by Prof. Hunter Adams.

Executive Summary

Currently, the course ECE 4760 revolves around the microcontroller PIC32MX250F128B which is a 32-bit RISC CPU with a 40MHz clock, 128kB flash memory, 32kB SRAM and a few useful peripherals. This microcontroller was launched in November 2007, with the rest of the PIC32MX series which makes it about 14 years old. In early 2021, the Raspberry Pi foundation launched a new microcontroller-the RP2040, which is a dual core, ARM Cortex M0 processor. This microcontroller packs a 125MHz clock which can be overclocked to 133MHz, 2MB on-board flash memory, 264kB SRAM and a lot of other useful peripherals including an innovative input/output processor that can be programmed to produce custom waveforms and serial protocols. The system is programmable in Python or C, but the I/O processor is programmed in a custom assembly language (not ARM assembler). Prof. Hunter Adams and Prof. Bruce Land are considering replacing the existing PIC32 microcontrollers with RP2040 for the course ECE-4760 starting Fall 2022. I used a combination of dual core processor, DMA, and programmable I/O to create many programs and applications like SPI using PIO system and even created a TFT display library running independently on a PIO state machine that will be used by students taking the course starting next year. I also worked on some interesting applications such as Conway's game of life, Google Dino game, Fractals and so on. On top of that, I also tested the performance differences in single core vs dual core applications.

I worked closely with Prof. Hunter Adams and Prof. Bruce Land on testing the RP2040 microcontroller. Since the project was open ended without a fixed specification of the end goal but a more general objective, we met every week for two semesters and decided on the goals for each week. Some of the tasks like using a PIO state machine as an SPI channel to free up the existing SPI channels took multiple weeks, and hence the weekly meetings were concerned more generally with the problems that we faced and how to solve them. During the course of my project, I encountered many issues which I had to deal with and find solutions to. For instance, one of the first issues that I faced was lack of documentation for the microcontroller. Since it is a relatively new microcontroller, the documentation provided on the web is pretty sparse and needs a lot of digging into. In order to address this issue and help the upcoming students to get easier access to the information, Prof. Hunter Adams and I created two separate webpages with all our findings and instructions to replicate the programs we worked on. Another issue that was addressed during the course of this study was the contention among the two cores. Having multiple cores on the processor means taking care of shared memory from contention among the cores. We studied this contention, processor priority and ways to prevent contention in great detail and compared the performance of various codes when running on a single core vs when running on multiple cores.

At the end of the project, we created extensive documentation for ECE 4760 students to use, developed an LCD TFT library which runs on PIO state machines, leaving other ports on the microcontroller open to be used by other peripherals like port expanders. We tested the library by implementing a number of interesting programs, including the snake game and the Google Dino game.

Design Problem

The course ECE 4760 uses the microcontroller PIC32MX250F128B which is a 32-bit RISC CPU with a 40MHz clock, 128kB flash memory, 32kB SRAM and a few useful peripherals. This microcontroller came out over a decade ago which makes it pretty outdated as compared to the recently launched microcontrollers. The issue was to find a microcontroller which is easy to program, powerful enough to replace the existing system and still cost effective so that it can be mass purchased. Luckily, the Raspberry Pi foundation launched the RP2040 microcontrollers in January 2021 which Prof. Hunter Adams and Prof. Bruce Land decided to use for the course.

Once the microcontroller was selected, all that was left to do was to test if it was viable for the course and, if so, to create the necessary documentation and libraries that the students in the upcoming semesters will use. The project solves the problem of creating the documentation for all the necessary GPIO, PIO and peripherals and creating the required libraries for the TFT screen. All these files and the documentation have been uploaded in the form of a website and can be accessed by the students.

To summarize, I have worked on the following aspects of the microcontroller:

- Worked on testing the basic input output and peripheral systems.
- Created a documentation on how to use the same.
- Created a library for the TFT screen earlier used with the PIC32 system.
- Tested the existing labs like the boids lab and checked its limitations.
- Worked out the documentation for multi-core system and contention prevention to replace the protothreads used previously.

Introduction

The Raspberry Pi Pico is a tiny, fast, and versatile board built using RP2040, a brand-new microcontroller chip designed by Raspberry Pi foundation. It features:

- Dual-core Arm Cortex M0+ processor, flexible clock running up to 133 MHz
- 264KB of SRAM, and 2MB of on-board Flash memory
- Low-power sleep and dormant modes
- Drag-and-drop programming using mass storage over USB
- 26 × multi-function GPIO pins
- 2 × SPI, 2 × I2C, 2 × UART, 3 × 12-bit ADC, 16 × controllable PWM channels
- An on-board temperature sensor
- 8 × Programmable I/O (PIO) state machines for custom peripheral support
- Support for C and Python

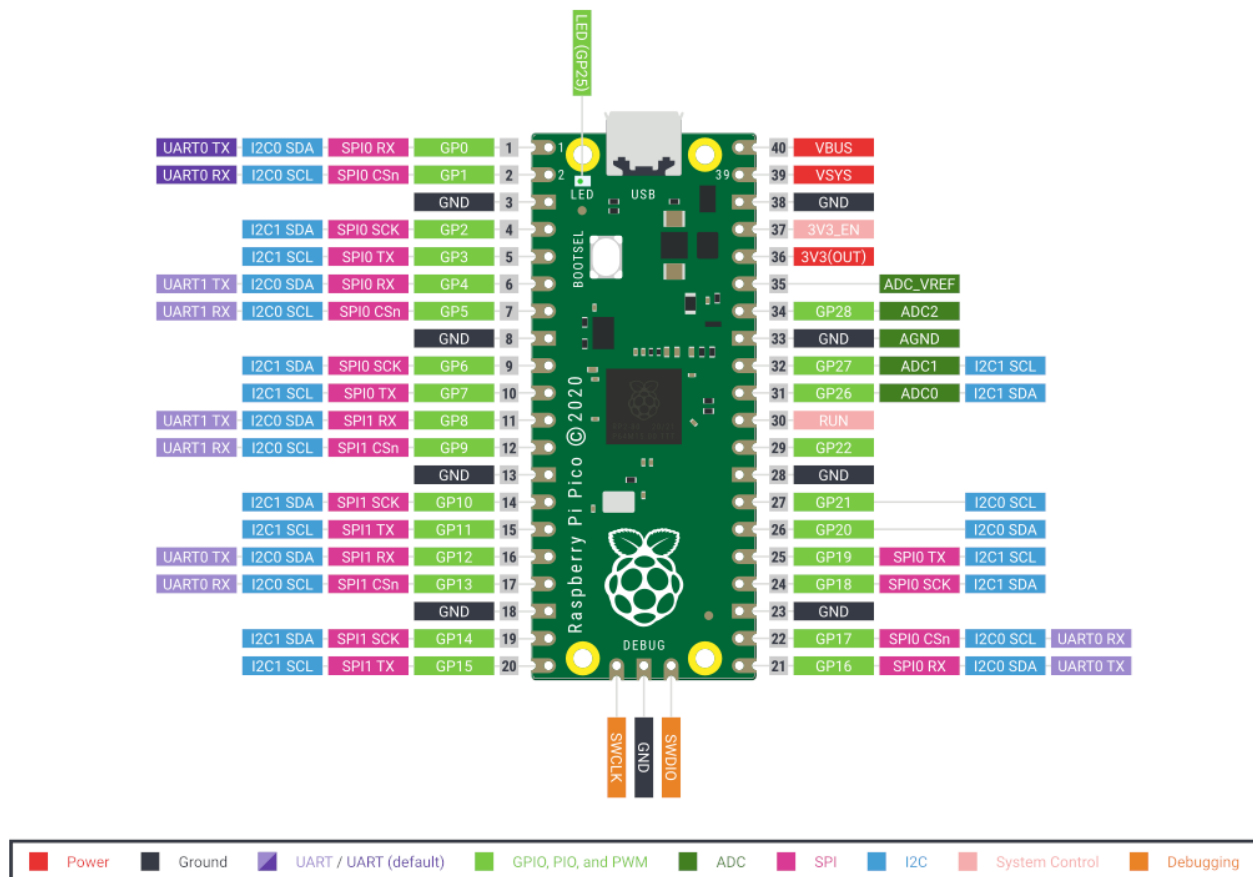


Fig. 1 The Raspberry Pi Pico pinout

The Raspberry Pi Pico can be programmed using C as well as the Python programming language. I chose to program using C because of the following reasons:

- A C program is compiled, and is thus faster than Python.
- I have more experience working with C and is therefore more comfortable for me.

Installing Necessary Software

Before I could get started with the project, it was necessary to install the required software. I used the webpage created by Prof. Hunter Adams (link [here](#)) to set it up. It included the following:

1. Installing the ARM GCC compiler
2. Installing CMake
3. Installing Visual Studio Code
4. Installing Python 3.x
5. Installing Git
6. Downloading the SDK and pico examples

C SDK Architecture

Prof. Hunter Adams' webpage explained this section very nicely and in great detail (link [here](#)). I have derived my following section from there and tried to shorten it down.

With the exception of the C/C++ standard libraries provided by the compiler, all libraries in the SDK are interface libraries. A CMake interface library is a collection of the following:

- Source files
- Include paths
- Compiler definitions (visible to code as #defines)
- Compile and link options
- Dependencies on other interface libraries

Each of these interface libraries contributes source files, compiler definitions, and compile/link options to the build, forming a tree of dependencies. All of these dependencies are gathered in a recursive manner. They're gathered based on the libraries you provided in your CMakeLists.txt file, as well as the libraries those libraries depend on, and so on.

High-level API's

There are high-level libraries (pico xxxx) that allow the user to do things that are cross-cutting between different pieces of hardware. The sleep_ routines in pico time, for example, must be aware of the RP2040's timer hardware as well as how it enters and exits low-power states.

Runtime support libraries

A runtime library is a set of low-level routines used by a compiler to invoke some of the behaviors of a runtime environment, by inserting calls to the runtime library into compiled executable binary.

Hardware support libraries

Individual libraries (hardware xxx) that provide actual APIs for interacting with each piece of physical hardware/peripheral are known as hardware support libraries. They're little and merely provide shallow abstractions. Rather than accessing registers directly, they usually provide functions for setting or dealing with peripheral hardware at a functional level. The goal of these libraries is to have a very low runtime cost. The hardware structs and hardware regs libraries, which include definitions of memory-mapped register layout on the RP2040, are their only dependencies.

Hardware structs library

The hardware structs library contains a set of C structures that reflect the RP2040 registers' memory mapped layout in the system address space. Both the hardware libraries and the hardware regs register headers use the same names for the struct headers. So, if you use the hardware pio library's functionality through hardware/pio.h, the hardware structs library (which is a dependency of hardware pio) contains a header you can include as hardware/structs/pio.h if you need to access a register directly, and this in turn pulls in hardware/regs/pio.h for register field definitions.

Hardware registers library

These are the most basic libraries available. The hardware regs library is a set of inclusion files for all RP2040 registers that was produced automatically from the hardware. These are strongly annotated, and they define the offset of each register, as well as the structure of the fields in those registers and the field's access type (e.g. read-only).

The build system

CMake is used to handle builds in the Pico SDK. The CMakeLists.txt project files define how your application or project should be built. "CMake is crucial to the way the SDK is constructed, and how applications are configured and built," according to the SDK handbook.

Some of the most commonly used syntax and ideas are as follows:

- The method `add_executable(programName fileName.c)` in this file declares that a program named `programName` should be built from the C source file `fileName.c`. This will also be the program's target name for building, allowing the user to build the app by typing `make programName` in the build directory.
- `target_link_libraries(programName library1 library2... libraryN)` loads the SDK functionality required by the program. If you don't request a library, it will not be included in the binary of your software.
- UF2 files are generated by `pico add extra_outputs(programName)` for Pico USB loading. If we didn't include this, the system would generate an ELF file (executable linkable format) that could be loaded onto the Pico using a debugger like `gdb` or `openocd` via the Serial Wire Debug connection. This also generates `.hex`, `.bin`, `.map`, and `.dis` files.

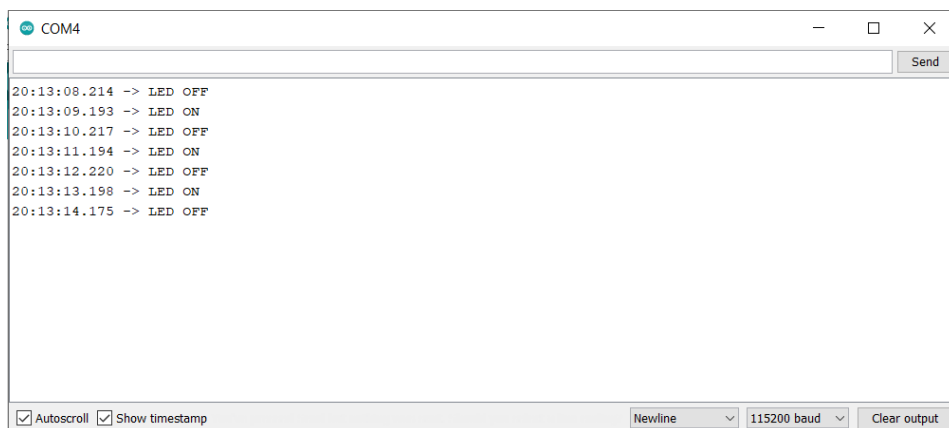
Hello World

As is natural with any new programming language, I started off with the `Hello World` of the embedded world: the LED blinking program. This was also an introduction to the digital output on a GPIO pin. This program was a slight modification of the blink program provided in the pico examples by the RaspberryPi foundation.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are interface libraries. The next include pulls in hardware API which is not already brought in by pico/stdlib.h. As the name suggests, this interface library gives us access to the API's associated with the hardware GPIO pins on the RP2040.
2. The next section of the code is basically a single line which #define's the LED pin number (GPIO 25 is linked to the on-board LED).
3. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
4. In the next 2 lines of the code, I initialized the LED pin and configured it to be the output pin. The gpio_init() function is used to initialize the pin and the gpio_set_dir() function is used to set the pin direction which can be GPIO_OUT (output) or GPIO_IN (input).
5. The last part of the program is the infinite while loop. This is the loop which runs forever and executes the code sequentially. It basically contains 2 subsections: turning the LED on and turning the LED off. I also used the printf() statement to print the output to the screen. In order to see the output, I used the serial monitor provided by the Arduino IDE. Then I used gpio_put() to set the pins HIGH or LOW. Lastly, I used the sleep_ms() to put the CPU to sleep for 1 second for both HIGH and LOW.

In order to view the output, I used the serial monitor provided by the Arduino IDE. As it shows, the LED is toggling at an interval of 1 second.



```
COM4
20:13:08.214 -> LED OFF
20:13:09.193 -> LED ON
20:13:10.217 -> LED OFF
20:13:11.194 -> LED ON
20:13:12.220 -> LED OFF
20:13:13.198 -> LED ON
20:13:14.175 -> LED OFF
```

Fig. 2 Output of the Hello World program

Analog to Digital Converter

This program was an introduction to ADC input in order to control the brightness of an LED connected to a GPIO pin. A 3-pin potentiometer was used to control the input to the ADC of the Pico. It was a combination of the ADC and PWM examples provided by the RaspberryPi foundation. This was also an introduction to the input on a GPIO pin.

The RaspberryPi Pico has a 12-bit ADC. That means that the range of ADC input is 0 to 4095 for the given voltage range (0-3.3V here). Then, I used this input to control the brightness of the LED. Since the maximum of ADC input is 4095 and the maximum PWM interrupt cycles in a signal is 65536, I chose a multiplier of 16 in order to scale the ADC input to the duty cycle.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries. The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/irq.h, hardware/pwm.h, hardware/adc.h, hardware/gpio.h and pico/time.h. As the names suggest, these interface libraries give us access to the API's associated with the hardware PWM, hardware IRQ, hardware ADC, hardware GPIO and pico time on the RP2040.
2. The wrapHandler() is the function that is called every time the PWM timer throws an interrupt. I cleared the interrupt as soon as the interrupt handler is called. Since I wanted to change the brightness according to the input from the ADC, I used a multiplier of 16 in order to scale the ADC input to the duty cycle. I used the pwm_set_gpio_level() function to change the duty cycle. Since the PWM in the RP2040 is 16-bit wide, it can take in values from 0 to 65536. If the input value is 0 then the duty cycle is 0% while if the input is 65536, the duty cycle is 100% (only if the PWM wrapping is set to be 0xFFFF). Every time the handler function is called, the new brightness level is sent into the pwm_set_gpio_level() function to change the duty cycle.
3. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
4. I used the function gpio_set_function(LED, GPIO_FUNC_PWM); to set the LED as a PWM pin. Then I used the pwm_gpio_to_slice_num() function to get the PWM slice number for the LEDPin. Next, I used the pwm_clear_irq() to clear the interrupt on the given PWM slice. This allows me to enable the PWM interrupt on the given slice using the function pwm_set_irq_enabled().

After setting up the PWM interrupt for the given pin, I had to configure the interrupt handler function. I used the irq_set_exclusive_handler() function to do so. Now, whenever the PWM interrupt flag is set, it calls the interrupt handler function. All I needed to do next was to call the irq_set_enabled() function to enable the interrupt.

