Lab Prototype Board for the Pi Pico RP2040

A Design Project Report
Presented to the School of Electrical and Computer Engineering of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by
Andrew Tsai, Felipe Shiwa
MEng Field Advisor: Van Hunter Adams
Degree Date: January 2021

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Lab Prototype Board for the Pi Pico RP2040

Authors: Andrew Tsai, Felipe Shiwa

Abstract: The RP2040 is a low-cost, feature-rich microcontroller recently developed and released by the Raspberry Pi Foundation. Due to its potential as a teaching tool and for student projects, it has been chosen for evaluation as a potential replacement for the PIC32, the current microcontroller used in ECE4760, Designing with Microcontrollers. In this report, we detail our efforts to familiarize ourselves with the Raspberry Pi Pico and subsequently design a prototype lab board that implements it. We then assess our board as a replacement for the current PIC32-based lab board used by ECE4760 by analyzing its programmability and ease of interfacing, some of its hardware and software limitations, and lastly integrability into the course curriculum. We conclude with results from our development efforts in both hardware and software as well as a discussion on the educational value that the Pico and RP2040 can provide for the course.

Individual Contributions

This project was carried out across the Spring and Fall 2021 semesters by two group members, Andrew Tsai and Felipe Shiwa. The specific project work is described in detail in the body of the report. The individual contributions for the first semester are as follows. Andrew developed the ex_protothread and ex_array programs, while Felipe developed the ex_multicore and ex_count programs. For the n_queens and n_bodies projects, these were already previously written in C, while Andrew configured n_queens to run on the Pico and Felipe handled the n_bodies simulation. However, it is important to note a significant degree of pair programming was used throughout a majority of the development effort in this semester. Both group members had an equal understanding of the work being done, even if one member was leading development. Program testing and evaluation was always performed with both members present.

For the second semester, Felipe started out by figuring out the pin assignments and developed the corresponding Pico pinout section of the schematic, while Andrew provided feedback and afterwards finished the schematic implementation with routing the rest of the peripherals. Both members co-developed the prototype breadboard and pair programming was used for writing the pico_master project for testing the peripherals. Then, since Andrew had more PCB design experience, he took the lead for designing the PCB layout, while Felipe simultaneously worked on refactoring the software library for the 1.14 LCD display. When the first iteration of the board arrived, Andrew would populate the board components incrementally and at various points hand off the board to Felipe to run software tests and verify its functionality. Both members contributed to putting together the list of board revisions. While waiting for version 1.1, Felipe finished the custom, rewritten LCD display library, while Andrew put together the pico_board_lib library by consolidating some of the previously written code. Both members then verified that the libraries worked together without conflicts after obtaining the revised board version, and co-developed the final demonstration program. The report was written and proofread in equal parts by both members.

Executive Summary

The RP2040 is a microcontroller recently developed and released by the Raspberry Pi Foundation. It is a low-cost yet feature-rich device that has great potential as a teaching tool and for student projects. For our Master of Engineering project, we developed a lab board prototype for the Raspberry Pi Pico (referred to as the Pico), a commercially available microcontroller platform using the RP2040 released by the Raspberry Pi foundation. The key objectives of this project were to construct such a prototype board, and to subsequently assess its potential as a replacement for the PIC32-based lab board currently used by ECE4760, Designing with Microcontrollers. We evaluated whether the microcontroller's advanced specifications would sufficiently benefit the course and enhance student experiences to warrant the changes that would need to be made to the existing course structure. To do this, we assessed its programmability and ease of interfacing, some of its hardware and software limitations, and lastly integrability into the course curriculum.

Our project work was split across two semesters, Spring 2021 and Fall 2021. In the Spring semester, we set out to address the issues of programmability, ease of interfacing, and discovering the hardware and software limitations. To gain familiarity with the RP2040 and the Pico board, we created a variety of software programs benchmarking the performance of the microcontroller. The main focus was on the multiprocessing capabilities of the RP2040, but we also explored additional interfaces such as driving a VGA display. These software programs provided valuable experience and improved our understanding of its capabilities.

In our Fall semester, we set out to tackle the problem of integration into the course curriculum. We applied the knowledge gained in the Spring to design a PCB for the Pico. This PCB used sockets for mounting the Pico to the board, and provided connectivity to key on-board peripherals commonly used in the course lab exercises: a 12-bit DAC, LCD display, GPIO port expander, and inertial measurement unit (IMU). Additionally, we developed a software library for interfacing with the on-board peripherals centered around the calls from the RP2040 C SDK library. This library demonstrates how to communicate with the peripherals we include on our PCB board across the different protocols for each of the components, preparing students to utilize their own choice of peripherals for a design project. We also demonstrated the board's capabilities in the context of the course with a sample design project where we utilized the prototype board to drive a dual-display, motion-controlled asteroids game.

After working with and designing around the Pico board for two semesters, we can conclude that it has all of the requisite features for the course, and still offers much more. It is indeed an upgrade to the PIC32MX in terms of performance, user-friendliness, and capability range, and therefore can bring significant educational value to the course's lab exercises. Furthermore, the board that we designed can broaden the scope of student projects if it is indeed used. There are many features on the microcontroller that the course curriculum simply will not have the scope to cover in depth, but students can take advantage of regardless. Basic examples of this include the greatly expanded graphics capabilities over the PIC32, the peripheral I/O features, and even multiprocessing using the two existing cores. These and the other features can provide students with greater creative freedom for their course design project, which is optimal for a Culminating Design Experience course.

Introduction

The RP2040 is a microcontroller recently developed and released by the Raspberry Pi Foundation. It is a low-cost yet feature-rich device that has great potential as a teaching tool and for student projects. For our Master of Engineering project, we developed a lab board prototype for the Pico, a commercially available microcontroller platform using the RP2040. The key objectives of this project were to construct such a prototype board, and to subsequently assess its potential as a replacement for the PIC32-based lab board currently used by ECE4760, Designing with Microcontrollers. In this report, we will start by describing the background behind this project, the problem statement and specific issues to address. We then follow with our design approaches to solving the problems, our testing procedures, and results obtained for both semesters of work. We conclude with our analysis on the results of our work and our assessment on the potential of the Pico to improve the course.

Background

Designing with Microcontrollers, a Culminating Design Experience course for the ECE major at Cornell, has been a popular elective offering for both undergraduate and graduate engineering students for many years. The current curriculum is centered around usage of the PIC32MX microcontroller for a variety of application-specific lab exercises as well as an end-of-semester student design project. The three lab exercises currently implemented in the course focus on audio synthesis, animation and graphics, and sensor-based closed-loop control, respectively. The software framework used throughout the course is based on protothreads as a scheduler for the code running on the PIC32MX. Released in 2007, the PIC32MX comes with a single MIPS32 M4K core which can be overclocked up to ~70MHz, up to four SPI interfaces, a 10-bit ADC, and function remapping through Peripheral Pin Select, among other features. While this is sufficient for an already-wide range of electrical lab exercises and projects, it is always worthwhile to examine whether advancements in microcontroller technology can lead to improvements for the course, and in this area the RP2040 is a very strong candidate. Released in 2020, this microcontroller features a dual-core Arm Cortex-M0+ processor running at up to 133MHz, 3 12-bit ADC's, 2 each of SPI, I^2C, and UART controllers, programmable I/O state machines, and much more. These additional features have the potential to enrich the course curriculum as a whole by expanding the capabilities of the hardware and software used by students.

Problem Statement and Issues to Address

Despite its potential, it is necessary to analyze the RP2040 to assess its practicality in the classroom and usability from a student perspective. We were tasked with evaluating whether the microcontroller's advanced specifications would sufficiently benefit the course and enhance student experiences to warrant the changes that would need to be made to the pre-established course structure. In order to perform this assessment, several key issues must be addressed,

including: programmability, ease of interfacing, finding the hardware and software limitations, and integrability into the course curriculum. Our work is spread across two semesters, Spring 2021 and Fall 2021, with different goals in each semester. In the Spring, our work was focused on discovering the programmability, ease of interfacing, and limitations of the Pico. After obtaining this background, we set out to tackle the issue of integrability into ECE4760 in the Fall. The following sections describe in detail our work in each semester and how our efforts achieved the goals we have laid out.

Design and Testing - Spring 2021

In the Spring semester, we focused on learning how to program the RP2040 and understanding the capabilities provided by its architecture. Through this process, we explored the ease of interfacing and programmability of the microcontroller using the C SDK for the Pico, including the development, build, and compilation processes for running projects. Additionally, we explore the hardware capabilities of the microcontroller through multiple small projects, leading up to testing of the limitations of the microcontroller with a gravitational n-bodies simulation drawn on a VGA screen.