Now that the interrupt was setup, it was time to configure the PWM. In order to do so, I used the pwm_get_default_config() function to get the default configurations for the PWM. According to the SDK documentation "PWM config is free running at system clock speed, no phase correction, wrapping at 0xffff, with standard polarities for channels A and B." Right now, the PWM interrupt will be thrown every single clock cycle. In order to avoid that, I used the pwm_config_set_clkdiv() function to set the clock divider to 4 so that the PWM interrupt is

thrown every 4 clock cycles. Lastly, I initialized the PWM with the set configurations using the `pwm_init()` function.

5. In order to use the ADC, I first initialised the ADC using the `adc_init()` function. I then initialized the GPIO 26 using the `adc_gpio_init()` function. From the datasheet, I know that the ADC inputs 0 to 3 are connected to GPIOs 26 to 29 respectively. In order to select the input, I used the `adc_select_input()` function.
6. The last part of the program is the infinite while loop. For this code, the infinite while loop has only one function: read the input from the ADC and sleep for 10 milliseconds. In order to read the input from the ADC, I used the `adc_read()` function. The `sleep_ms()` function is important because ADC takes some time to read and store the output. If the sleep function is removed, the `adc_read()` function is called repeatedly and the CPU doesn't have enough time to store the result.

I used the clock divider to be 4. Therefore, the PWM interrupt was being called at $\frac{sys_clk}{4} = \frac{125}{4} MHz = 31.25MHz$. Moreover, 1 PWM cycle consists of 65536 interrupt cycles. Therefore, the PWM frequency that should be generated is $\frac{interrupt\ frequency}{65536} = \frac{31.25}{65536} MHz = 476.83Hz$.

The scope trace below shows the PWM output from the LED pin. The text in the top left corner of the screen confirms that the frequency of the generated PWM wave is infact 476.9 Hz. Moreover, when I change rotate the potentiometer, it also changes the duty cycle of the PWM signal and behaves as it should.

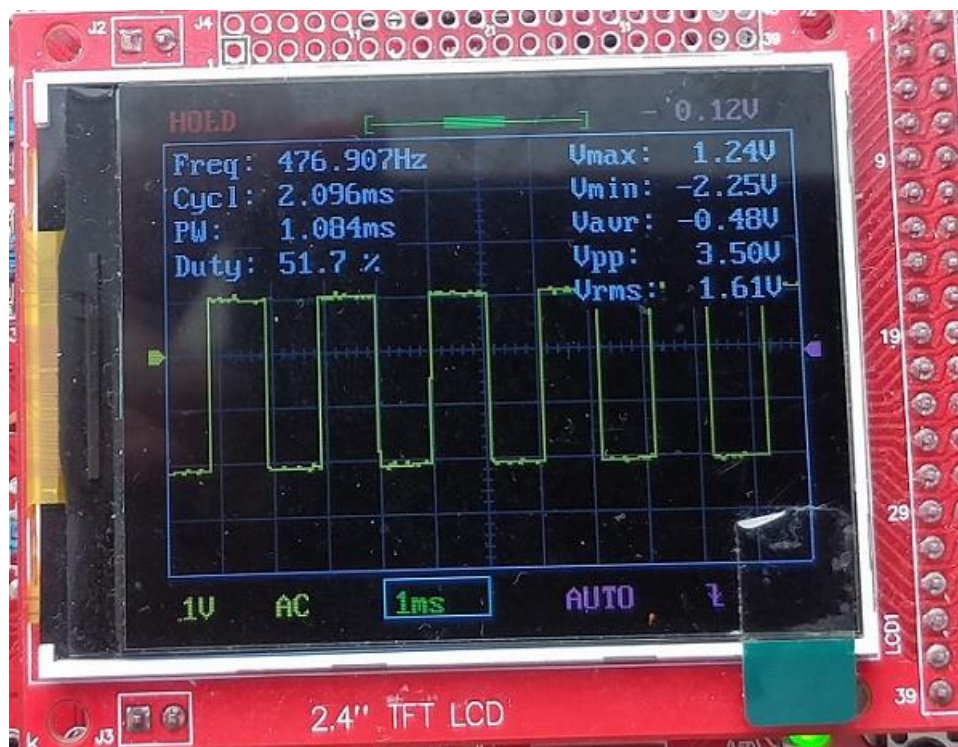


Fig. 3 The output of the ADC program

ADC Input UART

This program was an introduction to digital input in order to control the number of LEDs glowing while simultaneously controlling their brightness based on ADC input from a 3-pin potentiometer. This was also an introduction to the output using UART. The RaspberryPi Pico has a 12-bit ADC. That means that the range of ADC input is 0 to 4095 for the given voltage range (0-3.3V here). Then, I used this input to control the brightness of the LED. Since the maximum of ADC input is 4095 and the maximum PWM interrupt cycles in a signal is 65536, I chose a multiplier of 16 in order to scale the ADC input to the duty cycle. It was a combination of the ADC, PWM and UART examples provided by the RaspberryPi foundation.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries. The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/irq.h, hardware/pwm.h, hardware/adc.h, hardware/gpio.h, hardware/uart.h and pico/time.h. As the names suggest, these interface libraries give us access to the API's associated with the hardware PWM, hardware IRQ, hardware ADC, hardware GPIO, hardware UART and pico time on the RP2040.
2. The next section of the code is the #define's and the global variables which will be used throughout the code. The #define's include the pushbutton, the UARTTX and the UARTRX GPIO pins. The global variables include a GPIO array, an array of PWM slices, a level indicator which keeps a track of the number of LEDs that are glowing and the brightness tracker. We chose GPIO 0 and GPIO 1 for UART because they are directly connected to the UART0.
3. The wrapHandler() is the function that is called every time the PWM timer throws an interrupt. I cleared the interrupt as soon as the interrupt handler is called. Since I wanted to change the brightness according to the input from the ADC, I used a multiplier of 16 in order to scale the ADC input to the duty cycle. I used the pwm_set_gpio_level() function to change the duty cycle. Since the PWM in the RP2040 is 16-bit wide, it can take in values from 0 to 65536. If the input value is 0 then the duty cycle is 0% while if the input is 65536, the duty cycle is 100% (only if the PWM wrapping is set to be 0xFFFF). Every time the handler function is called, the new brightness level is sent into the pwm_set_gpio_level() function to change the duty cycle of the required number of pins. For the rest of the pins, the duty cycle is set to 0.
4. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
5. In order to initialize the UART, I used the uart_init() function. It puts the UART into a known state, and enables it. Next, in order to map the UART functionality to the GPIO pins, we used the gpio_set_function() for both UARTTX and UARTRX pins.
6. I used the function gpio_set_function(); to set the LED pins as PWM pins. Then I used the pwm_gpio_to_slice_num() function to get the PWM slice numbers for the LED pins. Next, I used the pwm_clear_irq() to clear the interrupts on the given PWM slices. This allows me to enable the PWM interrupt on the given slices using the function pwm_set_irq_enabled(). After setting up the PWM interrupt for the given pin, I had to configure the interrupt handler function. I used the irq_set_exclusive_handler() function to do so. Now, whenever the PWM

interrupt flag is set, it calls the interrupt handler function. All I needed to do next was to call the `irq_set_enabled()` function to enable the interrupt.

Now that the interrupt was setup, it was time to configure the PWM. In order to do so, I used the `pwm_get_default_config()` function to get the default configurations for the PWM. According to the SDK documentation "PWM config is free running at system clock speed, no phase correction, wrapping at 0xffff, with standard polarities for channels A and B." Right now, the PWM interrupt will be thrown every single clock cycle. In order to avoid that, I used the `pwm_config_set_clkdiv()` function to set the clock divider to 4 so that the PWM interrupt is thrown every 4 clock cycles. Lastly, I initialized the PWM with the set configurations using the `pwm_init()` function.

7. In the next 2 lines of the code, I initialized the button pin and configured it to be the input pin. The `gpio_init()` function is used to initialize the pin and the `gpio_set_dir()` function is used to set the pin direction which can be `GPIO_OUT` (output) or `GPIO_IN` (input).
8. In order to use the ADC, I first initialized the ADC using the `adc_init()` function. I then initialized the GPIO 26 using the `adc_gpio_init()` function. From the datasheet, I know that the ADC inputs 0 to 3 are connected to GPIOs 26 to 29 respectively. In order to select the input, I used the `adc_select_input()` function.
9. The last part of the program is the infinite while loop. For this code, the infinite while loop follows the following algorithm over and over again:
 - Read the input from the ADC and store it as brightness.
 - Check if the pushbutton has been pressed.
 - If the pushbutton is pressed, increment the level (number of glowing LEDs).
 - If the level variable hits 6, reset it to 0;
 - Put "Hello world!\n" on the UART channel and print it on the screen.

In order to read the input from the ADC, I used the `adc_read()` function. The `gpio_get()` function is used to get the digital state of the GPIO pin (0 for low, non-zero for high). I also used a `sleep_ms(200)` to act as the debounce time for the pushbutton. Finally, the `uart_puts()` function is used to write string to UART for transmission.

I used the clock divider to be 4. Therefore, the PWM interrupt was being called at $\frac{sys_clk}{4} = \frac{125}{4} MHz = 31.25 MHz$. Moreover, 1 PWM cycle consists of 65536 interrupt cycles. Therefore, the PWM frequency that should be generated is $\frac{interrupt\ frequency}{65536} = \frac{31.25}{65536} MHz = 476.83 Hz$. The scope trace below shows the PWM output from the LED pin. The text in the top left corner of the screen confirms that the frequency of the generated PWM wave is in fact 476.69 Hz. Moreover, when I change rotate the potentiometer, it also changes the duty cycle of the PWM signals on all the turned-on LED pins and behaves as it should. Moreover, pressing the pushbutton also changes the number of LEDs that are glowing.

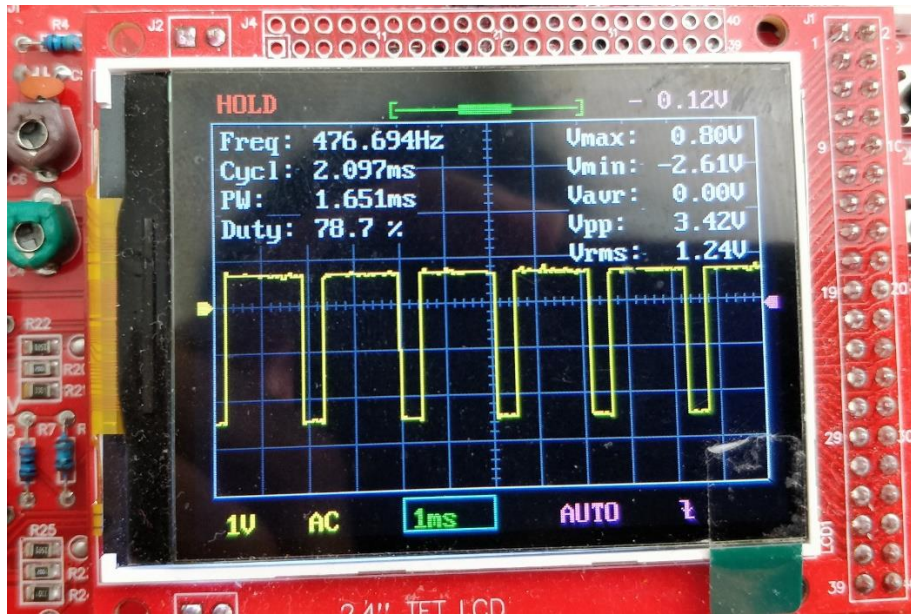


Fig. 4 PWM signal generated based on value from the ADC

I checked the output of the UART on the Arduino serial monitor and it looks as follows.

```
13:44:49.764 -> Hello world!  
13:44:49.810 -> Hello world!
```

Fig. 5 Output on the Arduino serial monitor

GPIO Interrupt

This program was an introduction to attaching interrupts on a GPIO pin. An interrupt can be generated for every GPIO pin in four scenarios: the GPIO is logical 1, the GPIO is logical 0, there is a falling edge or there is a rising edge.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries. The next include pulls in hardware APIs which are not already brought in by pico/stdlib.h. This is the hardware/gpio.h library which gives us access to the API associated with the hardware GPIO on the RP2040.
2. The next section of the code is the #define's and the global variables which will be used throughout the code. These are the pin number (I attached a push button to this pin) and the counter variable to keep track of the number of times the button was pressed.
Note: The counter variable needs to be volatile as it is being changed by an interrupt.
3. The gpio_callback() is the function that is called every time the pushbutton calls the interrupt. Since there can only be one interrupt callback associated with the GPIO pins, it takes in the GPIO pin and the event as arguments to take decisions based on them. My interrupt handler increments the counter variable and prints it out on the console.
4. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
5. I used the gpio_set_irq_enabled_with_callback() function enable interrupts for the GPIO pin. Note: Currently the GPIO parameter is ignored, and this callback will be called for any enabled GPIO IRQ on any pin.
6. The last part of the program is the infinite while loop. The infinite while loop for this program is an empty while loop which serves only to keep the core running and not let it exit.

In order to view the output, I used PuTTY. As it shows, the counter variable increments as I press the button.



```
COM4 - PuTTY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Fig. 6 The output of the button presses

Basic Timer

This program was an introduction to the timer the system timer peripheral on RP2040. It provides a global microsecond timebase for the system, and generates interrupts based on this timebase. This program fetches and prints the absolute time (the time elapsed since boot) on the serial monitor. The timer peripheral on RP2040 supports:

- A single 64-bit counter, incrementing once per microsecond
- This counter can be read from a pair of latching registers, for race-free reads over a 32-bit bus.
- Four alarms: match on the lower 32 bits of counter, IRQ on match.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries. The next include pulls in hardware API which is not already brought in by pico/stdlib.h. This is the include pico/time.h. As the name suggests, this interface library gives us access to the API associated with the pico time on the RP2040.
2. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
3. For this code, the infinite while loop follows the following algorithm over and over again:
 - Fetch the absolute time.
 - Print the fetched absolute time on the screen.
 - Sleep for 1000 milliseconds.

In order to fetch the absolute time, I used the get_absolute_time() function. This function returns the time since boot in microseconds as an unsigned 64-bit integer. Therefore, it is going to continue to run for 5851444 years after it boots up! Once the value is fetched, it is printed out on the serial monitor and the CPU sleeps for 1 second.

The output of the program is shown below. From the image it is verified that the time since boot (in microseconds) is being printed out at an interval of 1000 milliseconds.

```
16:28:28.473 -> Time: 13002380
16:28:29.497 -> Time: 14002534
16:28:30.474 -> Time: 15002649
16:28:31.496 -> Time: 16002734
16:28:32.475 -> Time: 17002814
16:28:33.497 -> Time: 18002900
16:28:34.479 -> Time: 19002986
16:28:35.505 -> Time: 20003070
16:28:36.486 -> Time: 21003154
16:28:37.469 -> Time: 22003239
```

Fig. 7 PWM signal generated based on value from the ADC

Direct Digital Synthesis using Interrupts

This program was an introduction to the SPI communication protocol on the Raspberry Pi Pico. RP2040 has two identical SPI controllers, both based on an ARM Primecell Synchronous Serial Port (SSP). I used an MCP4822 DAC to generate a sine wave of a given frequency. The Pico transmits data to the DAC using SPI. This program was also a test to see if the existing Lab 1 in the course ECE 4760 replicable using the RP2040 microcontroller.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the standard math header(math.h). The others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries.
The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/gpio.h, hardware/adc.h, hardware/irq.h, hardware/spi.h, pico/time.h. As the names suggest, these interface libraries give us access to the API's associated with the hardware GPIO, hardware adc, hardware irq, hardware spi and pico time on the RP2040.
2. The next section of the code is the #define's and the global variables which will be used throughout the code.

The following are the #define's to be used throughout the code:

- Chip Select pin
- Master Out Slave In pin
- Master In Slave Out pin
- The Serial Clock pin
- The SPI Port
- The number of elements in the sine table
- The sampling frequency for the generated signal
- Pre-calculated value for $2^{32}/F_s$
- The configuration bits (to be used as a mask) for the DAC

The following are the variables to be used throughout the code:

- The Phase Accumulator variable to keep a track of the "angle" of the phasor
 - The Phase Incrementor variable to keep a track of the amount to be added to the accumulator
 - The ADC input
 - The modified DAC data to be sent
 - The sine lookup table to contain the amplitudes for a single period of a sine wave
3. In order to get a constant sampling frequency of 44kHz, I used an interrupt which triggers every $23\mu s$ ($1 / 44000$). In order to handle the interrupt, I created an interrupt handler which does the following:
 - Reads the ADC value in the variable adcIn
 - Calculate the phase incrementor. The frequency to be generated is the input from the ADC (0 - 4095Hz)

- Calculate the phase accumulator
 - Calculate the DAC data. It is the data from the sine table indexed using the 8 MSBs
 - Mask the configurations bits on the DAC data
 - Write the DAC data to the SPI port
4. The first line in main() is a call to `stdio_init_all()`. This function initializes stdio to communicate through either UART or USB, depending on the configurations in the `CMakeLists.txt` file.
 5. The next line initializes a repeating timer called `timer` to use it to trigger the interrupts.
 6. Next, I initialize a 256-element wide sine table in order to contain the amplitudes for a single period of a sine wave.
 7. In order to use the ADC, I first initialised the ADC using the `adc_init()` function. I then initialized the GPIO 26 using the `adc_gpio_init()` function. From the datasheet, I know that the ADC inputs 0 to 3 are connected to GPIOs 26 to 29 respectively. In order to select the input, I used the `adc_select_input()` function.
 8. In order to initialise SPI instance, I used the `spi_init()` which takes in the SPI port and the baud rate as arguments. Then I used the `gpio_set_function()` to initialize the SPI pins with their respective functions. Lastly, I used the `spi_set_format()` function to configure how the SPI serialises and deserialises data on the wire.
 9. In order to initialize the interrupt on the timer, I used the `add_repeating_timer_us()` function. It takes the following parameters as arguments:
 - The delay in microseconds: The first argument is the delay in microseconds. If the delay is positive, then this is the delay between one callback ending and the next starting. If the delay is negative, then this is the time between the start of two callbacks.
 - The callback function: This is the function that will be called as soon as the interrupt occurs. This is the interrupt handler, so to speak.
 - The user data: This is the user data to pass to store in the `repeating_timer` structure for use by the callback.
 - The pointer to timer: This is the pointer to the user owned structure to store the repeating timer info in.
 10. The last part of the program is the infinite while loop. The infinite while loop for this program is an empty while loop which serves only to keep the core running and not let it exit.

In order to view the output of the DAC, I used an oscilloscope. As it is quite evident from the oscilloscope output, the output of the DAC is a sine wave of the desired frequency.

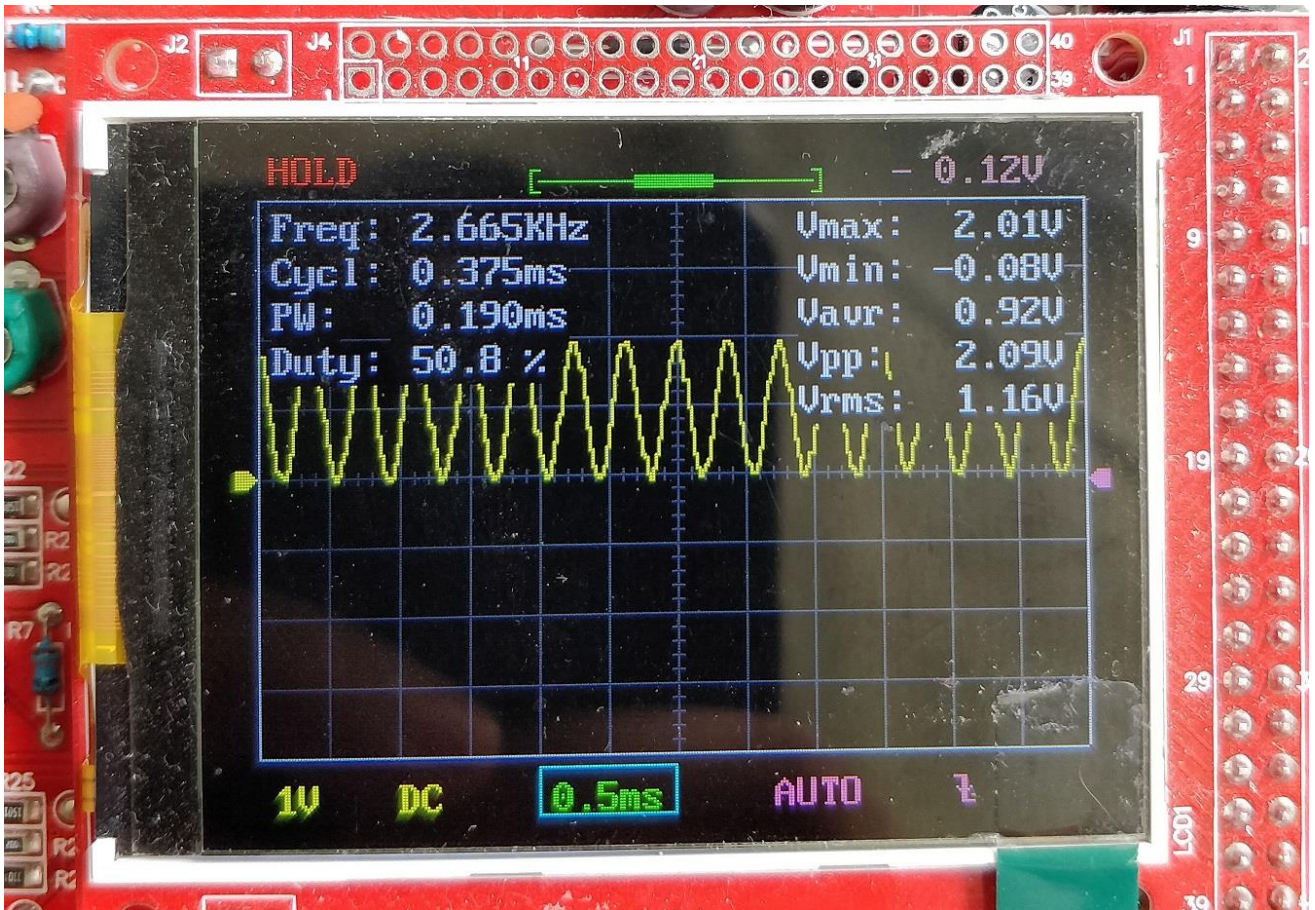


Fig. 8 Output of the DAC

Direct Digital Synthesis using PIO

This program was an introduction to the PIO subsystem on the Raspberry Pi Pico. I used an MCP4822 DAC to generate a sine wave of a given frequency. The Pico transmits data to the DAC using the PIO state machine running a pseudo-SPI system. This program was also a test to see if the existing Lab 1 in the course ECE 4760 replicable using the RP2040 microcontroller.

It is important to note that the data in the previous implementation was transferred via an SPI channel while the data in this program is transferred via the PIO subsystem. The PIO subsystem created was derived heavily from the SPI PIO example provided by the RaspberryPi foundation with a few modifications to fit my requirements.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the standard math header(math.h). The others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries.
The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/gpio.h, hardware/adc.h, hardware/irq.h, hardware/pwm.h, hardware/pio.h, pico/time.h and PIODDS.pio.h
2. The next section of the code is the #define's and the global variables which will be used throughout the code.

The following are the #define's to be used throughout the code:

- Chip Select pin
- Master Out Slave In pin
- The Serial Clock pin
- The number of elements in the sine table
- The sampling frequency for the generated signal
- Pre-calculated value for $2^{32}/F_s$
- The configuration bits (to be used as a mask) for the DAC

The following are the variables to be used throughout the code:

- The Phase Accumulator variable to keep a track of the "angle" of the phasor
- The Phase Incrementor variable to keep a track of the amount to be added to the accumulator
- The ADC input
- The modified DAC data to be sent
- The sine lookup table to contain the amplitudes for a single period of a sine wave
- The PIO instance
- The state machine to be attached to the PIO instance
- The PIO SPI instance structure

3. To transmit the data to the DAC using SPI, I used the `__time_critical_func` decorator to execute the `pio_spi_write16_blocking()` function from RAM.
4. In order to get a constant sampling frequency of 44kHz, I used an interrupt which triggers every $23\mu\text{s}$ ($1 / 44000$). In order to handle the interrupt, I created an interrupt handler which does the following:
 - Reads the ADC value in the variable `adcIn`
 - Calculate the phase incrementer. The frequency to be generated is the input from the ADC (0 - 4095Hz)
 - Calculate the phase accumulator
 - Calculate the DAC data. It is the data from the sine table indexed using the 8 MSBs
 - Mask the configurations bits on the DAC data
 - Write the DAC data to the PIO SPI port.
5. The first line in `main()` is a call to `stdio_init_all()`. This function initializes `stdio` to communicate through either UART or USB, depending on the configurations in the `CMakeLists.txt` file.
6. Initialize a repeating timer called `timer` to use it to trigger the interrupts.
7. Initialize a 256-element wide sine table in order to contain the amplitudes for a single period of a sine wave.
8. In order to use the ADC, I first initialised the ADC using the `adc_init()` function. I then initialized the GPIO 26 using the `adc_gpio_init()` function. From the datasheet, I know that the ADC inputs 0 to 3 are connected to GPIOs 26 to 29 respectively. In order to select the input, I used the `adc_select_input()` function.
9. In order to initialise the chip select pin, I used the `gpio_init()`. Next, I used the `gpio_set_dir()` function to set the direction of the chip select pin as output. Lastly, I used the `gpio_put()` function to drive the chip select pin high.
10. In order to initialize the interrupt on the timer, I used the `add_repeating_timer_us()` function. It takes the following parameters as arguments:
 - The delay in microseconds: The first argument is the delay in microseconds. If the delay is positive, then this is the delay between one callback ending and the next starting. If the delay is negative, then this is the time between the start of two callbacks.
 - The callback function: This is the function that will be called as soon as the interrupt occurs. This is the interrupt handler, so to speak.
 - The user data: This is the user data to pass to store in the `repeating_timer` structure for use by the callback.
 - The pointer to timer: This is the pointer to the user owned structure to store the repeating timer info in.
11. The last part of the program is the infinite while loop. The infinite while loop for this program is an empty while loop which serves only to keep the core running and not let it exit.

In order to view the output of the DAC, I used an oscilloscope. As it is quite evident from the oscilloscope output, the output of the DAC is a sine wave of the desired frequency.

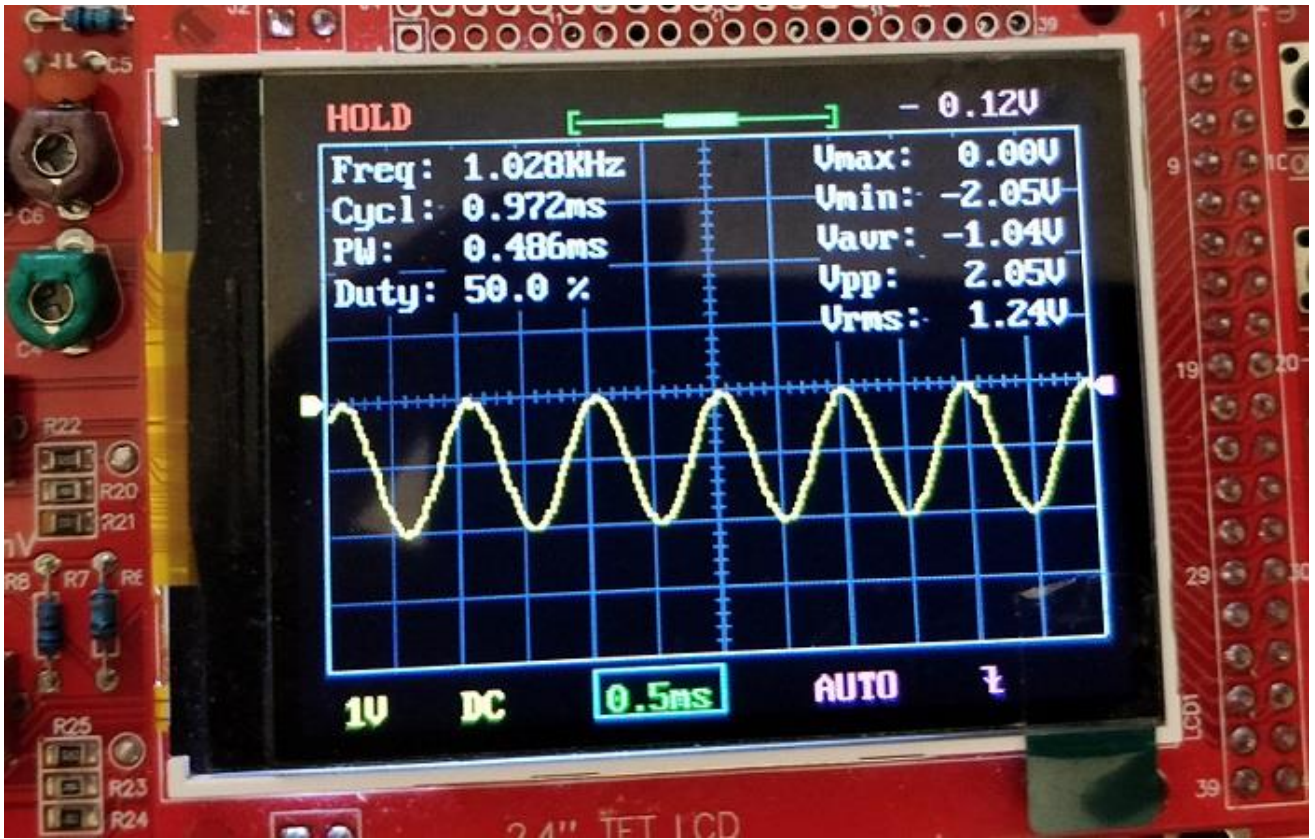


Fig. 9 Output of the DAC

Dual Core System

RP2040 has Dual Cortex M0+ processor cores which can run up to 133MHz independent of each other. However, the second core (core 1) is asleep on boot and needs to be woken up by a function call from the first core (core 0). This program blinks 2 LEDs by controlling them separately from each core.

Note: To start core 1, it has to be launched from core 0 at startup.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries.
The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/gpio.h, pico/time.h and pico/multicore.h. As the names suggest, these interface libraries give us access to the API's associated with the hardware GPIO, pico time and pico multicore on the RP2040.
2. The next section of the code is basically two #define's which define the GPIO pins for the LEDs.
3. The core 1 function is the function which runs on the core 1 once it wakes up from its slumber. In other terms, this function is the main() function for core 1 and runs independent of the actual main() function running on core 0 (unless there is an intra-core communication). The core1_entry() function initializes the LED2 pin and configures it to be the output pin. The gpio_init() function is used to initialize the pin and the gpio_set_dir() function is used to set the pin direction which can be GPIO_OUT (output) or GPIO_IN (input). Then, in an infinite while loop, it turns the LED on and off at regular intervals using the gpio_put() function and sleeps for 300 milliseconds using the sleep_ms() function. Note: This will only put core 1 to sleep and not core 0.
4. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
5. In the next 2 lines of the code, I initialized the LED1 pin and configured it to be the output pin. The gpio_init() function is used to initialize the pin and the gpio_set_dir() function is used to set the pin direction which can be GPIO_OUT (output) or GPIO_IN (input).
6. In order to wake up the core 1 from sleep, I used the multicore_launch_core1() function. This function resets core 1 and enters the given function on core 1 using the default core 1 stack (below core 0 stack).
7. The infinite while loop is the loop which runs forever and executes the code sequentially. It basically contains 2 subsections: turning the LED1 on and turning the LED1 off. I also used the printf() statement to print the output to the screen. In order to see the output, I used the serial monitor provided by the Arduino IDE. Then I used gpio_put() to set the pins HIGH or LOW. Lastly, I used the sleep_ms() to put the CPU to sleep for 1 second for both HIGH and LOW. This infinite while loop runs in parallel with the infinite while loop on core 1.

In order to view the output, I used the serial monitor provided by the Arduino IDE. As it shows, the two LEDs are toggling simultaneously.

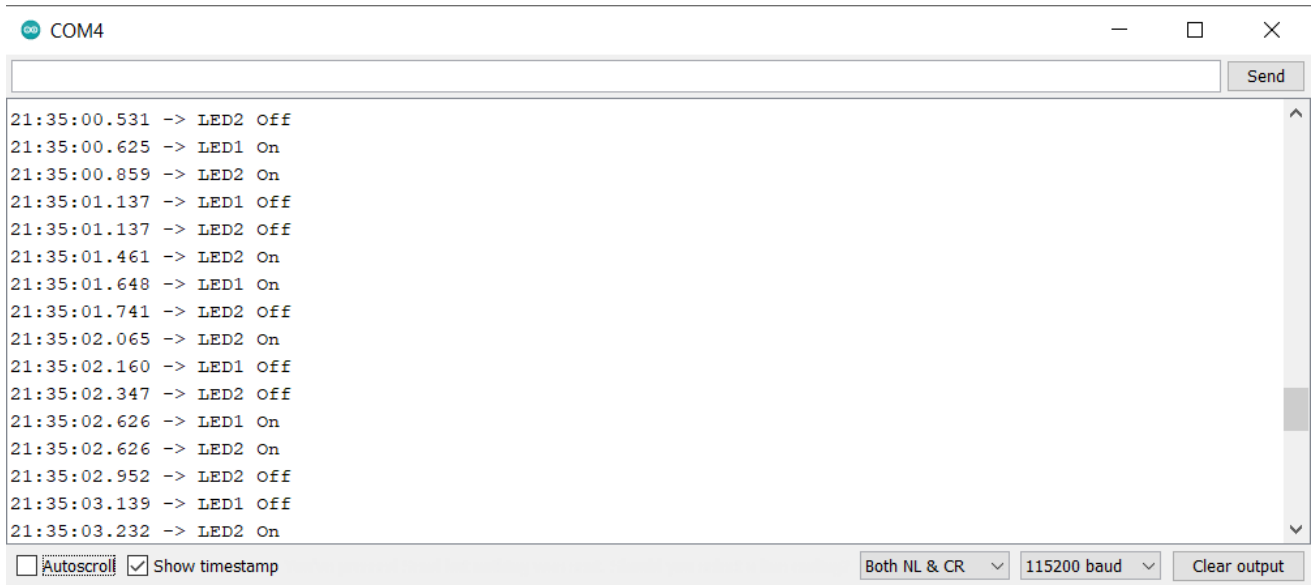


Fig. 10 Output of the dual core test program

Multi Core Performance Compare

RP2040 has Dual Cortex M0+ processor cores which can run up to 133MHz independent of each other. However, the second core (core 1) is asleep on boot and needs to be woken up by a function call from the first core (core 0). This program compares the performance of the two cores by incrementing two different variables on each core.

Note: To start core 1, it has to be launched from core 0 at startup.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries. The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/gpio.h, pico/time.h and pico/multicore.h. As the names suggest, these interface libraries give us access to the API's associated with the hardware GPIO, pico time and pico multicore on the RP2040.
2. The next section of the code is the #define's and the global variables which will be used throughout the code. The #define is the pushbutton pin declaration on GPIO 5. The variables declared are i and j which are used as the incrementers to test the performance of the cores.
3. The core 1 function is the function which runs on the core 1 once it wakes up from its slumber. In other terms, this function is the main() function for core 1 and runs independent of the actual main() function running on core 0 (unless there is an intra-core communication). The core1_entry() function follows the following algorithm:
 - Fetch the start time.
 - Print out the start time.
 - If j is less than 100000000, increment j.
 - If j is equal to 100000000, calculate the time taken by core 1 to finish the job and print it out. Increment j to avoid multiple prints.
 - If j is greater than 100000000 and the button has been pushed, reset j and the start time to start the job again.
4. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
5. In the next 2 lines of the code, I initialized the button pin and configured it to be the input pin. The gpio_init() function is used to initialize the pin and the gpio_set_dir() function is used to set the pin direction which can be GPIO_OUT (output) or GPIO_IN (input).
6. In order to wake up the core 1 from sleep, I used the multicore_launch_core1() function. This function resets core 1 and enters the given function on core 1 using the default core 1 stack (below core 0 stack).
7. The infinite loop is quite similar to the core1_entry() function. It runs on the core 0 and follows the following algorithm:
 - Fetch the start time.
 - Print out the start time.

- If i is less than 100000000, increment i .
- If i is equal to 100000000, calculate the time taken by core 0 to finish the job and print it out. Increment i to avoid multiple prints.
- If i is greater than 100000000 and the button has been pushed, reset i and the start time to start the job again.

In order to view the output, I used the serial monitor provided by the Arduino IDE. As it is quite evident from the provided data, core 0 takes an average of 90 milliseconds more than core 1 to finish the same job.

Table 1 Performance comparison of the two cores

Iteration	Time taken by Core 0	Time taken by Core 1
1	7779846	7690696
2	7791554	7697507
3	7781144	7692173
4	7791770	7702660
Average	7786078	7695759