Programming the RP2040 Using C

When developing programs to execute on the RP2040, users have the option of utilizing either Micropython [1] or C [2] to develop their programs, with an SDK available for each of the languages. Micropython is a full implementation of Python3 that runs on embedded hardware such as the Pico. Compared to programming in C, Python typically allows for faster development by the nature of the programming languages. While the Python SDK provides additional functionality to access some low-level functions specific to the RP2040, there were two concerns with the use of Micropython instead of C. First, compared to the full functionality provided by the C SDK for the RP2040, the Micropython SDK only provides partial control over the low-level hardware. We want to fully exploit the hardware provided by the microcontroller; development in Micropython would hinder this by limiting access to some of the low-level hardware specific to the RP2040. Second, the graduate course ECE5725, Design with Embedded Operating Systems, currently utilizes the Raspberry Pi 4 throughout the course. Program development in this course is centered around using Python, while the Designing with Microcontrollers course currently utilizes C to program the PIC32. To ensure students are exposed to different programming processes with platforms across different courses, it would also be beneficial to continue to use C programming in ECE4760.

While there are clear motivations for programming with C instead of Micropython, there are some challenges also present in programming the RP2040 using C. Part of programming in C involves project compilation. The C "Getting Started" guide [3] outlines the process for setting up an environment to compile projects across different operating systems. We have only experimented with programming on Windows machines, leaving programming on MacOS

devices untested. The process will be described in the software guide section. After setting up a project for compilation, a simple "make" call generates a .uf2 file containing the program to be executed. The program can then be loaded onto the RP2040 by holding down the program button on the Pico plugged into the computer while dragging and dropping the file in the file system. The difficult part of the process is the initial setup of a project, particularly when including multiple libraries containing different files and dependencies from each other. The project compilation setup is done through the use of CMakeLists.txt files located at the top of each directory within the project. These files include multiple configuration settings that can be necessary, including but not limited to recognizing sub-directories, creating link libraries, including link libraries, importing the C SDK into the project, and controlling UART and USB output. While they become more manageable with example projects to compare against and experience from creating new projects, this is one of the more challenging components when programming the RP2040 in C for any project.

Low-Level Architecture of the RP2040

Before diving into programming with the RP2040 and testing different components of it, we wanted to understand the hardware that was available within the RP2040. In Figure 1, the block diagram provides a simplified view of the available hardware. The most notable differences from the PIC32 that we found could be interesting to explore in a course setting were the Programmable I/O (PIO) state machines and the availability of a second processing core for multiprocessor execution. Typically, the method for running multiple tasks using the PIC32 would be through time-sharing between threads using the protothreads library running on a single-core CPU. However, with a second core available, it is possible to divide the processing for tasks between the cores for parallel execution, improving performance in such situations. The PIO state machines on the RP2040 allow for small programs to be written that control the outputs of GPIO pins without the need of any CPU processing. These can prove quite useful when performing simple repeated tasks by offloading the processing from the processor, further improving the amount of parallel execution from the microcontroller. These are used when we explore driving a VGA display using the RP2040.

Though both the PIO state machines and second processing core are beneficial improvements provided by the RP2040 hardware over the PIC32, we decided to focus primarily on the multiprocessing capabilities and limitations of the RP2040. Multiprocessing already appears across multiple ECE technical classes. However, with the different compilation process for programs on the RP2040, we found that we were unable to use typical libraries such as pthreads and OpenMP for typical multiprocessing. Having access to multiprocessing for student projects can allow for additional creative freedom in students projects if the Pico were to be used, having dedicated cores for executing specific tasks. As a result, we sought to gain a better understanding of how to make use of multiprocessing on the RP2040 and how existing multiprocessing programs using pthreads and OpenMP can be converted to be executed on the RP2040.

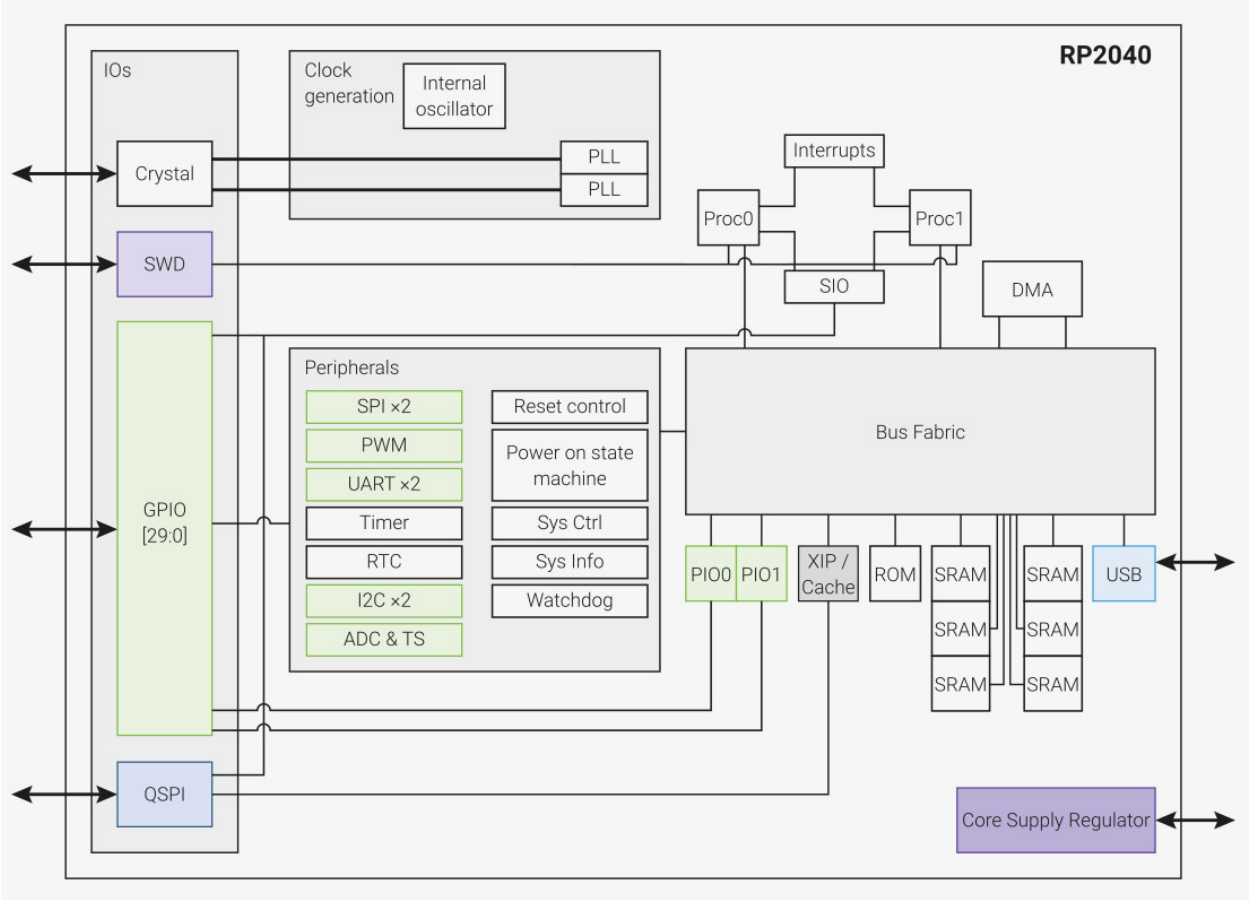


Figure 1: Block Diagram of the RP2040. For controlling connected peripherals, the RP2040 has 2 UART, SPI, and I^2C controllers, along with PWM controllers and a 12-bit ADC. Additional features of interest as shown include the two-processor configuration, PIO state machines, the Direct Memory Access (DMA) controller and the USB controller, as the Pico has a microUSB port available.

Exploration of Multi-Processing

To first understand how to set up a project and execute code on the second processor, we started by looking at the example program for the Pico [4] using the multicore library. The parallelization setup on the RP2040 has some differences compared to using other parallelization libraries in the structure of the code when setting up parallel execution. Typically, a process is launched and returns back to the main process upon completion. On the Pico, the programs are structured to launch the second core using an initialization function as shown in Figure 2 below. The entry function for launching the second core for execution is similar to a typical main entry function, which does not support any additional arguments. In contrast, pthreads sets up a thread to execute a program while additionally supporting arguments that can change the execution configurations. Despite the lack of support for arguments, however, simple multiprocessing programs using pthreads should be easily adaptable to execute on the Pico.

```
// Launch execution on the second core
multicore_launch_core1(core1_entry);
```

Figure 2: Entry Point for Second Core for Multicore Execution. The function passed into the entry point for the second core is formatted just as how the main entry point of the program for the RP2040 is formatted. If the tasks performed by the two cores are discrete, then this line is similar to executing the program outlined by core1_entry() on a second microcontroller instead while acting on the same set of GPIO pins.