```

COM4
21:59:11.845 -> Core 0 Start Time: 10001627.
21:59:11.845 -> Core 1 Start Time: 10001665.
21:59:19.548 -> Core 1 Time Taken: 7690696.
21:59:19.641 -> Core 0 Time Taken: 7779846.
21:59:46.737 -> Core 1 Start Time: 44880989.
21:59:46.737 -> Core 0 Start Time: 44880989.
21:59:54.425 -> Core 1 Time Taken: 7697507.
21:59:54.519 -> Core 0 Time Taken: 7791554.
22:00:15.801 -> Core 0 Start Time: 73975331.
22:00:15.801 -> Core 1 Start Time: 73975331.
22:00:23.526 -> Core 1 Time Taken: 7692173.
22:00:23.574 -> Core 0 Time Taken: 7781144.
22:00:33.022 -> Core 0 Start Time: 91195413.
22:00:33.022 -> Core 1 Start Time: 91195413.
22:00:40.748 -> Core 1 Time Taken: 7702660.
22:00:40.842 -> Core 0 Time Taken: 7791770.
22:00:43.779 -> Core 0 Start Time: 101940673.
22:00:43.779 -> Core 1 Start Time: 101940673.
22:00:51.466 -> Core 1 Time Taken: 7690458.
22:00:51.559 -> Core 0 Time Taken: 7779999.
  
```

Fig. 11 Output of the Multicore Performance Comparison

Multi Core Contention

RP2040 has Dual Cortex M0+ processor cores which can run up to 133MHz independent of each other. However, the second core (core 1) is asleep on boot and needs to be woken up by a function call from the first core (core 0). This program demonstrates the result of any arising contention between core 0 and core 1 for a shared memory space.

Note: To start core 1, it has to be launched from core 0 at startup.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries. The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/gpio.h, hardware/timer.h, pico/time.h and pico/multicore.h. As the names suggest, these interface libraries give us access to the API's associated with the hardware GPIO, hardware timer, pico time and pico multicore on the RP2040.
2. The next section of the code is the #define's and the global variables which will be used throughout the code. The #define is the pushbutton pin declaration on GPIO 5. The variables declared are the increment variable i and the tracker variables core0 and core1. The increment variable i is shared by both cores as it is declared in the global memory space.
3. The core 1 function is the function which runs on the core 1 once it wakes up from its slumber. In other terms, this function is the main() function for core 1 and runs independent of the actual main() function running on core 0 (unless there is an intra-core communication). The core1_entry() function checks if i is less than 100000000. If it is, it increments both i and core1 and repeats the process for eternity.
4. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
5. In the next 2 lines of the code, I initialized the button pin and configured it to be the input pin. The gpio_init() function is used to initialize the pin and the gpio_set_dir() function is used to set the pin direction which can be GPIO_OUT (output) or GPIO_IN (input).
6. In order to wake up the core 1 from sleep, I used the multicore_launch_core1() function. This function resets core 1 and enters the given function on core 1 using the default core 1 stack (below core 0 stack).
7. The infinite loop is quite similar to the core1_entry() function. It runs on the core 0 and follows the following algorithm:
 - Fetch the start time.
 - Print out the start time.
 - If i is less than 100000000, increment i and core0 tracker.
 - If i is equal to 100000000, calculate the time taken by core 0 to finish the job and print it out. Also, print out the values of core0 and core1 tracker variables. Increment i to avoid multiple prints.

- If i is greater than 100000000 and the button has been pushed, reset i and the tracker variables and the start time to start the job again.

In order to view the output, I used the serial monitor provided by the Arduino IDE. As it is quite evident from the provided data, core 0 takes priority over core 1 when both the cores try to simultaneously access a part of the shared memory. This is why, all the increments take place in core 0.

```
15:37:23.209 -> Start Time: 14530281.  
15:37:32.920 -> Time Taken: 9722711.  
15:37:32.920 -> Core0: 100000000, core1: 0  
15:37:41.474 -> Start Time: 32794643.  
15:37:51.183 -> Time Taken: 9722712.  
15:37:51.183 -> Core0: 100000000, core1: 0  
15:37:53.640 -> Start Time: 44987104.  
15:38:03.389 -> Time Taken: 9722631.  
15:38:03.389 -> Core0: 100000000, core1: 0  
15:38:49.250 -> Start Time: 100556926.  
15:38:58.948 -> Time Taken: 9722663.  
15:38:58.948 -> Core0: 100000000, core1: 0
```

Fig. 12 Output of the Multicore Contention

Multi Core Contention Prevention

RP2040 has Dual Cortex M0+ processor cores which can run upto 133MHz independent of each other. However, the second core (core 1) is asleep on boot and needs to be woken up by a function call from the first core (core 0). This program demonstrates the prevention of contention between core 0 and core 1 for a shared memory space using spin locks.

Note: To start core 1, it has to be launched from core 0 at startup.

As per Wikipedia: A spinlock is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (the one which holds the lock) blocks, or "goes to sleep".

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. One of these is standard C headers (stdio.h) and the others are headers which come from the C SDK for the Raspberry Pi Pico. The first of these, pico/stdlib.h is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries. The next includes pull in hardware APIs which are not already brought in by pico/stdlib.h. These include hardware/gpio.h, hardware/timer.h, hardware/sync.h, pico/time.h and pico/multicore.h. As the names suggest, these interface libraries give us access to the API's associated with the hardware GPIO, hardware timer, hardware sync, pico time and pico multicore on the RP2040.
2. The next section of the code is the #define's and the global variables which will be used throughout the code. The #define is the pushbutton pin declaration on GPIO 5. The variables declared include a spin lock number and a spin lock identifier. The next variables that are declared are the increment variable i and the tracker variables core0 and core1. The increment variable i is shared by both cores as it is declared in the global memory space.
3. The core 1 function is the function which runs on the core 1 once it wakes up from its slumber. In other terms, this function is the main() function for core 1 and runs independent of the actual main() function running on core 0 (unless there is an intra-core communication). The core1_entry() function grabs the spin lock (if it is unlocked) and checks if i is less than 100000000. If it is, it increments both i and core1 and releases the spinlock. It then proceeds to repeat the process for eternity. As long as the spin lock is acquired by core 1, core 0 will not be able to acquire it.
4. The first line in main() is a call to stdio_init_all(). This function initializes stdio to communicate through either UART or USB, depending on the configurations in the CMakeLists.txt file.
5. In the next 2 lines of the code, I initialized the button pin and configured it to be the input pin. The gpio_init() function is used to initialize the pin and the gpio_set_dir() function is used to set the pin direction which can be GPIO_OUT (output) or GPIO_IN (input).
6. The next line of code claims an unused spin lock using the spin_lock_claim_unused() function. This function takes a boolean value as an argument. If the argument is true, the function will

panic if no spin locks are available. Then, this spin lock is initialized using the `spin_lock_init()` function.

7. In order to wake up the core 1 from sleep, I used the `multicore_launch_core1()` function. This function resets core 1 and enters the given function on core 1 using the default core 1 stack (below core 0 stack).
8. The infinite loop is quite similar to the `core1_entry()` function. It runs on the core 0 and follows the following algorithm:
 - Fetch the start time.
 - Print out the start time.
 - If the spin lock is unclaimed, grab it. Else, wait for it to become available.
 - If `i` is less than 100000000, increment `i` and core0 tracker.
 - If `i` is equal to 100000000, calculate the time taken by core 0 to finish the job and print it out. Also, print out the values of core0 and core1 tracker variables. Increment `i` to avoid multiple prints.
 - If `i` is greater than 100000000 and the button has been pushed, reset `i` and the tracker variables and the start time to start the job again.
 - Release the spin lock.

In order to view the output, I used the serial monitor provided by the Arduino IDE. As it is quite evident from the provided data, implementing the spin lock avoids the contention for shared memory between both cores and allows safe access of data.

```
00:02:51.455 -> Start Time: 27108882.
00:03:12.698 -> Time Taken: 21240797.
00:03:12.698 -> Core0: 49577245, core1: 50422755
00:09:23.969 -> Start Time: 419622568.
00:09:45.216 -> Time Taken: 21239187.
00:09:45.216 -> Core0: 49577850, core1: 50422150
00:09:46.187 -> Start Time: 441832065.
00:10:07.436 -> Time Taken: 21250296.
00:10:07.436 -> Core0: 49576668, core1: 50423332
00:10:14.741 -> Start Time: 470406552.
00:10:36.006 -> Time Taken: 21246024.
00:10:36.006 -> Core0: 49577044, core1: 50422956
```

Fig. 13 Output of the Spin Lock Mechanism

Hello DMA

This program was an introduction to the Direct Memory Access (DMA) on the Raspberry Pi Pico. The RP2040 DMA controller has separate read and write master connections to the bus fabric, and performs bulk data transfers on a processor's behalf. This leaves processors free to attend to other tasks, or enter low-power sleep states. The data throughput of the DMA is also significantly higher than one of RP2040's processors. The DMA can perform one read access and one write access, up to 32 bits in size, every clock cycle. There are 12 independent channels, each which supervise a sequence of bus transfers. It is based on the hello DMA code provided in the pico examples by the RaspberryPi foundation.

The basic structure of the program looks as follows:

1. The first lines of code in the C source file include some header files. The headers which come from the C SDK for the Raspberry Pi Pico include `pico/stdlib.h` is what the SDK calls a "High-Level API." These high-level API's "provide higher level functionality that isn't hardware related or provides a richer set of functionalities above the basic hardware interfaces." The architecture of this SDK is described at length in the SDK manual. All libraries within the SDK are INTERFACE libraries.

The next includes pull in hardware APIs which are not already brought in by `pico/stdlib.h`. These include `hardware/dma.h` and `pico/time.h`. As the names suggest, these interface libraries give us access to the API's associated with the hardware dma and the pico time on the RP2040.

2. The next section of the code is the `#define`'s and the global variables which will be used throughout the code.

The following are the `#define`'s to be used throughout the code:

- The number of elements in the sine table

3. The following are the variables to be used throughout the code:

- The source array
- The destination array

4. The first line in `main()` is a call to `stdio_init_all()`. This function initializes `stdio` to communicate through either UART or USB, depending on the configurations in the `CMakeLists.txt` file.
5. In order to fetch an unused DMA channel, I used the `dma_claim_unused_channel()`.
6. The aim is to copy the contents from one memory location to other. Hence, I initialized a source array using a for loop.
7. Finally, I configured the DMA channel using a series of commands. First, I used the `dma_channel_get_default_config()` function to fetch the default configurations. Next, I used the `channel_config_set_transfer_data_size()` function to set the transfer size to 32 bits. Lastly, I used the `channel_config_set_read_increment()` and `channel_config_set_write_increment()` to set the read and write to increment the address after each transfer.

Once the configurations are set, I used the `dma_channel_configure()` function to set the destination, the source and the size of the transfer. The `dma_channel_wait_for_finish_blocking()` function waits until the DMA channel is done transferring.

8. To verify the result of the DMA transfer, I printed the contents of the destination array on the console using a for loop.

The output of the DMA transfer is shown below. As it is quite evident, the DMA transfer was successful.