With both cores having access to the same set of GPIO pins, both cores can attempt to control the output of the GPIO pins at the same time. We decided to test whether any issues arise from having both cores attempt to control the internal LED at different rates in the ex_multicore project[5], as shown in Figure 3. Our hypothesis was that the most recent core to send a GPIO command would "hijack" the previous command from the other core. In our test program, both cores concurrently run the loop that sends the GPIO commands to set the pin operations. Upon execution, we found that the behavior worked as we expected by having the last instruction to execute be the value visually observed. However, it remains unclear what happens when both cores attempt to set the same pin to opposite values on the same clock cycle.

```
// Drive LED at a fixed rate
while(1){
    gpio_put(LED_PIN, 1);
    sleep_ms(250);
    gpio_put(LED_PIN, 0);
    sleep_ms(250);
}
```

```
// Drive LED at a fixed rate
while (1){
    gpio_put(LED_PIN, 1);
    sleep_ms(1000);
    gpio_put(LED_PIN, 0);
    sleep_ms(1000);
}
```

Figure 3: Driving of Same Pin in Multicore Execution at Different Rates. The two code blocks above run in parallel on each of the respective cores, creating a looping pattern over the longer loop.

One of the core concepts in multicore programming is the distinction of variables local to a single process versus shared variables between multiple processes. Typically for pthreads programs, shared variables are managed by pre-allocating memory for variables to be shared and having both processes contain the pointer to the variable in memory. However, we found that the behavior of sharing variables is different in the Pico in the ex_count [6] project. Instead of operating within their own individual scopes, the second process launched as shown above in Figure E continues to operate in the same scope and maintain a shared view of global variables in the code.

Within this project, we proceeded to investigate how unprotected operations on these shared values affected the observed results on each of the cores. Two global integers were

incremented at the same time on both cores, with one declared as a pointer and having memory explicitly allocated for it, while the other was directly declared. No synchronization structures (locks, mutexes, semaphores, etc) were utilized to protect these variables, so we expected to have variable sums returned back on each iteration. However, the results we observed on the printouts varied for different configurations based on what values were printed and incremented. Additionally, the printouts stabilized after an initial set of iterations on the loop, which went against the expected results. However, the observed results were usually incorrect with both cases where only one core has the correct result and others where neither core yields the correct value. From this project, we confirmed that shared variables should always be protected with proper synchronization primitives to ensure the correct results are read despite the shared global scope.

We additionally explored how accessing a shared array in multicore execution worked within the Pico using the ex_array project [7]. One of the benefits of multicore is the ability to improve execution time by performing tasks in parallel when they are independent of each other. To simulate performing an operation on each entry of the array, the two cores were set to access alternating entries from each other. Comparing the execution of the array access in parallel against operation on a single core, we observed behavior slightly different from what was originally expected. When running only a single core, we saw a faster access time than executing in parallel and splitting the work between the two cores. However, if an arbitrary load (spin loop) was placed on the second core while running the array benchmark on an individual core, the execution time of the individual core nearly doubled. This suggests that some performance impact could be observed as a result of executing additional tasks in the second core. Additionally, the speedup observed for parallel execution was relatively far from the ideal 2x speedup when running in parallel compared to the single core execution with the second core running operations. One of the potential issues with multicore execution performance is cache and memory consistency impacting the observed performance. We tested accessing the array at variable strides to check whether any caching behavior could be observed. The performance of the array access remained constant as the stride increased at lower stride values, matching the expectation of no caching behavior since the RP2040 suggests that no data caches are used.

While time sharing can be replaced to some extent with multicore execution, we wanted to additionally explore the use of multicore execution combined with protothreads for programs where multiple tasks are involved and time sharing is still required. In the ex_protothread project [8], we set up scheduling of the same protothreads on both cores to examine the behavior of protothreads in the parallel execution. The example we used involved controlling two threads where one thread blocks and yields to another other thread for execution. Different patterns can be observed in the execution based on the scheduling configuration between the two cores, with some cases leading to the same thread executing on both cores at the same time. Further details can be found in the Github repository for the project. We suspect that this was the case because the context of the thread is stored within the protothread pointers. When both cores schedule the protothread to run, then both cores will start execution for the thread at the same starting point. Although it is possible to have time sharing run on both cores for threads

that depend on each other, different thread pointers should be used between cores to achieve more sensible behavior, though it may be possible to exploit the shared pointers between cores in some scenarios.

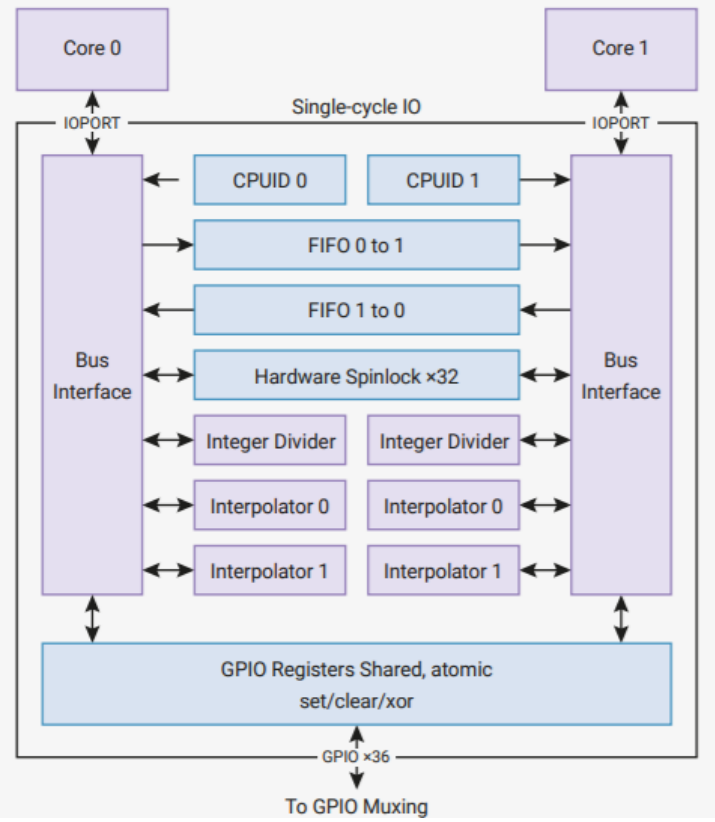


Figure 4: Communication between Cores with FIFO. One of the main synchronization structures that can be used between the two cores is communication through FIFOs between the cores. Through the use of blocking calls on the FIFO, barriers can be implemented in programs. Additionally, shared spin locks are available to protect critical sections as necessary.

When parallelizing work across multiple cores, synchronization structures are often required when operating on shared variables. We can see how the cores are connected to each other in Figure 4. In our testing of multi-processing on the RP2040, we created a gravitational n-bodies simulation where both cores were used to drive the simulation [9]. One of the key factors in this parallel execution is that all particles need to have their positions updated for the previous iteration before computing the next set of accelerations to be used for the following update. This requires a barrier in the program where both cores need to have reached the same point in the program. The barrier can be implemented by utilizing the blocking FIFO communication between the cores, as the cores will only proceed past the blocking calls once the message has been transmitted. Other programs could instead attempt to access the same shared variable at the same time, such as in the DDS example program done by Hunter Adams [10], where a global counter value is incremented on both cores. To protect this shared variable, a

spin lock is used to protect access to the critical section on both cores. This spin lock protects the critical section as expected, with minor caveats observed such as having to insert a short sleep statement after releasing the lock to prevent the same core from always acquiring the lock.

Overall, multicore execution on the RP2040 provides the typical benefits that would be observed as parallelizing a program using pthreads or a similar library. The RP2040 provides hardware to support synchronization between the two cores, along with the SDK facilitating the process of developing multicore programs. However, the structure for multicore execution on the RP2040 seems to be significantly more flexible, with a shared view of global variables and pointers that do not have to be explicitly shared, as well as communication over the FIFOs rather than through updating values in memory. Unexpected and therefore usually undesirable behavior can arise as a result of using the flexible structure for multicore execution without careful programming, as pointed out in some example projects. Additionally, a penalty on multicore execution compared to single core execution was observed, requiring extra considerations on when multicore execution provides a substantial benefit for program execution over execution on a single core.