```
11:37:25.805 -> Source[0]: 0          11:37:26.181 -> Destination[0]: 0
11:37:25.805 -> Source[1]: 1          11:37:26.181 -> Destination[1]: 1
11:37:25.805 -> Source[2]: 2          11:37:26.181 -> Destination[2]: 2
11:37:25.805 -> Source[3]: 3          11:37:26.181 -> Destination[3]: 3
11:37:25.805 -> Source[4]: 4          11:37:26.181 -> Destination[4]: 4
11:37:25.805 -> Source[5]: 5          11:37:26.181 -> Destination[5]: 5
11:37:25.805 -> Source[6]: 6          11:37:26.181 -> Destination[6]: 6
11:37:25.805 -> Source[7]: 7          11:37:26.181 -> Destination[7]: 7
11:37:25.805 -> Source[8]: 8          11:37:26.181 -> Destination[8]: 8
11:37:25.805 -> Source[9]: 9          11:37:26.181 -> Destination[9]: 9
11:37:25.805 -> Source[10]: 10         11:37:26.181 -> Destination[10]: 10
11:37:25.805 -> Source[11]: 11        11:37:26.181 -> Destination[11]: 11
11:37:25.805 -> Source[12]: 12        11:37:26.181 -> Destination[12]: 12
11:37:25.805 -> Source[13]: 13        11:37:26.181 -> Destination[13]: 13
11:37:25.805 -> Source[14]: 14        11:37:26.181 -> Destination[14]: 14
11:37:25.805 -> Source[15]: 15        11:37:26.214 -> Destination[15]: 15
11:37:25.805 -> Source[16]: 16        11:37:26.214 -> Destination[16]: 16
```

Fig. 14 Output of the Hello DMA code

VGA Screen Testing

At this point in my MEng program, Prof. Hunter Adams had created a VGA library to interface a VGA screen with the RaspberryPi Pico. I tested out various programs using the library provided by him.

Google Dino Game

This program was my attempt to port my Google Dino game to the RaspberryPi Pico using Prof. Hunter Adam's VGA Library for the RaspberryPi Pico. I created a dino game on the RaspberryPi Pico and implemented an additional function to draw bitmaps. The game is controlled using a pushbutton attached to a GPIO pin.

The code has a VGA library which has been explained above using Prof. Adams webpage, the Bitmap header file and the main file. I also created a custom remote controller connected to the GPIO pins to control the Dino.

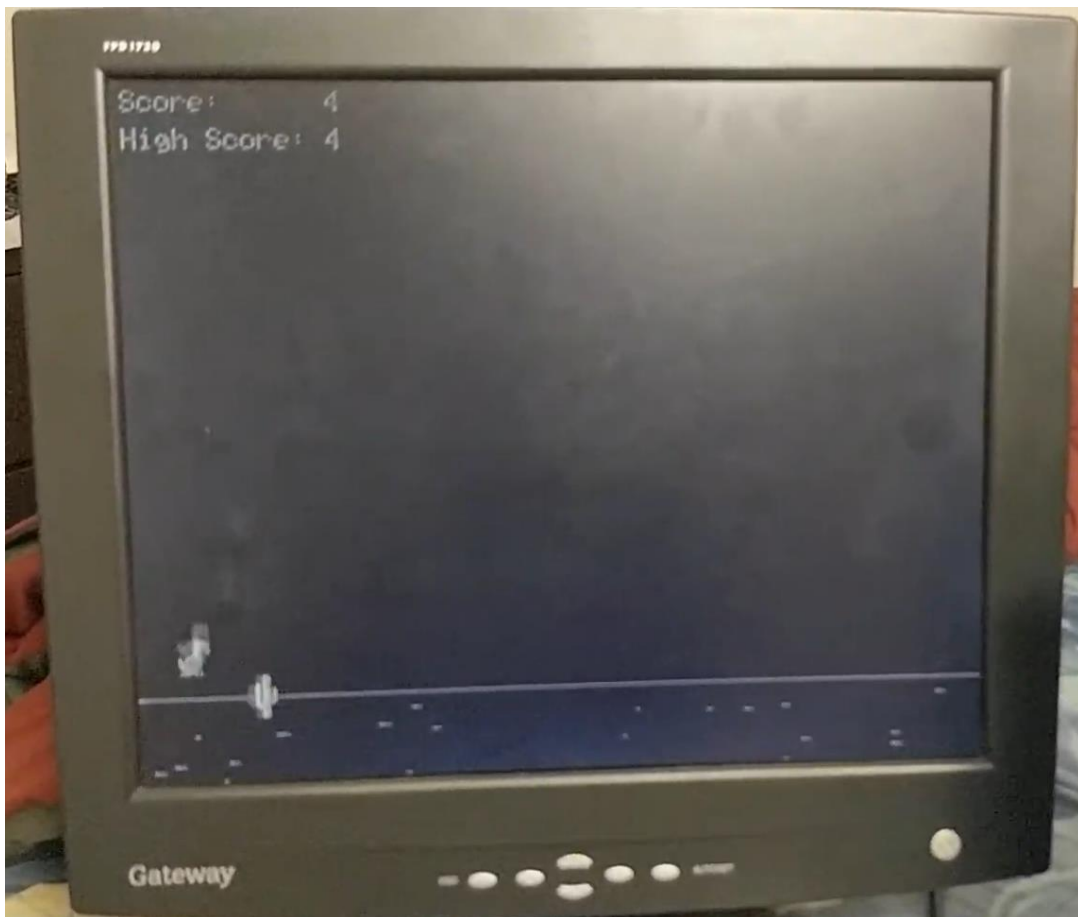


Fig. 15 Dino game on RaspberryPi Pico

Snake Game

This program was my attempt to recreate the Snake game on the RaspberryPi Pico using Prof. Hunter Adam's VGA Library for the RaspberryPi Pico. The game is controlled using pushbuttons attached to 4 different GPIO pins (one for each direction the snake can move in).

The code has a VGA library which has been explained above using Prof. Adams webpage and the main file. I also created a custom remote controller connected to the GPIO pins to control the snake.

1. First and foremost, the snake is implemented in the form of a linked list. According to Wikipedia, a linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence. In its most basic form, each node contains: data, and a reference (in other words, a link) to the next node in the sequence. Each node in our snake linked list holds the x-coordinate of the node, the y-coordinate of the node and the pointer to the next node. Therefore, moving the snake is nothing more than shifting the data of the nodes of the linked list.
2. The function to generate food is pretty simple. First, I randomly generate x and y coordinates for the food. Then, I iterate through the linked list. If the food coordinates coincide with the snake's body, I regenerate the food coordinates.
3. Two functions which are responsible for motion are `move()` and `eatAndMove()`. These functions are responsible for the normal motion and the motion of the snake respectively. They're quite similar, except for the fact that `eatAndMove()` increases the length of the snake by adding a new element at the end of the linked list. Both these functions work by shifting the data in the linked list.

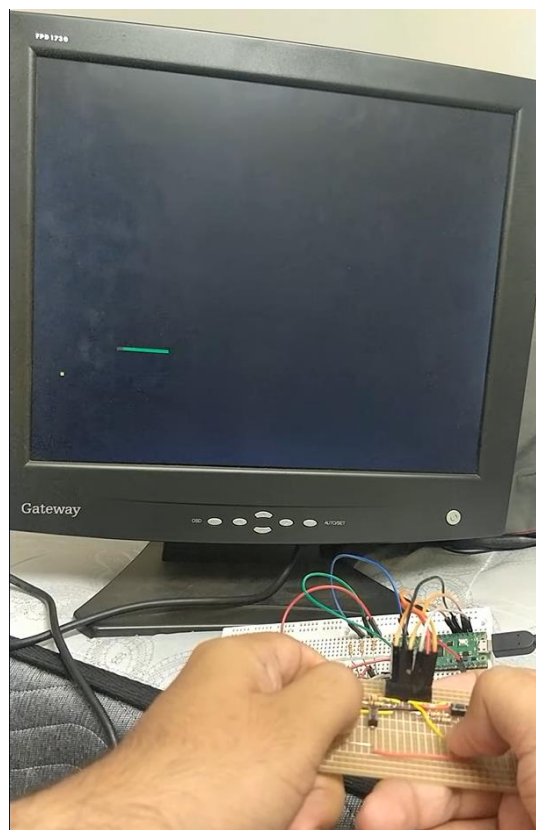


Fig. 16 Snake game and the controller on RaspberryPi Pico

SinCosTan Equation

This program was my attempt to plot a given equation on a VGA screen using Prof. Hunter Adam's VGA Library for the RaspberryPi Pico.

I chose the following fine detail equation to plot on the VGA screen:

$$\sin(\cos(\tan(xy))) = \sin(\cos(\tan(x))) + \sin(\cos(\tan(y)))$$

Using desmos, the plotted equation looks as shown.

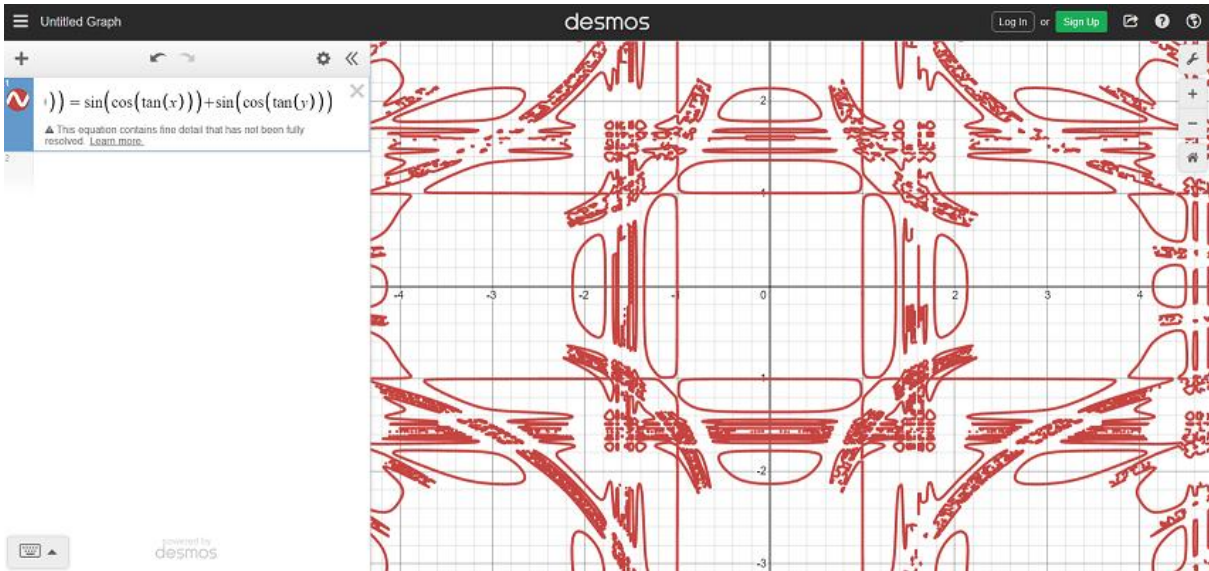


Fig. 17 The plot of the equation on Desmos

The code has a VGA library which has been explained above using Prof. Adams webpage and the main file. The main crux of the program lies in the fact that I needed to compute the equation within a given error as trying to find the exact solution can be really computationally expensive. Therefore, I divided the x-coordinates and the y-coordinates into a series of small steps and computed the equation on using those coordinates. If the answer is within the given error limit, draw a red circle at the given location.



Fig. 18 Plotting the equation on a VGA screen using RaspberryPi Pico

The Heart Equation

This program was my attempt to plot a given equation on a VGA screen using Prof. Hunter Adam's VGA Library for the RaspberryPi Pico.

I chose the following equation to plot on the VGA screen:

$$\left(\sin\left(a \times \frac{\pi}{10}\right) + x\right)^2 + \left(\cos\left(a \times \frac{\pi}{10}\right) + y\right)^2 = 0.7|x|y + 1$$

Here, different values of a give different graphs, which when put together give the shape of a heart. Using desmos, the plotted equation looks as shown.

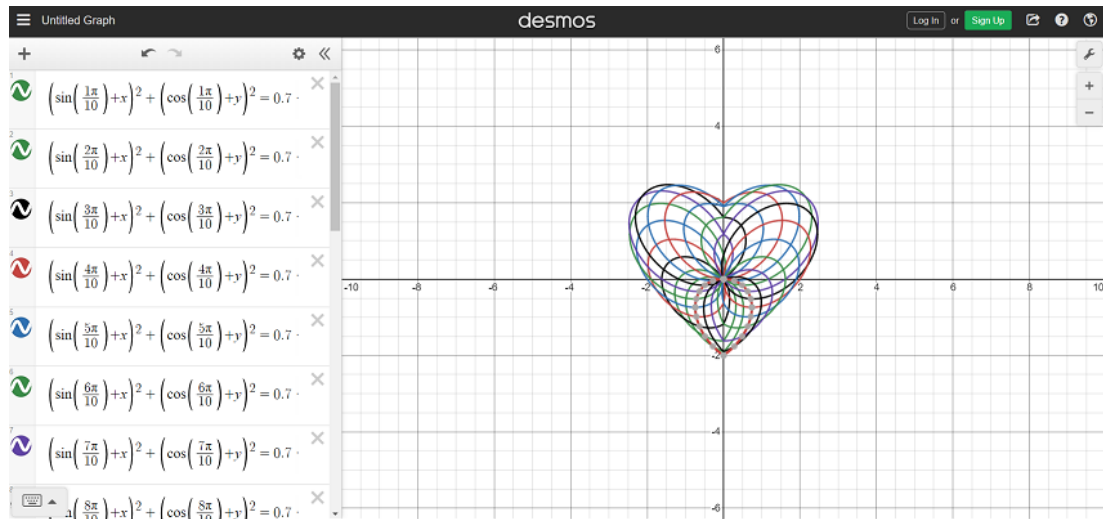


Fig. 19 The plot of the equation on Desmos

The code has a VGA library which has been explained above using Prof. Adams webpage and the main file. The main crux of the program lies in the fact that I needed to compute the equation within a given error as trying to find the exact solution can be really computationally expensive. Therefore, I divided the x-coordinates and the y-coordinates into a series of small steps and computed the equation on using those coordinates. If the answer is within the given error limit, draw a red circle at the given location.



Fig. 20 Plotting the equation on a VGA screen using RaspberryPi Pico

TFT Library Creation

This program was an implementation of the PIC32 TFT library for the Raspberry Pi Pico. The TFT display is interfaced using SPI. However, I used PIO to create my own SPI channel running at 31.25MHz to send data to the TFT display. The PIO driver code is pretty similar to the PIO DDS program except for a few modifications which have been addressed in the subsequent sections. The code for state machine also uses the SPI PIO example provided by the RaspberryPi foundations with some heavy modifications. The driver code uses a state machine to transmit 8 bits at a time using SPI protocol. At the end of each transaction, the state machine sets an interrupt flag that allows the CPU to register that the transaction is complete. Until the transaction is completed, the CPU is stalled by a while() loop.

Note: The fundamental difference between my program and the program for PIC32 is that instead of transmitting 16-bit words, I am transmitting 2 8-bit words whenever needed.

Testing

The output below shows output of the rectangle test program. As it is evident, the screen is constantly being redrawn with rectangles of different colours.



Fig. 21 Testing the TFT Library with rectangles

Next, I created a program to test the dino game I created for the TFT screen. The output of the program looks as follows.

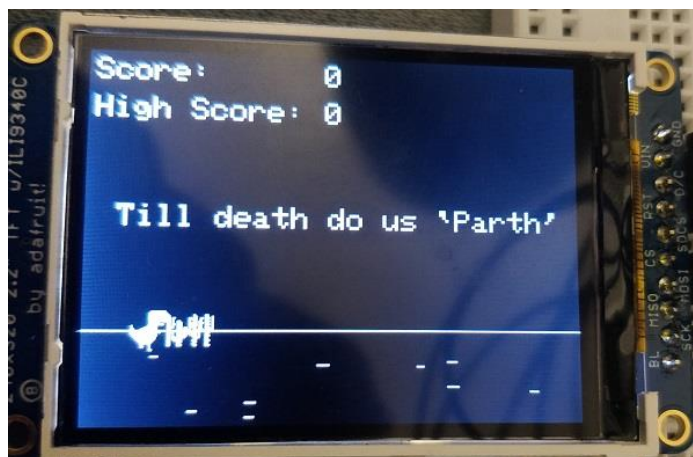


Fig. 22 Testing the TFT Library with Dino game

After the dino game, I tested the bouncing ball code from the ECE 4760 website and its output is shown below.



Fig. 23 Testing the TFT Library with bouncing ball code

For the next program, I implemented the boids algorithm from the ECE 4760 website. The program running on a single core can support roughly 250 boids while the program running on 2 cores can support roughly 315 boids.



Fig. 24 Testing the TFT Library with boids algorithm

Fun fact: While testing the boids algorithm, I found out that the algorithm used in the lab (the one I used it test as shown above) is slightly broken. It allows a few boids to 'leak' out of the screen. This is because as the turn factor tries to slow down the boids, the minimum speed check kicks in and speeds it up back again. This causes the boid to keep on going away from the screen forever. In order to prevent this, I simply changed the turning algorithm. I computed the existing angle, changed the angle based on the trajectory of the boid and computed the x-component and the y-component of the boid based on the new speed using the following equations:

$$vy_{new} = speed \times \sin(\text{angle}_{new})$$

$$vx_{new} = speed \times \cos(\text{angle}_{new})$$

Finally, I studied about fractals using the [Nature of Code](#) series by Daniel Shiffman as a starting point. According to Nature of Code's fractal page, “the term fractal (from the Latin fractus, meaning “broken”) was coined by the mathematician Benoit Mandelbrot in 1975. In his seminal work “The Fractal Geometry of Nature,” he defines a fractal as “a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole.” In short, a fractal is a shape which when divided into various parts, each part can represent the figure as a whole.”

In order to start working with fractals, I also had to study a bit about recursion. The repeated application of a rule to successive results is known as recursion. One of the most famous applications for recursion is the calculation of factorial. The factorial of a natural number is defined as:

$$n! = n \times (n - 1)!$$

$$0! = 1$$

For instance, solving for 5! looks like:

$$5! = 5 \times 4!$$

$$5! = 5 \times 4 \times 3!$$

$$5! = 5 \times 4 \times 3 \times 2!$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$\therefore 5! = 120$$

Note: Recursions must always have a base case and that too at a reasonable depth, else it will cause a stack overflow error.

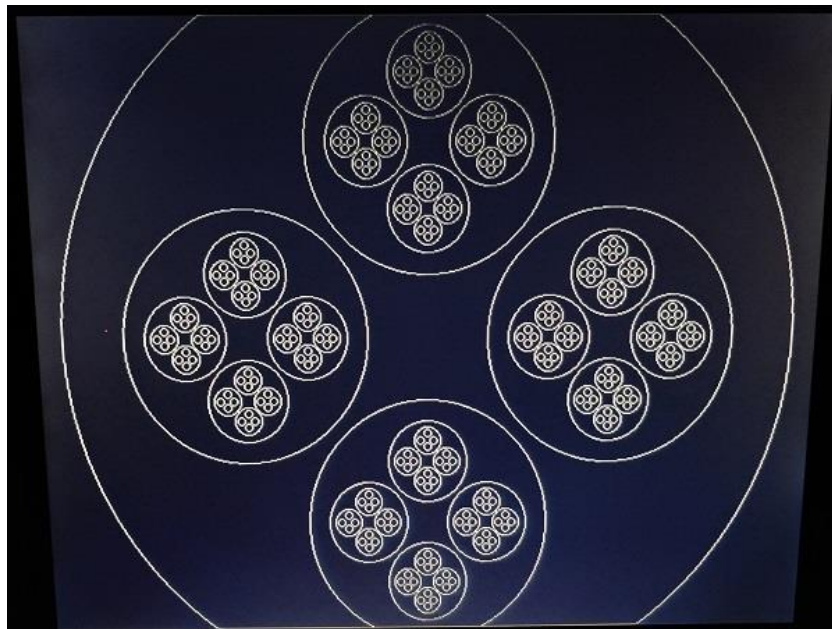


Fig. 25 Output of the circular fractal program



Fig. 26 Output of the line fractal program

Conclusion

I tested the RP2040 microcontroller quite rigorously and found it to be much better than the existing PIC32 system for the course ECE 4760. I also tested out various peripherals and conducted the existing lab work on it and found it to be exceeding the benchmarks of the PIC32. Moreover, I tested out the VGA library created by Prof. Hunter Adams and created by own TFT library for the students to use starting Fall 2022.

Gratitude

I am grateful to Prof. Hunter Adams and Prof. Bruce Land for guiding me through the course of this project, answering all my questions and assisting me with all the tasks that I was assigned. Most of all, I am grateful for the bowl of chocolates in the meeting room which I used to wait eagerly for every Friday for our meetings.

References

1. PicoSetup. (n.d.). Retrieved December 7, 2021, from Github.io website: <https://vha3.github.io/Pico/Setup/PicoSetup.html>
2. Raspberry Pi Documentation. (n.d.). Retrieved December 7, 2021, from Raspberrypi.com website: <https://www.raspberrypi.com/documentation/microcontrollers/>
3. Raspberry Pi Pico SDK: Raspberry Pi Pico SDK. (n.d.). Retrieved December 7, 2021, from Github.io website: <https://raspberrypi.github.io/pico-sdk-doxygen/>
4. (N.d.-a). Retrieved December 7, 2021, from Raspberrypi.com website: <https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf>
5. (N.d.-b). Retrieved December 7, 2021, from Raspberrypi.com website: <https://datasheets.raspberrypi.com/rp2040/hardware-design-with-rp2040.pdf>
6. (N.d.-c). Retrieved December 7, 2021, from Raspberrypi.com website: <https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf>
7. (N.d.-d). Retrieved December 7, 2021, from Raspberrypi.com website: <https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf>
8. (N.d.-e). Retrieved December 7, 2021, from Raspberrypi.com website: <https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-python-sdk.pdf>

Appendix

The following is the code for the TFT Library I created:

TFTMaster.h