Testing for Multi-Processing Performance

To evaluate the multiprocessing performance of the RP2040, we utilized an N-Queens program [11] and a gravitational n-bodies simulation [12]. The N-Queens program performs a recursive brute-force Depth-First Search to find the number of possible positions where N queens are placed on an N by N board with no queen in sight of any other queen. An example solution to the N-Queens problem is shown in Figure 5. Since this is a thoroughly-studied computational problem, complex and optimized solutions utilizing bit-masking exist that perform significantly better than our own implementation. However, our implementation has more straightforward and intuitive methods that one could reasonably expect students unfamiliar with the problem to use. The tested implementation does make use of the symmetry of the solutions across one axis to reduce the search space in half; however, this approach is utilized in both the serialized testing of the program along with the multicore execution of the program. Again, since the N-Queens problem is a well-defined problem and various resources are available online for the expected results, the correctness of the program could be verified by ensuring that the total number of solutions found matches the expected results. We decided that due to the highly parallelizable nature of this problem, it would be a good example program to evaluate the performance of multicore execution on the RP2040.

Parallelizing the execution of the N-Queens program was performed by assigning alternating rows to the two cores to perform the search over. This is a very simple and effective approach to parallelize the execution of the algorithm, as the only shared variable by the two threads is updating the total solutions counter. Typical of a pthreads implementation of this parallelization, this implementation accumulates the results of each of the cores once both cores have finished execution. A similar approach to pthreads is taken to start a second core executing the solver by having it allocate its own local variables to compute the solutions and call the solver utilizing its own parameters and local variables. Instead of allocating the variables for the

second thread of execution on the primary core, we can leave the setup to the second core after launching the second core and let the primary core start executing the solver. The shared FIFO between the two cores could have been used to sync them once the solvers finish execution to know when to accumulate the results of the two cores. However, for simplicity the current implementation returns the observed global values after each core returns, with the slower core to finish returning the accumulated result after both cores terminate.

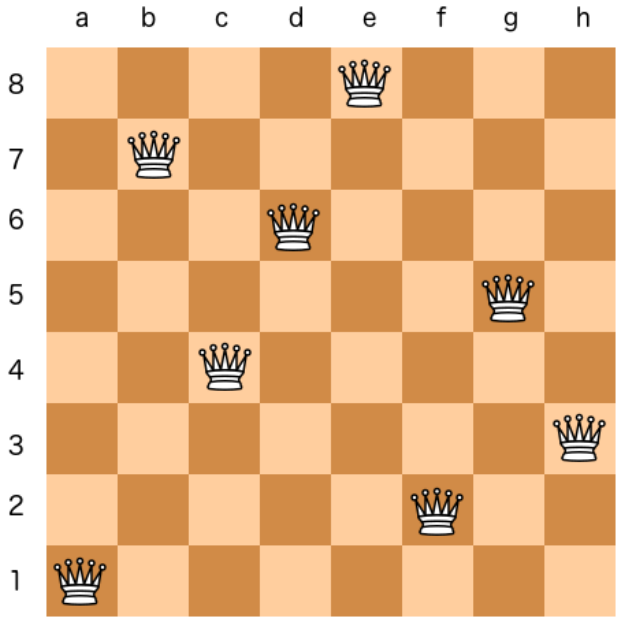


Figure 5: Example Solution for N-Queens problem for a board of size 8. The solutions for the problem can be verified by checking that no queens share a column, row, or diagonal with another queen.

The second program we decided to utilize to test the performance of multicore execution is a gravitational n-bodies simulation. The N-Bodies problem involves predicting the positions and velocities of n particles interacting based on Newtonian physics over time within a closed system. This means that particles interact with each other solely with gravitational forces, and no other force is applied to the system as shown in Figure 6. As a result, the expected total energy observed (gravitational potential energy and kinetic energy combined) should remain constant throughout the execution of the simulation.

$$f_{i,j}(t) = -\frac{gm_i m_j}{||r_i(t) - r_j(t)||_2^2} \frac{(r_i(t) - r_j(t))}{||r_i(t) - r_j(t)||_2}$$

Figure 6: Formula for Gravitational Force between Two Particles. In this formula, m represents the masses of the objects, r the positions of the objects, and g a gravitational constant. When applying this formula in the program, a constant is added to the difference in distances between objects to stabilize force values used.

The implementation of the n-bodies simulation utilizes iterative small time-step approximations of the movements of the particles according to Velocity Verlet integration. At each iteration, the velocity is updated based on the acceleration of an object, followed by

updating the position of the object based on the velocity. The acceleration for each object is then computed based on the positions of the objects, proceeding into the next iteration to repeat the same steps. To verify the total energy of the system, the velocity update is split into two half-steps, with the energy check performed in between the two half-steps. This provides a better measurement of the energy in the system by smoothing the effect of the updated acceleration over the following time step.

Similar to the N-Queens problem, the n-bodies program is able to be parallelized with work split across the cores to compute the forces on each of the bodies. However, the n-bodies program involves additional synchronization requirements that could limit the benefit gained from multicore execution. The only time that the information about all of the bodies is utilized by both cores is during the acceleration update step, where the position from each particle is compared to the position of all other particles to update the acceleration. As a result, a barrier must be placed before entering the acceleration update step on both cores. Additionally, to ensure that the positions of the objects do not change before both cores have finished execution of the acceleration step, a second barrier needs to be placed at some point before the position update happens.

Results - Spring 2021

Our development work this semester has been centered on creating small projects meant to test various software and hardware aspects of the Pico. After writing and running these programs, we have obtained both familiarity with the microcontroller as well as numerous key takeaways. We will summarize these in the following sections and justify our analysis with both quantitative and qualitative results.

Observations from Multicore Programming

After experimenting with the Pico's multicore execution capabilities through the various projects outlined above, we made several findings. While the project building process did not allow for the use of OS-based multiprocessing libraries such as pthread and openMP, we are still able to make use of the second core through the Pico's C SDK and hardware for synchronization across cores. The program structure for multicore execution on the Pico resembles the structure of a program written utilizing pthreads, with the main difference being that shared objects between cores make use of global variables rather than shared pointers to allocated objects. With a similar program structure to common parallel execution libraries, the RP2040 can be used in a course setting to effectively introduce students to parallel computing.

Compared to running parallel programs in a typical multicore processor, fewer concerns have to be considered when programming for parallel execution on the Pico. One such example is the management of cache coherence and how this can affect the performance of programs. When data caches are present, the access patterns to the data can affect how cache lines are made available to the cores, potentially resulting in an increased amount of memory transactions and negatively impacting multicore performance. This is not a concern since the

RP2040 does not utilize data caches between the cores and the SRAM memory, as seen in Figure 7 comparing the access times across different strides. As a result, students can just focus on handling and protecting shared variables, and achieving parallel execution where possible to improve program performance.

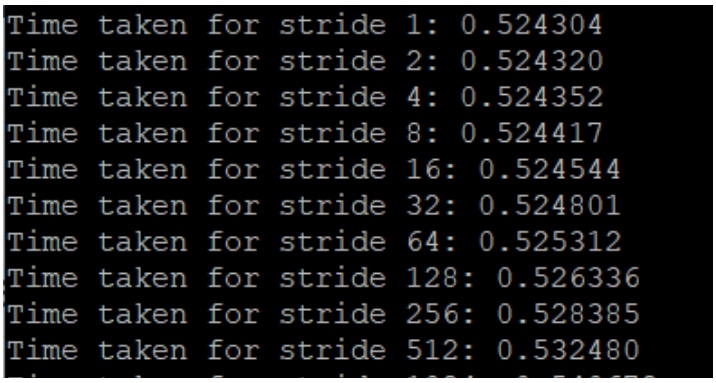


Figure 7: Access Times to an Array at Different Strides. By varying the stride length of the accesses to array entries, the number of cache hits and misses will change, impacting the access time. However, the access times remained similar across the different stride lengths, indicating no cache is present.

Another quirk of the different program structure for parallel execution is the update of global variables by both cores. When testing unprotected incrementation of a global integer on both cores, we saw strange behaviors in the program based on the variables that were printed and incremented, as shown in Figures 8 and 9. Flags are used to synchronize when both cores have finished evaluating the sum before printing the observed output, and under certain conditions the updates do not happen correctly. While this was an expected result, typically the observed sum of an unprotected variable is random. On the other hand, since the two cores are synchronized on the same clock, the output was consistent across multiple executions after the initial iterations. It remains unclear to us how quickly the updated value can be seen by the other processor. However, in this example configuration we found that if we increment both variables and print the results of both values, then the correct result was observed, as shown in Figure 10.

The strange mismatched view of global objects between the cores can be eliminated with proper synchronization and protection of shared variables, as seen in Hunter Adam's DDS program, where a global variable is incremented while protected by the spinlock. However, one of the typical performance limitations seen in parallel computing programs is the need for management of synchronization structures, often resulting in serialization of otherwise parallel sections of the code. For higher performance it is generally ideal to minimize the use of such synchronization when possible. From the different patterns observed in our experiments, we found that there are multiple instances where it is possible to have both cores update shared variables while yielding the correct result without these synchronization structures. This can be interesting for pushing the performance of certain applications by navigating through the unprotected access of shared variables. On the other hand, the observed incorrect results

similarly demonstrate the necessity of synchronization and protection of shared variables to maintain program correctness, serving as a teaching tool for parallel programming concepts where correctness needs to be ensured.

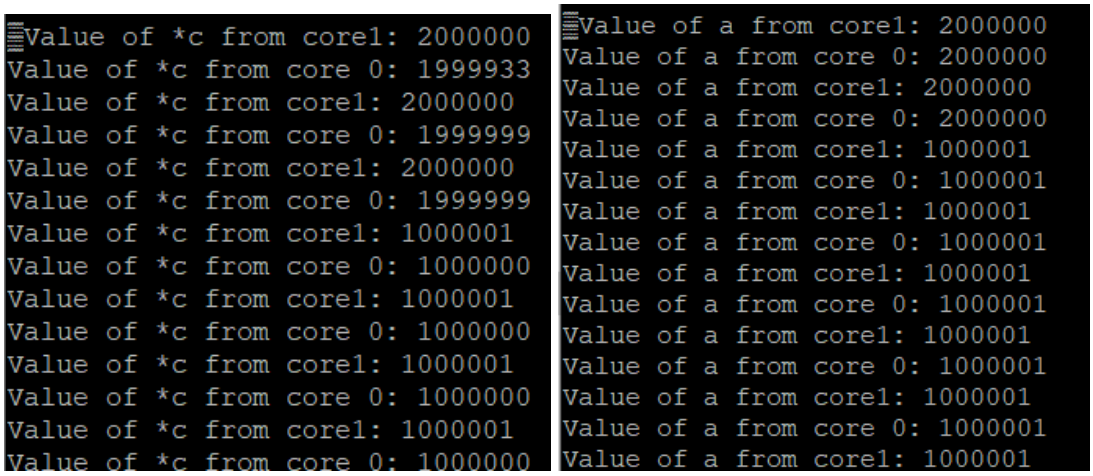


Figure 8: Increment of Global Integers While Printing Single Output. Whether the global value was declared directly as an integer or a pointer to an integer, incrementing both values by 1,000,000 on each core but only printing a single one led to the values not accumulating correctly when printed out.

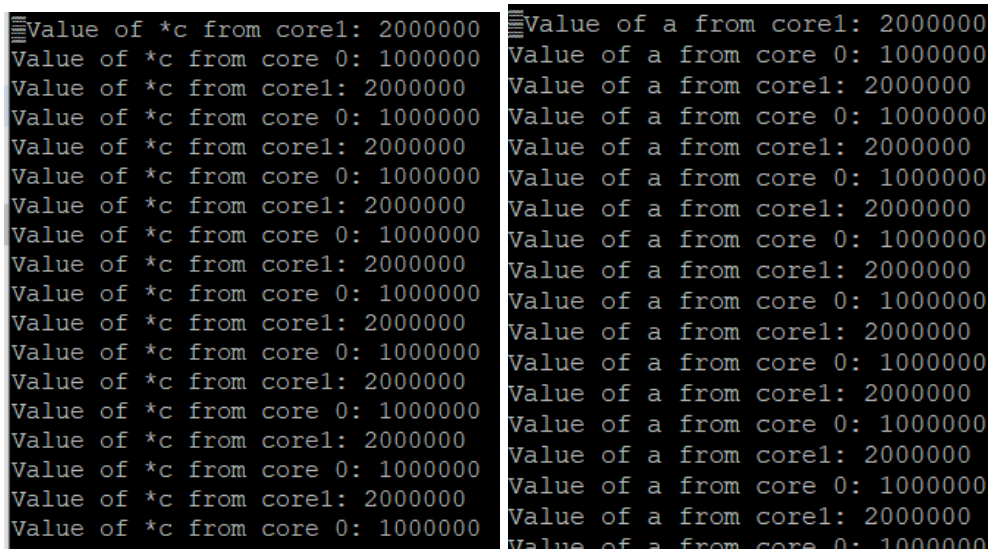


Figure 9: Increment of Single Global Integer While Printing Output. Operating instead on a single variable rather than two, incrementing the variable by 1,000,000 on each core, the correct result was observed only on one of the two cores.

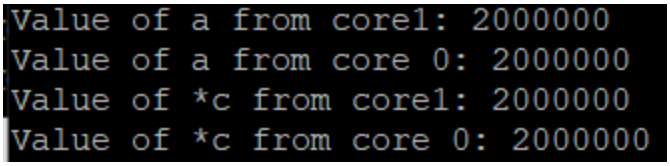


Figure 10: Increment of Two Global Integers and Printing Both Outputs. When both variables in the configuration were incremented by 1,000,000 each on each core and the results of both variables printed out to the terminal, the correct sum was observed instead.

Performance of Multicore Program Execution

Before exploring the programs used to evaluate the performance of the Pico, one detail we noticed about utilizing the multicore functionality of the RP2040 was that the use of the second core affects the performance compared to single core execution. As seen in Table 1, for performing a simple update of entries in an array serially was faster than utilizing both cores. This was potentially the case in this example as a result of iterating through a relatively short array, leading to the amount of work parallelized to be small relative to the overhead of setting up parallel execution. However, the key observation to note is the serial performance of updating the array when launching the second core to simply spin in a while loop during the serial array update on the first core. We found that the performance of the serial execution degraded significantly and roughly doubled in execution time. From this testing, we found that the cores can interfere with each other and cause degradation in performance. As a result, when running programs in multicore careful attention should be taken to ensure that programs do not perform useless instructions when possible to prevent such performance degradation.

Serial array update without second core	0.011015 sec
Multicore array update (two cores)	0.014958 sec
Serial array update while second core spins in while loop	0.021762 sec

Table 1: Multicore Performance Compared to Serial Execution for Incrementing Array Entries. The test involved stepping through an array with 32,768 entries and incrementing the entry by 2. Note that for multicore execution, timing starts before the second core is launched and ends once both cores have finished incrementing their corresponding entries.

To evaluate multicore performance on the Pico, we first compared the performance of an N-Queens program in serial execution versus parallel execution. For the serial execution of the program, the second core is launched but is set to sleep to reduce the impact on the performance of the core executing the N-Queens program. We decided to test using this program for N = 12 and N = 13, where the runtimes are longer as a result of the size of the search space. Looking at the collected results in Table 2, we can see that as expected the multicore version of the program outperformed the serial version of the program, achieving a speedup of over 150% compared to

serial execution for both sizes tested. While an improvement over the serial version is expected when running the program in a multicore configuration, the improvement was not as large as we had originally expected to observe. For a similar implementation of parallelizing N-Queens using the pthreads library and executing on a server multicore processor, the observed speedup was approximately 200% compared to the serial execution. Although the speedup was not as large as we originally expected, the performance of the program improved significantly and proves itself as potentially useful for compute-intensive projects in the future.

Nqueens	Serial Runtime (sec)	Multicore Runtime (sec)	Speedup
12	3.159503	1.8936	1.668516582
13	21.6433	13.8402	1.563799656

Table 2: Multicore Performance Compared to Serial Execution for N-Queens. The reported speedup is computed as the serial runtime divided by the parallel runtime.

After running the N-Queens program, we decided to also utilize the n-bodies simulation for testing multicore performance. While the N-Queens problem is highly parallelizable, the n-bodies simulation spends most of its time performing the acceleration update across all of the bodies. To make use of the second core, barriers have to be placed within the program to synchronize both cores to ensure the positions of the bodies are updated correctly. Analyzing the collected results in Table 3 and Figure 11, we can again see that multicore execution has improved the performance of the program significantly. Considering that a frame rate of about 30 frames per second (FPS) is adequate for a simulation, we can see that the serial execution of the program only managed proper execution up to 40 bodies in simulation. However, the multicore version of the program managed to achieve over an 80% performance improvement over the serial version and was able to animate 50 bodies with a frame rate of over 30 FPS. Unlike the N-Queens results, this increased performance in this case was more surprising when compared to a similar program. A similar version utilizing OpenMP to parallelize the acceleration computation for particles on a server multicore processor managed to achieve only about half of the speedup achieved in the RP2040. This may be the case as a result of the access pattern with caching affecting the performance of the program when using OpenMP. Additionally, the serial reference in this case does not launch the second core, thus no additional operations impede the serial execution of the program. While the achievable frame rate is lower compared to the server processors used in the desktop versions, the RP2040 achieves impressive performance while utilizing its multicore capabilities.