```
1.  /* Code rewritten from Adafruit Arduino library for the TFT
2.  *   by Syed Tahmid Mahbub
3.  *   The TFT itself is Adafruit product 1480
4.  *   Included below is the text header from the original Adafruit library
5.  *   followed by the code
6.  */
7.
8.  /*****
9.   This is an Arduino Library for the Adafruit 2.2" SPI display.
10.  This library works with the Adafruit 2.2" TFT Breakout w/SD card
11.  ----> http://www.adafruit.com/products/1480
12.
13.  Check out the links above for our tutorials and wiring diagrams
14.  These displays use SPI to communicate, 4 or 5 pins are required to
15.  interface (RST is optional)
16.  Adafruit invests time and resources providing this open source code,
17.  please support Adafruit and open-source hardware by purchasing
18.  products from Adafruit!
19.
20.  Written by Limor Fried/Ladyada for Adafruit Industries.
21.  MIT license, all text above must be included in any redistribution
22.  *****/
23.
24. /**
25.  * Parth Sarthi Sharma (pss242@cornell.edu)
26.  *
27.  *
28.  * HARDWARE CONNECTIONS
29.  * - GPIO 18 ----> TFT Chip Select
30.  * - GPIO 19 ----> TFT MOSI
31.  * - GPIO 17 ----> TFT SCK
32.  * - GPIO 16 ----> TFT D/C
33.  * - GPIO 20 ----> TFT RST
34.  *
35.  * RESOURCES USED
36.  * - PIO state machines 0 on PIO instance 0
37.  *
38.  * NOTE
39.  * - This is a translation of the display primitives
40.  *   for the PIC32 written by Bruce Land and students
41.  *   with an added bitmaps function.
42.  *
43.  */
44.
45. #ifndef _TFT_MASTER_H_
46. #define _TFT_MASTER_H_
47.
48. #define CS 18
49. #define MOSI 19
50. #define SCK 17
51. #define DC 16
52. #define RST 20
53.
54. #define ILI9340_TFTWIDTH 240
55. #define ILI9340_TFTHEIGHT 320
56.
57. #define ILI9340_NOP 0x00
58. #define ILI9340_SWRESET 0x01
59. #define ILI9340_RDDID 0x04
60. #define ILI9340_RDDST 0x09
```

```
61.
62. #define ILI9340_SLPIN    0x10
63. #define ILI9340_SLPOUT  0x11
64. #define ILI9340_PTLON   0x12
65. #define ILI9340_NORON   0x13
66.
67. #define ILI9340_RDMODE   0x0A
68. #define ILI9340_RDMADCTL 0x0B
69. #define ILI9340_RDPIXFMT 0x0C
70. #define ILI9340_RDIMGFMT 0x0A
71. #define ILI9340_RDSELDIAG 0x0F
72.
73. #define ILI9340_INVOFF   0x20
74. #define ILI9340_INVON   0x21
75. #define ILI9340_GAMMASET 0x26
76. #define ILI9340_DISPOFF 0x28
77. #define ILI9340_DISPON   0x29
78.
79. #define ILI9340_CASET    0x2A
80. #define ILI9340_PASET    0x2B
81. #define ILI9340_RAMWR    0x2C
82. #define ILI9340_RAMRD    0x2E
83.
84. #define ILI9340_PTLAR    0x30
85. #define ILI9340_MADCTL   0x36
86.
87. #define ILI9340_MADCTL_MY 0x80
88. #define ILI9340_MADCTL_MX 0x40
89. #define ILI9340_MADCTL_MV 0x20
90. #define ILI9340_MADCTL_ML 0x10
91. #define ILI9340_MADCTL_RGB 0x00
92. #define ILI9340_MADCTL_BGR 0x08
93. #define ILI9340_MADCTL_MH 0x04
94.
95. #define ILI9340_PIXFMT   0x3A
96.
97. #define ILI9340_FRMCTR1  0xB1
98. #define ILI9340_FRMCTR2 0xB2
99. #define ILI9340_FRMCTR3 0xB3
100. #define ILI9340_INVCTR   0xB4
101. #define ILI9340_DFUNCTR  0xB6
102.
103. #define ILI9340_PWCTR1   0xC0
104. #define ILI9340_PWCTR2   0xC1
105. #define ILI9340_PWCTR3   0xC2
106. #define ILI9340_PWCTR4   0xC3
107. #define ILI9340_PWCTR5   0xC4
108. #define ILI9340_VMCTR1   0xC5
109. #define ILI9340_VMCTR2   0xC7
110.
111. #define ILI9340_RDID1    0xDA
112. #define ILI9340_RDID2    0xDB
113. #define ILI9340_RDID3    0xDC
114. #define ILI9340_RDID4    0xDD
115.
116. #define ILI9340_GMCTRP1  0xE0
117. #define ILI9340_GMCTRN1 0xE1
118.
119. #define ILI9340_PWCTR6   0xFC
120.
121. //Color definitions
122. #define ILI9340_BLACK    0x0000
123. #define ILI9340_BLUE     0x001F
124. #define ILI9340_RED      0xF800
125. #define ILI9340_GREEN    0x07E0
126. #define ILI9340_CYAN    0x07FF
127. #define ILI9340_MAGENTA  0xF81F
128. #define ILI9340_YELLOW   0xFFE0
129. #define ILI9340_WHITE    0xFFFF
130.
131. #define tabspace 4
132.
133. #define swap(a, b) {short t = a; a = b; b = t;}
```

```

134.
135. void tft_init_hw(void);
136. void tft_spiwrite(unsigned char c);
137. void tft_spiwrite8(unsigned char c);
138. void tft_spiwrite16(unsigned short c);
139. void tft_writecommand(unsigned char c);
140. void tft_writecommand16(unsigned short c);
141. void tft_writedata(unsigned char c);
142. void tft_writedata16(unsigned short c);
143. void tft_commandList(unsigned char *addr);
144. void tft_begin(void);
145. void tft_setAddrWindow(unsigned short x0, unsigned short y0, unsigned short x1, unsigned short
y1);
146. void tft_pushColor(unsigned short color);
147. void tft_drawPixel(short x, short y, unsigned short color);
148. void tft_drawFastVLine(short x, short y, short h, unsigned short color);
149. void tft_drawFastHLine(short x, short y, short w, unsigned short color);
150. void tft_fillScreen(unsigned short color);
151. void tft_fillRect(short x, short y, short w, short h, unsigned short color);
152. unsigned short tft_Color565(unsigned char r, unsigned char g, unsigned char b);
153. void tft_setRotation(unsigned char m);
154. void tft_drawLine(short x0, short y0, short x1, short y1, unsigned short color);
155. void tft_drawRect(short x, short y, short w, short h, unsigned short color);
156. void tft_drawCircle(short x0, short y0, short r, unsigned short color);
157. void tft_drawCircleHelper(short x0, short y0, short r, unsigned char cornername, unsigned
short color);
158. void tft_fillCircle(short x0, short y0, short r, unsigned short color);
159. void tft_fillCircleHelper(short x0, short y0, short r, unsigned char cornername, short delta,
unsigned short color);
160. void tft_drawTriangle(short x0, short y0, short x1, short y1, short x2, short y2, unsigned
short color);
161. void tft_fillTriangle(short x0, short y0, short x1, short y1, short x2, short y2, unsigned
short color);
162. void tft_drawRoundRect(short x0, short y0, short w, short h, short radius, unsigned short
color);
163. void tft_fillRoundRect(short x0, short y0, short w, short h, short radius, unsigned short
color);
164. void tft_drawBitmap(short x, short y, const unsigned char *bitmap, short w, short h, unsigned
short color);
165. void tft_drawChar(short x, short y, unsigned char c, unsigned short color, unsigned short bg,
unsigned char size);
166. void tft_setCursor(short x, short y);
167. void tft_setTextColor(unsigned short c);
168. void tft_setTextColor2(unsigned short c, unsigned short bg);
169. void tft_setTextSize(unsigned char s);
170. void tft_setTextWrap(char w);
171. void tft_gfx_setRotation(unsigned char r);
172. void tft_write(unsigned char c);
173. void tft_writeString(char* str);
174.
175. #endif
176.
177.
178.

```

TFTMaster.c

```

1.  /* Code rewritten from Adafruit Arduino library for the TFT
2.  *   by Syed Tahmid Mahbub
3.  *   The TFT itself is Adafruit product 1480
4.  *   Included below is the text header from the original Adafruit library
5.  *   followed by the code
6.  */
7.
8.  /*
9.  This is the core graphics library for all our displays, providing a common
10. set of graphics primitives (points, lines, circles, etc.). It needs to be

```



```

83.     uint cs_pin; //Chip select
84. } pio_spi_inst_t;
85.
86. PIO pio = pio0; //Identifier for the first (PIO 0) hardware PIO instance
87. uint sm = 0; //The state machine
88.
89. pio_spi_inst_t spi = { //The SPI instance
90.     .pio = pio0,
91.     .sm = 0,
92.     .cs_pin = CS
93. };
94.
95. uint offset; //Offset for the program to load
96.
97. volatile char flag = 1; //flag to mark completion of an SPI transaction
98.
99. void pioPinHandler(){ //The PIO interrupt handler
100.     pio_interrupt_clear(pio0, 0); //Clear a particular PIO interrupt
101.     flag = 0; //Clear the flag
102. }
103.
104. //Function to initialize all the hardware associated with the TFT
105. void tft_init_hw(void){
106.     _width = ILI9340_TFTWIDTH;
107.     _height = ILI9340_TFTHEIGHT;
108.
109.     gpio_init(CS); //Initialize the CS pin as GPIO
110.     gpio_set_dir(CS, GPIO_OUT); //Set the pin direction as output
111.     gpio_put(CS, 1); //Drive CS high
112.
113.     gpio_init(RST); //Initialize the RST pin as GPIO
114.     gpio_set_dir(RST, GPIO_OUT); //Set the pin direction as output
115.
116.     gpio_init(DC); //Initialize the DC pin as GPIO
117.     gpio_set_dir(DC, GPIO_OUT); //Set the pin direction as output
118.
119.     offset = pio_add_program(spi.pio, &spi_cpha0_cs_program); //Attempt to load the program
120.     pio_spi_cs_init(spi.pio, spi.sm, offset, 8, 1, false, false, SCK, MOSI); //Initialize the
    SPI program
121.     pio_interrupt_clear(spi.pio, 0); //Clear a particular PIO interrupt
122.     pio_set_irq0_source_enabled(spi.pio, PIO_INTR_SM0_LSB, true); //Enable/Disable a single
    source on a PIO's IRQ 0
123.     irq_set_exclusive_handler(PIO0_IRQ_0, pioPinHandler); //Set an exclusive interrupt handler
    for an interrupt on the executing core
124.     irq_set_enabled(PIO0_IRQ_0, true); //Enable or disable a specific interrupt on the executing
    core
125.     sleep_ms(500); //Sleep for 500ms
126. }
127.
128. static inline void _rst_low(){ //Function to set the RST pin low
129.     gpio_put(RST, 0);
130. }
131.
132. static inline void _rst_high(){ //Function to set the RST pin high
133.     gpio_put(RST, 1);
134. }
135.
136. static inline void _dc_low(){ //Function to set the RST pin low
137.     gpio_put(DC, 0);
138. }
139.
140. static inline void _dc_high(){ //Function to set the DC pin high
141.     gpio_put(DC, 1);
142. }
143.
144. static inline void _cs_low(){ //Function to set the RST pin low
145.     gpio_put(CS, 0);
146. }
147.
148. static inline void _cs_high(){ //Function to set the CS pin high
149.     gpio_put(CS, 1);
150. }
151.

```

```

152. //Function to transmit a word to the SPI PIO
153. void __time_critical_func(pio_spi_write8_blocking)(const pio_spi_inst_t *spi, const uint8_t
    *src, size_t len){
154.     size_t tx_remain = len;
155.     io_rw_8 *txfifo = (io_rw_8 *) &spi->pio->txf[spi->sm];
156.     while (tx_remain){
157.         if (tx_remain && !pio_sm_is_tx_fifo_full(spi->pio, spi->sm)){
158.             *txfifo = *src++;
159.             --tx_remain;
160.         }
161.     }
162. }
163.
164.
165. void tft_spiwrite8(unsigned char c) { //Write 8 bits to the PIO
166.     pio_spi_write8_blocking(&spi, &c, 1);
167. }
168.
169. void tft_spiwrite16(unsigned short c) { //Write 16 bits to the PIO
170.     uint8_t data = (uint8_t) (c >> 8);
171.     pio_spi_write8_blocking(&spi, &data, 1); //Send lower 8 bits
172.     while(flag); //Wait for the transaction to complete
173.     flag = 1;
174.     data = (uint8_t) (c & 0xFF);
175.     pio_spi_write8_blocking(&spi, &data, 1); //Send upper 8 bits
176. }
177.
178. void tft_writecommand(unsigned char c) { //Send a command to the TFT screen
179.     _dc_low();
180.     _cs_low();
181.     tft_spiwrite8(c);
182.     while(flag);
183.     flag = 1;
184.     _cs_high();
185. }
186.
187. void tft_writedata(unsigned char c) { //Send 8-bit data to the TFT screen
188.     _dc_high();
189.     _cs_low();
190.     tft_spiwrite8(c);
191.     while(flag);
192.     flag = 1;
193.     _cs_high();
194. }
195.
196. void tft_writedata16(unsigned short c) { //Send 16-bit data to the TFT screen
197.     _dc_high();
198.     _cs_low();
199.     tft_spiwrite16(c);
200.     while(flag);
201.     flag = 1;
202.     _cs_high();
203. }
204.
205. void tft_begin(void) { //Initialize the TFT screen
206.     sleep_ms(500);
207.     _rst_low();
208.     _dc_low();
209.     _cs_high();
210.
211.     _rst_high();
212.     sleep_ms(5);
213.     _rst_low();
214.     sleep_ms(20);
215.     _rst_high();
216.     sleep_ms(150);
217.
218.     tft_writecommand(0xEF);
219.     tft_writedata(0x03);
220.     tft_writedata(0x80);
221.     tft_writedata(0x02);
222.
223.     tft_writecommand(0xCF);

```