NBodies	Serial FPS	Multicore FPS	Speedup
20	122.1	221.68	1.815561016
30	55.38	102.11	1.843806428
40	31.45	58.45	1.858505564

50	20.24	37.62	1.858695652

Table 3: Multicore Performance Compared to Serial Execution for N-Bodies Simulation. The timing used for the calculation of frames per second (FPS) does not include the energy calculation step and print to the serial terminal. These operations are not important to the operation of the simulation, leaving only the core of the simulation.

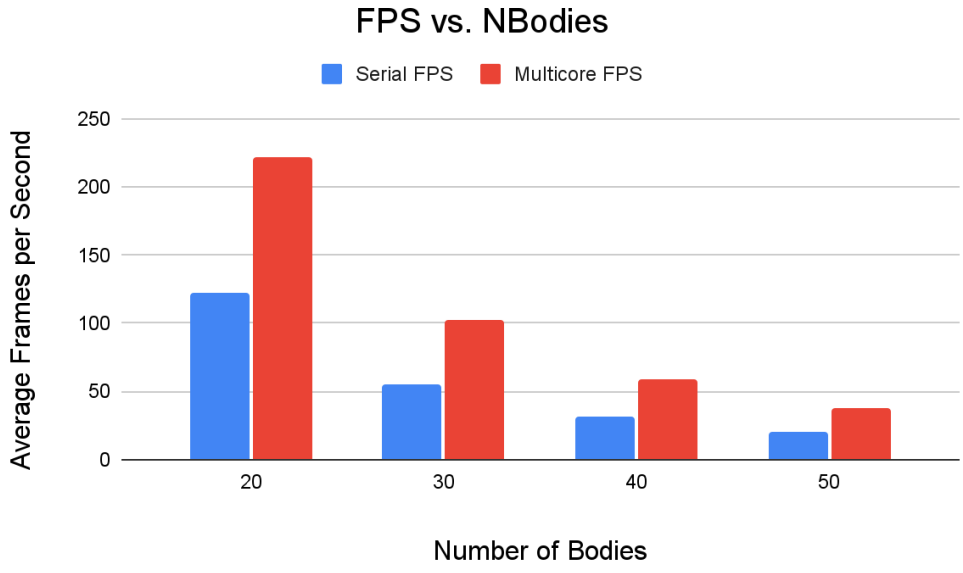


Figure 11: Comparison of Frames Per Second for Number of Bodies Simulated.

Visualization of N-Bodies Simulation on VGA Display

The last of our work for this semester involved displaying our n-bodies simulation on a VGA screen through the vga_n_bodies project [13]. We started with our simulation code, and incorporated Hunter Adams' sample program for drawing VGA using the programmable I/O (PIO) state machines [14]. To do this, three state machines are instantiated, one each for HSync, VSync, and sending RGB data. A pixel array corresponding to the screen dimensions is instantiated, and the contents of the array are sent to the RGB state machine through a Direct Memory Access (DMA) channel. With this method, the only function the CPU needs to perform to display to the VGA screen is modifying the data in the pixel array corresponding to what should be drawn on the screen; the DMA channel and PIO state machines completely handle the rest. This is an extremely powerful graphics tool, since generally microcontroller-based graphics programs need to dedicate a significant amount of processing power to maintaining a display. With PIO, not only can a display be maintained through using barely any computing resources, but also it can be done through VGA, a protocol that requires precise execution at 25MHz.

Figure 12: VGA Display of N-Bodies. Note that there tend to be many clusters of two bodies next to each other, since this simulation instance had a high gravitational parameter scaling with decreased distances. If two bodies approached each other at high velocity, the behavior would result in the bodies slingshotting away, while if approaching at low velocities, orbiting behavior would often be the result.

Within the actual display, a green square of pixels is used to represent each body, with the program running on single-core execution to produce 40 bodies within the simulation at a stable 30+ frames per second [Figure 12]. This demonstrates the efficiency of offloading the display to PIO, allowing the core to dedicate itself to running the simulation. While we did previously have multi-core execution for n-bodies, we chose to reserve the second core since we planned to develop a game-like interactive system with the simulation on one core and the rest of the logic on the second core. Unfortunately, we did not have enough time to fully develop this until the second semester, where we used the same concept to demonstrate our full system (see "Results - Fall 2021"). By this point at the end of the Spring semester, however, we achieved our goal of familiarizing ourselves with developing for the RP2040, interfacing with it through the Pico, and understanding the capabilities of the system to prepare ourselves for our work in the Fall semester.

Design and Testing - Fall 2021

In the Fall semester, we set out to tackle the problem of integration into the course curriculum. There were several key objectives that we had to accomplish to solve this problem. First, we had to design a prototype Printed Circuit Board (PCB) for the Pico. This board needed to serve a similar function to the existing board used in the ECE4760 course. Second, we had to develop a software library which would provide the resources necessary to interface with the on-board peripherals. Lastly, we wanted to demonstrate the capabilities of the board and associated software in the context of the course through a sample design project.

The Prototype Board

The key design requirement for the prototype board was to allow for a similar set of functionalities as the existing board used in the Designing with Microcontrollers course, known as "Sean Carroll's Big Board" (SECABB) [15]. The SECABB features a port expander, 12-bit DAC, TFT header-socket, programming header-plug, and power supply, in addition to interfaces for the GPIO pins offered by the PIC32. Thus, the first step before even attempting any PCB design work was to ensure that these peripherals, or similar replacements, could be driven by the Pico. The method we chose to do so was to use a breadboard, since it would allow us to hook up various peripherals quickly without the need to solder [Figure 13].

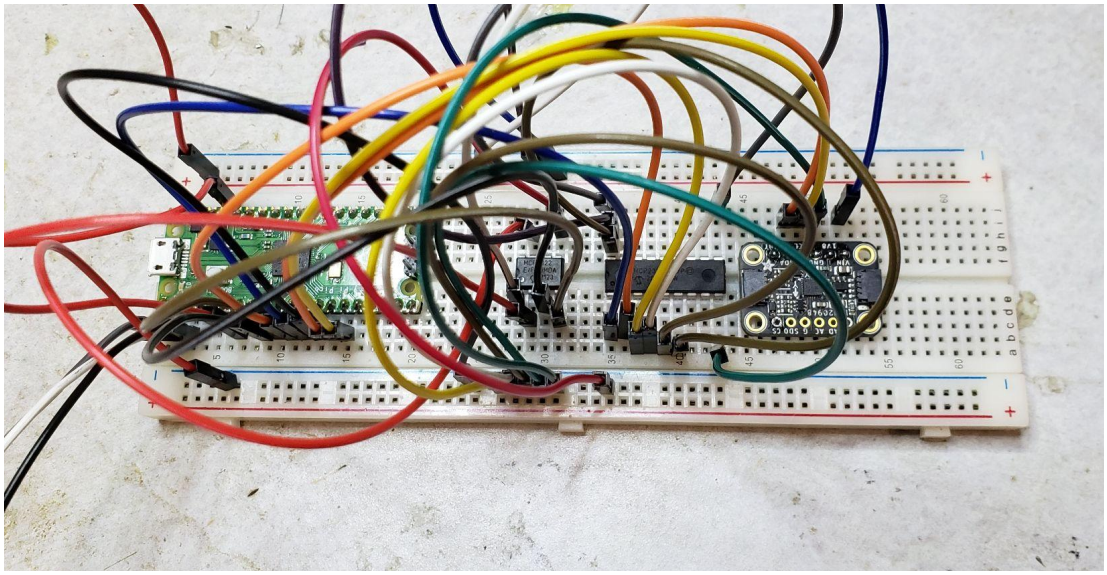


Figure 13: Breadboard Prototype. Used to connect and test various peripherals with the Pico, including (left to right) the 12-bit DAC, the port expander IC, and the ICM-20948 IMU.

For the DAC, we decided to use the MCP4822, a 12-bit DAC that uses SPI for communication [16]. This is the same DAC used on the SECABB, and we felt there was no need to change it since its resolution is sufficiently high for audio synthesis. The SECABB uses a 16 I/O-pin port expander, while we felt that the MCP23008 [17], which has 8 pins, would be sufficient for the Pico because there are already 29 potentially available GPIO pins that users can take advantage of on the RP2040. The additional 8 pins could effectively cover the pins consumed by other on-board peripherals, and if more are needed, the same I^2C bus used to communicate with the 8-pin port expander can be used to interface with up to 7 other external expanders, nearly guaranteeing that there will be no shortage of usable I/O. We retained the same header setup for the SECABB TFT display, though we decided that we also wanted to incorporate the Waveshare Pico LCD 1.14 [18] as an alternative display option. The benefit of doing so comes from the fact that the 1.14 LCD display attaches directly to the back of the Pico, giving students flexibility to remove the Pico from the prototype board if so desired and still have an option for a display. The final on-board peripheral we decided to implement was the

ICM-20948, a 9-DOF inertial measurement unit (IMU). This peripheral would add a host of sensing capabilities to the board, since it has a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer.

After deciding on the peripherals, we then had to decide on the pin assignments. The DAC used the SPI communication protocol, so we assigned GPIO 4, 5, 6 and 7 to use SPI channel 0. The IMU and the port expander both used I^2C; rather than placing them on the same bus, we used I^2C channel 0 for the port expander on GPIO 8 and 9, and I^2C channel 1 for the IMU on GPIO 10 and 11. SPI 1, labeled as the extra SPI channel, was allocated for the large TFT screen. Shown in Figure 14 is the corresponding pinout diagram.

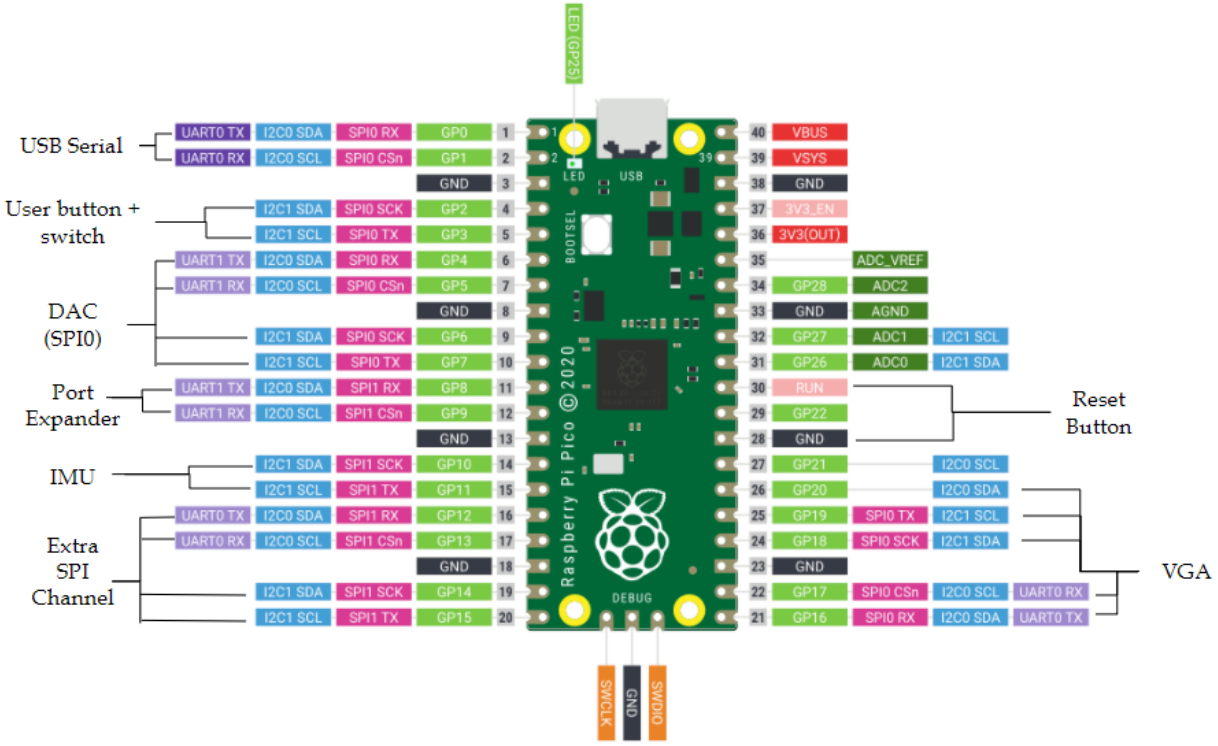


Figure 14: Pin Assignment Diagram. This demonstrates our first specification for the pin assignments mapping our planned peripherals to the Pico GPIO pins. Note that there still remain a number of unassigned pins to give room for other GPIO functions on top of what we plan to provide with the prototype board.

In addition to the aforementioned peripherals, a VGA interface, user button and switch as well as a reset button were implemented. The VGA pinout was based on a previous demo program written by Hunter Adams [14], in which GPIO 18, 19, and 20 were used for the red, green and blue VGA lines while GPIO 16 and 17 were used for the horizontal sync and vertical sync lines. We added a reset button since the Pico did not have a built-in reset, and instead relies on grounding the RUN pin (pin 30). A reset button is practical primarily for the ability to restart programs without needing to disconnect the power supply. An on-board button and switch can

also be helpful to allow students to interact with their programs, with a switch offering physical toggle capability, bypassing the need for additional software as a button for toggling would.

The final step before moving on to PCB design was to write a program that used the DAC, port expander, and IMU simultaneously. This was to prove that these peripherals could all work together with our pin assignments, which is necessary for verifying our circuit design before printing a PCB only to realize the idea was not feasible to begin with. We started by incorporating software for the DAC, for which we used Hunter's dual-core Direct Digital Synthesis program [10] to output two sine waves through the A and B output channels of the DAC connected to SPI channel 0. Figure 15 shows the initializations in the C program for SPI 0.

```
//SPI configurations
#define PIN_MISO 4
#define PIN_CS   5
#define PIN_SCK  6
#define PIN_MOSI 7
#define SPI_PORT spi0

// Initialize SPI channel (channel, baud rate set to 20MHz)
spi_init(SPI_PORT, 20000000) ;
// Format (channel, data bits per transfer, polarity, phase, order)
spi_set_format(SPI_PORT, 16, 0, 0, 0);

// Map SPI signals to GPIO ports
gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
gpio_set_function(PIN_CS, GPIO_FUNC_SPI) ;
```

Figure 15: SPI Channel 0 Initialization. This gives an example of fully initializing an SPI channel. The GPIO pin numbers are assigned to their corresponding SPI function, matching the previously given assignment. Afterwards, the SPI channel is initialized and the default gpio_set_function() method from the Pico standard library is used to explicitly set the pin functions.

We then had to initialize the I^2C channels for the port expander and IMU. Since the I^2C protocol is slightly more complicated than SPI, we wrote the helper functions `i2c_initialize`, `i2c_write`, and `i2c_read` to make working with the peripherals much easier. For an I^2C read operation, for example, the RP2040 has to first send the target device address and register on the corresponding I^2C bus before subsequently reading the response from the device. Packaging up this functionality into a single `i2c_write` function made our code to interface with these peripherals much easier to manage. For the port expander, our program simply set all 8 pins as inputs, and continuously read the status of the pins. This was given as an 8-bit value, with each bit corresponding to one pin; for example, a value of 255 meant that all 8 pins were set to high voltage (0b11111111), and a value of 3 signified only the first and second pins were high (0b00000011).

Initializing the IMU was a more complicated process. Following the datasheet for the ICM-20948 [19], it was not immediately clear what needed to be set and what could be left alone or assumed as defaults. We eventually discovered that there were multiple user bank register maps within the device, and that settings had to be changed across multiple banks. First, in bank 0, the power management register had to be written to reset the internal registers and restore default settings. Then, the following steps had to be performed: clear the power management sleep bit, swap to register bank 2, set the accelerometer sample rate prescaler, configure the accelerometer settings, then swap back to register bank 0. The IMU would be ready to send accelerometer readings once these initialization steps were finished. After incrementally adding, initializing, and testing the peripherals all together, we were confident that designing a PCB with these peripherals and using our pin configuration could yield a result that we wanted.

Designing the PCB

For the PCB design, we chose to use KiCAD as our schematic and layout design tool since we already had familiarity with the software. We thought it would be an effective strategy to take the pin assignments that we had settled on previously to easily generate a schematic, then subsequently translate the schematic into a PCB layout. Following are the steps we took to carry out this design strategy.

The first step in producing the schematic was to place the headers that would correspond to the headers on the Pico, since one of the design aspects of the board was for the Pico to attach onto it. A primary consideration was maximizing flexibility; for this prototype design, we wanted to give ourselves and other users as much freedom as possible when it came to using the Pico with the board. We incorporated this flexibility by planning to break out all of its pins externally on the board, so that every pin could potentially be redirected away from its intended peripheral (if any) and used as if the user had a direct connection to the Pico pin. With this in mind, we started the schematic design with four sets of 1x20 headers, two for attaching the Pico and two for breaking out each of the its pins [Figure 16].