```
224. tft_writedata(0x00);
225. tft_writedata(0xC1);
226. tft_writedata(0x30);
227.
228. tft_writecommand(0xED);
229. tft_writedata(0x64);
230. tft_writedata(0x03);
231. tft_writedata(0x12);
232. tft_writedata(0x81);
233.
234. tft_writecommand(0xE8);
235. tft_writedata(0x85);
236. tft_writedata(0x00);
237. tft_writedata(0x78);
238.
239. tft_writecommand(0xCB);
240. tft_writedata(0x39);
241. tft_writedata(0x2C);
242. tft_writedata(0x00);
243. tft_writedata(0x34);
244. tft_writedata(0x02);
245.
246. tft_writecommand(0xF7);
247. tft_writedata(0x20);
248.
249. tft_writecommand(0xEA);
250. tft_writedata(0x00);
251. tft_writedata(0x00);
252.
253. tft_writecommand(ILI9340_PWCTR1); //Power control
254. tft_writedata(0x23); //VRH[5:0]
255.
256. tft_writecommand(ILI9340_PWCTR2); //Power control
257. tft_writedata(0x10); //SAP[2:0]; BT[3:0]
258.
259. tft_writecommand(ILI9340_VMCTR1); //VCM control
260. tft_writedata(0x3e);
261. tft_writedata(0x28);
262.
263. tft_writecommand(ILI9340_VMCTR2); //VCM control2
264. tft_writedata(0x86);
265.
266. tft_writecommand(ILI9340_MADCTL); //Memory Access Control
267. tft_writedata(ILI9340_MADCTL_MX | ILI9340_MADCTL_BGR);
268.
269. tft_writecommand(ILI9340_PIXFMT);
270. tft_writedata(0x55);
271.
272. tft_writecommand(ILI9340_FRMCTR1);
273. tft_writedata(0x00);
274. tft_writedata(0x18);
275.
276. tft_writecommand(ILI9340_DFUNCTR); //Display Function Control
277. tft_writedata(0x08);
278. tft_writedata(0x82);
279. tft_writedata(0x27);
280.
281. tft_writecommand(0xF2); //3Gamma Function Disable
282. tft_writedata(0x00);
283.
284. tft_writecommand(ILI9340_GAMMASET); //Gamma curve selected
285. tft_writedata(0x01);
286.
287. tft_writecommand(ILI9340_GMCTR1); //Set Gamma
288. tft_writedata(0x0F);
289. tft_writedata(0x31);
290. tft_writedata(0x2B);
291. tft_writedata(0x0C);
292. tft_writedata(0x0E);
293. tft_writedata(0x08);
294. tft_writedata(0x4E);
295. tft_writedata(0xF1);
296. tft_writedata(0x37);
```

```

297.  tft_writedata(0x07);
298.  tft_writedata(0x10);
299.  tft_writedata(0x03);
300.  tft_writedata(0x0E);
301.  tft_writedata(0x09);
302.  tft_writedata(0x00);
303.
304.  tft_writecommand(ILI9340_GMCTRN1); //Set Gamma
305.  tft_writedata(0x00);
306.  tft_writedata(0x0E);
307.  tft_writedata(0x14);
308.  tft_writedata(0x03);
309.  tft_writedata(0x11);
310.  tft_writedata(0x07);
311.  tft_writedata(0x31);
312.  tft_writedata(0xC1);
313.  tft_writedata(0x48);
314.  tft_writedata(0x08);
315.  tft_writedata(0x0F);
316.  tft_writedata(0x0C);
317.  tft_writedata(0x31);
318.  tft_writedata(0x36);
319.  tft_writedata(0x0F);
320.
321.  tft_writecommand(ILI9340_SLPOUT); //Exit Sleep
322.  sleep_ms(120);
323.  tft_writecommand(ILI9340_DISPON); //Display on
324.  sleep_ms(500);
325. }
326.
327. /* Draw a pixel at location (x,y) with given color
328.  * Parameters:
329.  *     x: x-coordinate of pixel to draw; top left of screen is x=0
330.  *       and x increases to the right
331.  *     y: y-coordinate of pixel to draw; top left of screen is y=0
332.  *       and y increases to the bottom
333.  *     color: 16-bit color value
334.  * Returns:  Nothing
335.  */
336. void tft_drawPixel(short x, short y, unsigned short color) {
337.     if((x < 0) || (x >= _width) || (y < 0) || (y >= _height)){
338.         return;
339.     }
340.
341.     _dc_low();
342.     _cs_low();
343.     tft_spiwrite8(ILI9340_CASET);
344.     wait16;
345.     while(flag);
346.     flag = 1;
347.     _cs_high();
348.
349.     _dc_high();
350.     _cs_low();
351.     tft_spiwrite16(x);
352.     wait16;wait16;wait8;
353.     while(flag);
354.     flag = 1;
355.     _cs_high();
356.
357.     _cs_low();
358.     tft_spiwrite16(x + 1);
359.     wait16;wait16;wait8;
360.     while(flag);
361.     flag = 1;
362.     _cs_high();
363.
364.     _dc_low();
365.     _cs_low();
366.     tft_spiwrite8(ILI9340_PASET);
367.     wait16;wait8;
368.     while(flag);
369.     flag = 1;

```



```

370.     _cs_high();
371.
372.     _dc_high();
373.     _cs_low();
374.     tft_spiwrite16(y);
375.     wait16;wait16;wait8;
376.     while(flag);
377.     flag = 1;
378.     _cs_high();
379.
380.     _cs_low();
381.     tft_spiwrite16(y + 1);
382.     wait16;wait16;wait8;
383.     while(flag);
384.     flag = 1;
385.     _cs_high();
386.
387.     _dc_low();
388.     _cs_low();
389.     tft_spiwrite8(ILI9340_RAMWR);
390.     wait16;wait8;
391.     while(flag);
392.     flag = 1;
393.     _cs_high();
394.
395.     _dc_high();
396.     _cs_low();
397.     tft_spiwrite16(color);
398.     wait16;wait16;wait8;
399.     while(flag);
400.     flag = 1;
401.     _cs_high();
402. }
403.
404. void tft_setAddrWindow(unsigned short x0, unsigned short y0, unsigned short x1, unsigned short
y1) {
405.     tft_writecommand(ILI9340_CASET); //Column addr set
406.     tft_writedata16(x0);
407.     tft_writedata16(x1);
408.
409.     tft_writecommand(ILI9340_PASET); //Row addr set
410.     tft_writedata16(y0);
411.     tft_writedata16(y1);
412.
413.     tft_writecommand(ILI9340_RAMWR); //Write to RAM
414. }
415.
416. void tft_pushColor(unsigned short color) {
417.     _dc_high();
418.     _cs_low();
419.     tft_spiwrite16(color);
420.     while(flag);
421.     flag = 1;
422.     _cs_high();
423. }
424.
425. /* Draw a vertical line at location from (x,y) to (x,y+h-1) with color
426. * Parameters:
427. *     x:  x-coordinate line to draw; top left of screen is x=0
428. *         and x increases to the right
429. *     y:  y-coordinate of starting point of line; top left of screen is y=0
430. *         and y increases to the bottom
431. *     h:  height of line to draw
432. *     color:  16-bit color value
433. * Returns:  Nothing
434. */
435. void tft_drawFastVLine(short x, short y, short h, unsigned short color) {
436.     //Rudimentary clipping
437.     if((x >= _width) || (y >= _height)){
438.         return;
439.     }
440.     if((y + h - 1) >= _height){
441.         h = _height - y;

```

```

442.     }
443.
444.     tft_setAddrWindow(x, y, x, y + h - 1);
445.     _dc_high();
446.     _cs_low();
447.
448.     while (h--) {
449.         tft_spiwrite16(color);
450.         while(flag);
451.         flag = 1;
452.     }
453.     _cs_high();
454. }
455.
456. /* Draw a horizontal line at location from (x,y) to (x+w-1,y) with color
457. * Parameters:
458. *     x: x-coordinate starting point of line; top left of screen is x=0
459. *         and x increases to the right
460. *     y: y-coordinate of starting point of line; top left of screen is y=0
461. *         and y increases to the bottom
462. *     w: width of line to draw
463. *     color: 16-bit color value
464. * Returns:    Nothing
465. */
466. void tft_drawFastHLine(short x, short y, short w, unsigned short color){
467.     //Rudimentary clipping
468.     if((x >= _width) || (y >= _height)){
469.         return;
470.     }
471.     if((x + w - 1) >= _width){
472.         w = _width - x;
473.     }
474.
475.     tft_setAddrWindow(x, y, x + w - 1, y);
476.     _dc_high();
477.     _cs_low();
478.
479.     while (w--) {
480.         tft_spiwrite16(color);
481.         while(flag);
482.         flag = 1;
483.     }
484.     _cs_high();
485. }
486.
487. /* Fill entire screen with given color
488. * Parameters:
489. *     color: 16-bit color value
490. * Returns:    Nothing
491. */
492. void tft_fillScreen(unsigned short color) {
493.     tft_fillRect(0, 0, _width, _height, color);
494. }
495.
496. /* Draw a filled rectangle with starting top-left vertex (x,y),
497. * width w and height h with given color
498. * Parameters:
499. *     x: x-coordinate of top-left vertex; top left of screen is x=0
500. *         and x increases to the right
501. *     y: y-coordinate of top-left vertex; top left of screen is y=0
502. *         and y increases to the bottom
503. *     w: width of rectangle
504. *     h: height of rectangle
505. *     color: 16-bit color value
506. * Returns:    Nothing
507. */
508. void tft_fillRect(short x, short y, short w, short h, unsigned short color) {
509.     //Rudimentary clipping
510.     if((x >= _width) || (y >= _height)){
511.         return;
512.     }
513.     if((x + w - 1) >= _width){
514.         w = _width - x;

```

```

515.     }
516.     if((y + h - 1) >= _height){
517.         h = _height - y;
518.     }
519.     tft_setAddrWindow(x, y, x + w - 1, y + h - 1);
520.     _dc_high();
521.     _cs_low();
522.     for(y = h; y > 0; y--) {
523.         for(x = w; x > 0; x--) {
524.             tft_spiwrite16(color);
525.             while(flag);
526.             flag = 1;
527.         }
528.     }
529.     // while(flag);
530.     // flag = 1;
531.     _cs_high();
532. }
533.
534. /* Pass 8-bit (each) R,G,B, get back 16-bit packed color
535.  * Parameters:
536.  *     r: 8-bit R/red value from RGB
537.  *     g: 8-bit g/green value from RGB
538.  *     b: 8-bit b/blue value from RGB
539.  * Returns:
540.  *     16-bit packed color value for color info
541.  */
542. inline unsigned short tft_Color565(unsigned char r, unsigned char g, unsigned char b) {
543.     return ((r & 0xF8) << 8) | ((g & 0xFC) << 3) | (b >> 3);
544. }
545.
546. void tft_setRotation(unsigned char m) {
547.     unsigned char rotation;
548.     tft_writecommand(ILI9340_MADCTL);
549.     rotation = m % 4; //Can't be higher than 3
550.     switch (rotation) {
551.         case 0: tft_writedata(ILI9340_MADCTL_MX | ILI9340_MADCTL_BGR);
552.             _width = ILI9340_TFTWIDTH;
553.             _height = ILI9340_TFTHEIGHT;
554.             break;
555.         case 1: tft_writedata(ILI9340_MADCTL_MV | ILI9340_MADCTL_BGR);
556.             _width = ILI9340_TFTHEIGHT;
557.             _height = ILI9340_TFTWIDTH;
558.             break;
559.         case 2: tft_writedata(ILI9340_MADCTL_MY | ILI9340_MADCTL_BGR);
560.             _width = ILI9340_TFTWIDTH;
561.             _height = ILI9340_TFTHEIGHT;
562.             break;
563.         case 3: tft_writedata(ILI9340_MADCTL_MV | ILI9340_MADCTL_MY | ILI9340_MADCTL_MX |
ILI9340_MADCTL_BGR);
564.             _width = ILI9340_TFTHEIGHT;
565.             _height = ILI9340_TFTWIDTH;
566.             break;
567.     }
568. }
569.
570. /* Draw a circle outline with center (x0,y0) and radius r, with given color
571.  * Parameters:
572.  *     x0: x-coordinate of center of circle. The top-left of the screen
573.  *         has x-coordinate 0 and increases to the right
574.  *     y0: y-coordinate of center of circle. The top-left of the screen
575.  *         has y-coordinate 0 and increases to the bottom
576.  *     r: radius of circle
577.  *     color: 16-bit color value for the circle. Note that the circle
578.  *           isn't filled. So, this is the color of the outline of the circle
579.  * Returns: Nothing
580.  */
581. void tft_drawCircle(short x0, short y0, short r, unsigned short color) {
582.     short f = 1 - r;
583.     short ddF_x = 1;
584.     short ddF_y = -2 * r;
585.     short x = 0;
586.     short y = r;

```

```

587.
588.     tft_drawPixel(x0, y0 + r, color);
589.     tft_drawPixel(x0, y0 - r, color);
590.     tft_drawPixel(x0 + r, y0, color);
591.     tft_drawPixel(x0 - r, y0, color);
592.
593.     while(x < y) {
594.         if(f >= 0){
595.             y--;
596.             ddF_y += 2;
597.             f += ddF_y;
598.         }
599.         x++;
600.         ddF_x += 2;
601.         f += ddF_x;
602.
603.         tft_drawPixel(x0 + x, y0 + y, color);
604.         tft_drawPixel(x0 - x, y0 + y, color);
605.         tft_drawPixel(x0 + x, y0 - y, color);
606.         tft_drawPixel(x0 - x, y0 - y, color);
607.         tft_drawPixel(x0 + y, y0 + x, color);
608.         tft_drawPixel(x0 - y, y0 + x, color);
609.         tft_drawPixel(x0 + y, y0 - x, color);
610.         tft_drawPixel(x0 - y, y0 - x, color);
611.     }
612. }
613.
614. //Helper function for drawing circles and circular objects
615. void tft_drawCircleHelper( short x0, short y0, short r, unsigned char cornername, unsigned
short color){
616.     short f = 1 - r;
617.     short ddF_x = 1;
618.     short ddF_y = -2 * r;
619.     short x = 0;
620.     short y = r;
621.
622.     while(x < y){
623.         if(f >= 0){
624.             y--;
625.             ddF_y += 2;
626.             f += ddF_y;
627.         }
628.         x++;
629.         ddF_x += 2;
630.         f += ddF_x;
631.         if(cornername & 0x4){
632.             tft_drawPixel(x0 + x, y0 + y, color);
633.             tft_drawPixel(x0 + y, y0 + x, color);
634.         }
635.         if (cornername & 0x2) {
636.             tft_drawPixel(x0 + x, y0 - y, color);
637.             tft_drawPixel(x0 + y, y0 - x, color);
638.         }
639.         if (cornername & 0x8) {
640.             tft_drawPixel(x0 - y, y0 + x, color);
641.             tft_drawPixel(x0 - x, y0 + y, color);
642.         }
643.         if (cornername & 0x1) {
644.             tft_drawPixel(x0 - y, y0 - x, color);
645.             tft_drawPixel(x0 - x, y0 - y, color);
646.         }
647.     }
648. }
649.
650. /* Draw a filled circle with center (x0,y0) and radius r, with given color
651. * Parameters:
652. *     x0: x-coordinate of center of circle. The top-left of the screen
653. *         has x-coordinate 0 and increases to the right
654. *     y0: y-coordinate of center of circle. The top-left of the screen
655. *         has y-coordinate 0 and increases to the bottom
656. *     r: radius of circle
657. *     color: 16-bit color value for the circle
658. * Returns: Nothing

```

```

659.  */
660. void tft_fillCircle(short x0, short y0, short r, unsigned short color) {
661.     tft_drawFastVLine(x0, y0 - r, 2 * r + 1, color);
662.     tft_fillCircleHelper(x0, y0, r, 3, 0, color);
663. }
664.
665. //Helper function for drawing filled circles
666. void tft_fillCircleHelper(short x0, short y0, short r, unsigned char cornername, short delta,
    unsigned short color) {
667.     short f = 1 - r;
668.     short ddF_x = 1;
669.     short ddF_y = -2 * r;
670.     short x = 0;
671.     short y = r;
672.
673.     while(x < y){
674.         if (f >= 0) {
675.             y--;
676.             ddF_y += 2;
677.             f += ddF_y;
678.         }
679.         x++;
680.         ddF_x += 2;
681.         f += ddF_x;
682.
683.         if (cornername & 0x1) {
684.             tft_drawFastVLine(x0 + x, y0 - y, 2 * y + 1 + delta, color);
685.             tft_drawFastVLine(x0 + y, y0 - x, 2 * x + 1 + delta, color);
686.         }
687.         if (cornername & 0x2) {
688.             tft_drawFastVLine(x0 - x, y0 - y, 2 * y + 1 + delta, color);
689.             tft_drawFastVLine(x0 - y, y0 - x, 2 * x + 1 + delta, color);
690.         }
691.     }
692. }
693.
694. //Bresenham's algorithm
695.
696. /* Draw a straight line from (x0,y0) to (x1,y1) with given color
697.  * Parameters:
698.  *     x0: x-coordinate of starting point of line. The x-coordinate of
699.  *         the top-left of the screen is 0. It increases to the right.
700.  *     y0: y-coordinate of starting point of line. The y-coordinate of
701.  *         the top-left of the screen is 0. It increases to the bottom.
702.  *     x1: x-coordinate of ending point of line. The x-coordinate of
703.  *         the top-left of the screen is 0. It increases to the right.
704.  *     y1: y-coordinate of ending point of line. The y-coordinate of
705.  *         the top-left of the screen is 0. It increases to the bottom.
706.  *     color: 16-bit color value for line
707.  */
708. void tft_drawLine(short x0, short y0, short x1, short y1, unsigned short color) {
709.     short steep = abs(y1 - y0) > abs(x1 - x0);
710.     if(steep){
711.         swap(x0, y0);
712.         swap(x1, y1);
713.     }
714.
715.     if (x0 > x1) {
716.         swap(x0, x1);
717.         swap(y0, y1);
718.     }
719.
720.     short dx, dy;
721.     dx = x1 - x0;
722.     dy = abs(y1 - y0);
723.
724.     short err = dx / 2;
725.     short ystep;
726.
727.     if (y0 < y1) {
728.         ystep = 1;
729.     }
730.     else {

```

```

731.     ystep = -1;
732. }
733.
734. for(; x0 <= x1; x0++){
735.     if (steep) {
736.         tft_drawPixel(y0, x0, color);
737.     }
738.     else {
739.         tft_drawPixel(x0, y0, color);
740.     }
741.     err -= dy;
742.     if (err < 0) {
743.         y0 += ystep;
744.         err += dx;
745.     }
746. }
747. }
748.
749. /* Draw a rectangle outline with top left vertex (x,y), width w
750. * and height h at given color
751. * Parameters:
752. *     x: x-coordinate of top-left vertex. The x-coordinate of
753. *        the top-left of the screen is 0. It increases to the right.
754. *     y: y-coordinate of top-left vertex. The y-coordinate of
755. *        the top-left of the screen is 0. It increases to the bottom.
756. *     w: width of the rectangle
757. *     h: height of the rectangle
758. *     color: 16-bit color of the rectangle outline
759. * Returns: Nothing
760. */
761. void tft_drawRect(short x, short y, short w, short h, unsigned short color) {
762.     tft_drawFastHLine(x, y, w, color);
763.     tft_drawFastHLine(x, y + h - 1, w, color);
764.     tft_drawFastVLine(x, y, h, color);
765.     tft_drawFastVLine(x + w - 1, y, h, color);
766. }
767.
768. /* Draw a rounded rectangle outline with top left vertex (x,y), width w,
769. * height h and radius of curvature r at given color
770. * Parameters:
771. *     x: x-coordinate of top-left vertex. The x-coordinate of
772. *        the top-left of the screen is 0. It increases to the right.
773. *     y: y-coordinate of top-left vertex. The y-coordinate of
774. *        the top-left of the screen is 0. It increases to the bottom.
775. *     w: width of the rectangle
776. *     h: height of the rectangle
777. *     color: 16-bit color of the rectangle outline
778. * Returns: Nothing
779. */
780. void tft_drawRoundRect(short x, short y, short w, short h, short r, unsigned short color) {
781.     tft_drawFastHLine(x + r, y, w - 2 * r, color);
782.     tft_drawFastHLine(x + r, y + h - 1, w - 2 * r, color);
783.     tft_drawFastVLine(x, y + r, h - 2 * r, color);
784.     tft_drawFastVLine(x + w - 1, y + r, h - 2 * r, color);
785.
786.     tft_drawCircleHelper(x + r, y + r, r, 1, color);
787.     tft_drawCircleHelper(x + w - r - 1, y + r, r, 2, color);
788.     tft_drawCircleHelper(x + w - r - 1, y + h - r - 1, r, 4, color);
789.     tft_drawCircleHelper(x + r, y + h - r - 1, r, 8, color);
790. }
791.
792. //Fill a rounded rectangle
793. void tft_fillRoundRect(short x, short y, short w, short h, short r, unsigned short color) {
794.     tft_fillRect(x + r, y, w - 2 * r, h, color);
795.
796.     tft_fillCircleHelper(x + w - r - 1, y + r, r, 1, h - 2 * r - 1, color);
797.     tft_fillCircleHelper(x + r, y + r, r, 2, h - 2 * r - 1, color);
798. }
799.
800. /* Draw a triangle outline with vertices (x0,y0),(x1,y1),(x2,y2) with given color
801. * Parameters:
802. *     x0: x-coordinate of one of the 3 vertices
803. *     y0: y-coordinate of one of the 3 vertices

```

```

804. *      x1: x-coordinate of one of the 3 vertices
805. *      y1: y-coordinate of one of the 3 vertices
806. *      x2: x-coordinate of one of the 3 vertices
807. *      y2: y-coordinate of one of the 3 vertices
808. *      color: 16-bit color value for outline
809. * Returns: Nothing
810. */
811. void tft_drawTriangle(short x0, short y0, short x1, short y1, short x2, short y2, unsigned
short color) {
812.     tft_drawLine(x0, y0, x1, y1, color);
813.     tft_drawLine(x1, y1, x2, y2, color);
814.     tft_drawLine(x2, y2, x0, y0, color);
815. }
816.
817. /* Draw a filled triangle with vertices (x0,y0),(x1,y1),(x2,y2) with given color
818. * Parameters:
819. *      x0: x-coordinate of one of the 3 vertices
820. *      y0: y-coordinate of one of the 3 vertices
821. *      x1: x-coordinate of one of the 3 vertices
822. *      y1: y-coordinate of one of the 3 vertices
823. *      x2: x-coordinate of one of the 3 vertices
824. *      y2: y-coordinate of one of the 3 vertices
825. *      color: 16-bit color value
826. * Returns: Nothing
827. */
828. void tft_fillTriangle (short x0, short y0, short x1, short y1, short x2, short y2, unsigned
short color) {
829.     short a, b, y, last;
830.
831.     if(y0 > y1){
832.         swap(y0, y1);
833.         swap(x0, x1);
834.     }
835.     if(y1 > y2){
836.         swap(y2, y1);
837.         swap(x2, x1);
838.     }
839.     if(y0 > y1){
840.         swap(y0, y1);
841.         swap(x0, x1);
842.     }
843.
844.     if(y0 == y2){
845.         a = b = x0;
846.         if(x1 < a){
847.             a = x1;
848.         }
849.         else if(x1 > b){
850.             b = x1;
851.         }
852.         if(x2 < a){
853.             a = x2;
854.         }
855.         else if(x2 > b){
856.             b = x2;
857.         }
858.         tft_drawFastHLine(a, y0, b - a + 1, color);
859.         return;
860.     }
861.
862.     short
863.     dx01 = x1 - x0,
864.     dy01 = y1 - y0,
865.     dx02 = x2 - x0,
866.     dy02 = y2 - y0,
867.     dx12 = x2 - x1,
868.     dy12 = y2 - y1,
869.     sa = 0,
870.     sb = 0;
871.
872.     // For upper part of triangle, find scanline crossings for segments
873.     // 0-1 and 0-2. If y1=y2 (flat-bottomed triangle), the scanline y1
874.     // is included here (and second loop will be skipped, avoiding a /0

```

```

875. // error there), otherwise scanline y1 is skipped here and handled
876. // in the second loop...which also avoids a /0 error here if y0=y1
877. // (flat-topped triangle).
878.
879. if(y1 == y2){
880.     last = y1;
881. }
882. else{
883.     last = y1 - 1;
884. }
885.
886. for(y=y0; y<=last; y++) {
887.     a = x0 + sa / dy01;
888.     b = x0 + sb / dy02;
889.     sa += dx01;
890.     sb += dx02;
891.     /* longhand:
892.     a = x0 + (x1 - x0) * (y - y0) / (y1 - y0);
893.     b = x0 + (x2 - x0) * (y - y0) / (y2 - y0);
894.     */
895.     if(a > b){
896.         swap(a,b);
897.     }
898.     tft_drawFastHLine(a, y, b - a + 1, color);
899. }
900. // For lower part of triangle, find scanline crossings for segments
901. // 0-2 and 1-2. This loop is skipped if y1 = y2.
902. sa = dx12 * (y - y1);
903. sb = dx02 * (y - y0);
904. for(; y<=y2; y++) {
905.     a = x1 + sa / dy12;
906.     b = x0 + sb / dy02;
907.     sa += dx12;
908.     sb += dx02;
909.     /* longhand:
910.     a = x1 + (x2 - x1) * (y - y1) / (y2 - y1);
911.     b = x0 + (x2 - x0) * (y - y0) / (y2 - y0);
912.     */
913.     if(a > b){
914.         swap(a, b);
915.     }
916.     tft_drawFastHLine(a, y, b - a + 1, color);
917. }
918. }
919.
920. //Function to draw the bitmaps on the TFT
921. void tft_drawBitmap(short x, short y, const unsigned char *bitmap, short w, short h, unsigned
short color){
922.     //Rudimentary clipping
923.     if((x >= _width) || (y >= _height)) return;
924.     if((x + w - 1) >= _width) w = _width - x;
925.     if((y + h - 1) >= _height) h = _height - y;
926.
927.     short i, j, byteWidth = (w + 7) / 8;
928.     tft_setAddrWindow(x, y, x + w - 1, y + h - 1);
929.     _dc_high();
930.     _cs_low();
931.     for(j = 0; j < h; j++){
932.         for(i = 0; i < w; i++){
933.             if(pgm_read_byte(bitmap + j * byteWidth + i / 8) & (128 >> (i & 7))) {
934.                 tft_spiwrite16(color);
935.                 while(flag);
936.                 flag = 1;
937.             }
938.             else{
939.                 tft_spiwrite16(0x0000);
940.                 while(flag);
941.                 flag = 1;
942.             }
943.         }
944.     }
945.     _cs_high();
946. }

```



```

947.
948. void tft_write(unsigned char c){
949.     if (c == '\n') {
950.         cursor_y += textsize*8;
951.         cursor_x = 0;
952.     }
953.     else if (c == '\r') {
954.
955.     }
956.     else if (c == '\t'){
957.         int new_x = cursor_x + tabspace;
958.         if (new_x < _width){
959.             cursor_x = new_x;
960.         }
961.     }
962.     else {
963.         tft_drawChar(cursor_x, cursor_y, c, textcolor, textbgcolor, textsize);
964.         cursor_x += textsize * 6;
965.         if (wrap && (cursor_x > (_width - textsize * 6))) {
966.             cursor_y += textsize * 8;
967.             cursor_x = 0;
968.         }
969.     }
970. }
971.
972. /* Print text onto screen
973.  * Call tft_setCursor(), tft_setTextColor(), tft_setTextSize()
974.  * as necessary before printing
975.  */
976. inline void tft_writeString(char* str){
977.     while (*str){
978.         tft_write(*str++);
979.     }
980. }
981.
982. //Draw a character
983. void tft_drawChar(short x, short y, unsigned char c, unsigned short color, unsigned short bg,
unsigned char size) {
984.     char i, j;
985.     if((x >= _width) || (y >= _height) || ((x + 6 * size - 1) < 0) || ((y + 8 * size - 1) < 0)){
986.         return;
987.     }
988.
989.     for(i = 0; i < 6; i++){
990.         unsigned char line;
991.         if(i == 5){
992.             line = 0x0;
993.         }
994.         else{
995.             line = pgm_read_byte(font + (c * 5) + i);
996.         }
997.         for(j = 0; j < 8; j++){
998.             if (line & 0x1) {
999.                 if(size == 1){
1000.                     tft_drawPixel(x + i, y + j, color);
1001.                 }
1002.                 else{
1003.                     tft_fillRect(x + (i * size), y + (j * size), size, size, color);
1004.                 }
1005.             }
1006.             else if (bg != color) {
1007.                 if (size == 1){
1008.                     tft_drawPixel(x + i, y + j, bg);
1009.                 }
1010.                 else {
1011.                     tft_fillRect(x + i * size, y + j * size, size, size, bg);
1012.                 }
1013.             }
1014.             line >>= 1;
1015.         }
1016.     }
1017. }
1018.

```

```

1019. /*Set size of text to be displayed
1020. * Parameters:
1021. *     s = text size (1 being smallest)
1022. * Returns: nothing
1023. */
1024. inline void tft_setCursor(short x, short y){
1025.     cursor_x = x;
1026.     cursor_y = y;
1027. }
1028.
1029. inline void tft_setTextSize(unsigned char s) {
1030.     textsize = (s > 0) ? s : 1;
1031. }
1032.
1033. inline void tft_setTextColor(unsigned short c) {
1034.     textcolor = textbgcolor = c;
1035. }
1036.
1037. /* Set color of text to be displayed
1038. * Parameters:
1039. *     c = 16-bit color of text
1040. *     b = 16-bit color of text background
1041. */
1042. inline void tft_setTextColor2(unsigned short c, unsigned short b) {
1043.     textcolor = c;
1044.     textbgcolor = b;
1045. }
1046.
1047. inline void tft_setTextWrap(char w) {
1048.     wrap = w;
1049. }
1050.
1051. /* Returns current roation of screen
1052. *     0 = no rotation (0 degree rotation)
1053. *     1 = rotate 90 degree clockwise
1054. *     2 = rotate 180 degree
1055. *     3 = rotate 90 degree anticlockwise
1056. */
1057. inline unsigned char tft_getRotation(void) {
1058.     return rotation;
1059. }
1060.
1061. /* Set display rotation in 90 degree steps
1062. * Parameters:
1063. *     x: dictate direction of rotation
1064. *     0 = no rotation (0 degree rotation)
1065. *     1 = rotate 90 degree clockwise
1066. *     2 = rotate 180 degree
1067. *     3 = rotate 90 degree anticlockwise
1068. * Returns: Nothing
1069. */
1070. void tft_gfx_setRotation(unsigned char x) {
1071.     rotation = (x & 3);
1072.     switch(rotation) {
1073.         case 0:
1074.             case 2: _width = ILI9340_TFTWIDTH;
1075.                 _height = ILI9340_TFTHEIGHT;
1076.                 break;
1077.         case 1:
1078.             case 3: _width = ILI9340_TFTHEIGHT;;
1079.                 _height = ILI9340_TFTWIDTH;
1080.                 break;
1081.     }
1082. }
1083.
1084. //Return the size of the display (per current rotation)
1085. inline short tft_width(void) {
1086.     return _width;
1087. }
1088.
1089. inline short tft_height(void) {
1090.     return _height;

```

SPIOPIO.pio

```

1. ;Parth Sarthi Sharma (pss242@cornell.edu)
2. ;SPI driver for TFT
3.
4. .program spi_cpha0_cs ;Program name
5. .side_set 1 ;Set 1 pin for sideset
6.
7. ; Drive SPI
8. ; Pin assignments:
9. ; - SCK is side-set bit 0
10. ; - MOSI is OUT bit 0 (host-to-device)
11.
12. .wrap_target ;Free 0 cycle unconditional jump
13. bitloop: ;Bitloop label
14. public entry_point: ;The entry point for the program
15.     out pins, 1         side 0x0 [1] ;Output the bit on pin, sideset the clock
16.     jmp x-- bitloop    side 0x1 [1] ;Jump to bitloop if bit counter still available
17.
18.     out pins, 1         side 0x0 ;Output the bit on pin, sideset the clock
19.     mov x, y           side 0x0   ;Reload bit counter from Y
20.     jmp !osre bitloop  side 0x1 [1] ;Fall-through if TXF empties
21.
22.     irq 0              side 0x0 [1] ;Set IRQ 0 flag
23. .wrap
24.
25. ;Helper function
26.
27. % c-sdk {
28. #include "hardware/gpio.h" //The hardware GPIO library
29. static inline void pio_spi_cs_init(PIO pio, uint sm, uint prog_offs, uint n_bits, int clkdiv,
    bool cpha, bool cpol, uint pin_sck, uint pin_mosi){ //The PIO SPI initialize functions
30.     pio_sm_config c = spi_cpha0_cs_program_get_default_config(prog_offs); //Get default
    configurations for the PIO state machine
31.     sm_config_set_out_pins(&c, pin_mosi, 1); //Set the 'out' pins in a state machine
    configuration
32.     sm_config_set_sideset_pins(&c, pin_sck); //Set the 'sideset' pins in a state machine
    configuration
33.     sm_config_set_out_shift(&c, false, true, n_bits); //Setup 'out' shifting parameters in a
    state machine configuration
34.     sm_config_set_clkdiv(&c, clkdiv); //Set the state machine clock divider
35.
36.     pio_sm_set_pins_with_mask(pio, sm, 0, (1u << pin_sck) | (1u << pin_mosi)); //Use a state
    machine to set a value on multiple pins for the PIO instance
37.     pio_sm_set_pindirs_with_mask(pio, sm, (1u << pin_sck) | (1u << pin_mosi), (1u << pin_sck) |
    (1u << pin_mosi)); //Use a state machine to set the pin directions for multiple pins for the PIO
    instance
38.     pio_gpio_init(pio, pin_mosi); //Setup the function select for a GPIO to use output from the
    given PIO instance
39.     pio_gpio_init(pio, pin_sck); //Setup the function select for a GPIO to use output from the
    given PIO instance
40.     //pio_gpio_init(pio, pin_sck + 1); //Setup the function select for a GPIO to use output from
    the given PIO instance
41.     gpio_set_outover(pin_sck, cpol ? GPIO_OVERRIDE_INVERT : GPIO_OVERRIDE_NORMAL); //Set GPIO
    output override
42.
43.
44.     uint entry_point = prog_offs + spi_cpha0_cs_offset_entry_point; //The offset entry point
45.     pio_sm_init(pio, sm, entry_point, &c); //Resets the state machine to a consistent state, and
    configures it
46.     pio_sm_exec(pio, sm, pio_encode_set(pio_x, n_bits - 2)); //Put n_bits - 2 in pio_x
47.     pio_sm_exec(pio, sm, pio_encode_set(pio_y, n_bits - 2)); //Put n_bits - 2 in pio_y
48.     pio_sm_set_enabled(pio, sm, true); //Enable or disable a PIO state machine
49. }
50. %}

```