Comparing this to the pin assignment diagram in Figure 14, it can be seen that we retained each of the peripheral pin assignments through the labels on each line. The next step was to connect the lines to their respective peripherals for the DAC, port expander, IMU, TFT display, and the user buttons and switch. The IMU and TFT display components only needed headers to attach the peripherals since they had their own breakout circuitry, while a different approach was needed for the DAC and port expander, which came in integrated circuit (IC) packages. For these two components, we decided the best method would be to use PDIP sockets that the components can slot into. With this design, attaching them to and removing them from the board becomes much easier and less committal than soldering them directly, which is especially useful if the IC fails and needs to be replaced. This also maintains our design philosophy of maximizing flexibility, since each of these peripherals as well as the Pico itself can be easily removed from the board if the need arises.

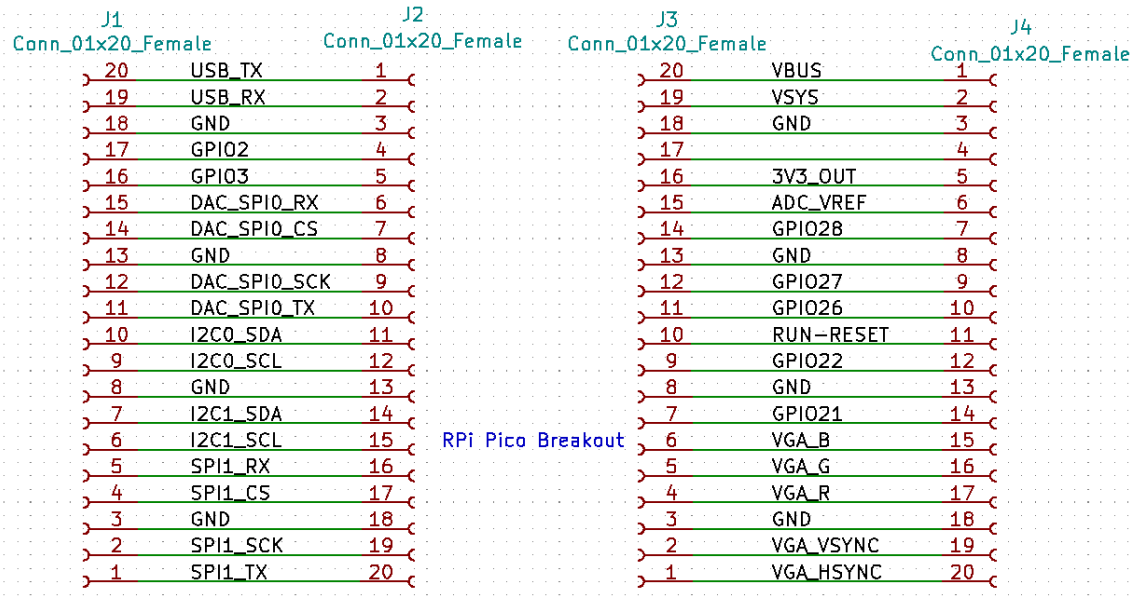


Figure 16: Schematic for the Pico Headers. The inner set of headers is meant for the attachment of the Pico, while the outer set is for breaking out each pin to provide access in the case that other peripherals want to be used instead of those provided on the board.

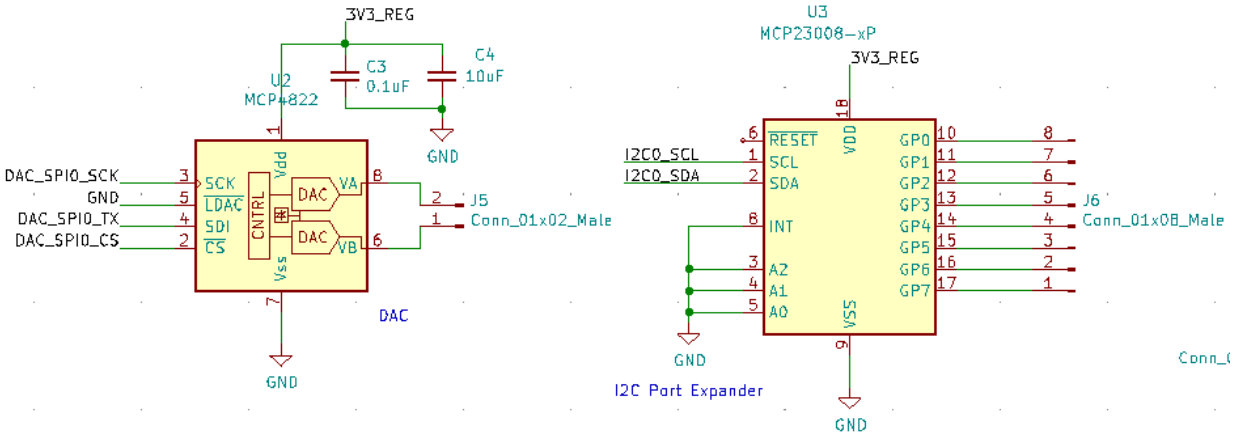


Figure 17: DAC and I²C Port Expander Pinouts.

.

Figure 17 contains the wiring diagrams for the DAC and the port expander. The A0, A1, and A2 pins on the port expander are the addressing pins, which in this design are held to ground. As previously mentioned more port expanders can be connected on the same I²C0 bus, in which case the other expanders' addressing pins would be set to different values so they can be individually addressed by the Pico. A row of 8 headers is attached to the expander's GP pins to give external access to the added GPIO. The user button and toggle switch, which are connected to the Pico's GPIO 2 and 3 pins, respectively, use external pull-down circuits to protect the GPIO pins from the current draw when the button is pressed or the switch toggled. The reset button is wired slightly differently, providing a direct connection from the RUN pin to ground which is

the requisite behavior to trigger a reset on the Pico. An image of the full schematic can be found in the Appendix [Figure A1].

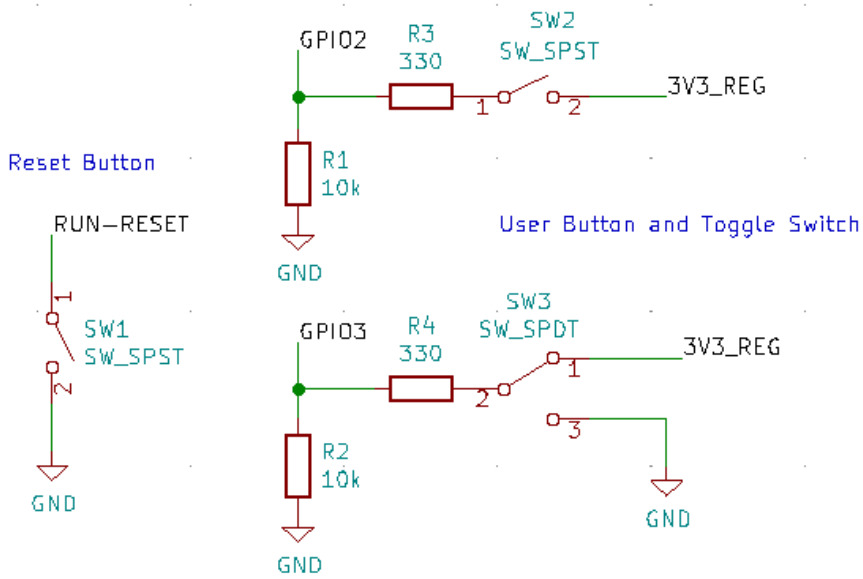


Figure 18: Wiring Diagrams for Buttons and Switches. Note the difference between the SPST button and SPDT switch, both of which employ a 10k pulldown resistor to GND on the GPIO connection.

After completing and reviewing our schematic design, including verification through the KiCAD Design Rules Checker, we were ready to move onto the PCB design. This procedure involved generating a netlist based on the schematic diagram, and then reading the netlist through KiCAD's PCB design tool. The first step after bringing in the netlist was to determine the size of the board. We wanted the Pico to attach to the board directly in the middle, and for the vertical height of the board to match the vertical height of the Pico to some extent. The horizontal width of the board had to accommodate the peripherals we included in the schematic, as well as room for the headers. An additional consideration we had to make was to ensure that the TFT display could attach without interfering with the other peripherals or the Pico itself. The final layout for the board was 3.57x2.14 inches, with header placements such that the TFT display would attach to the back side of the board, while the rest of the peripherals would attach to the front [Figure 19].

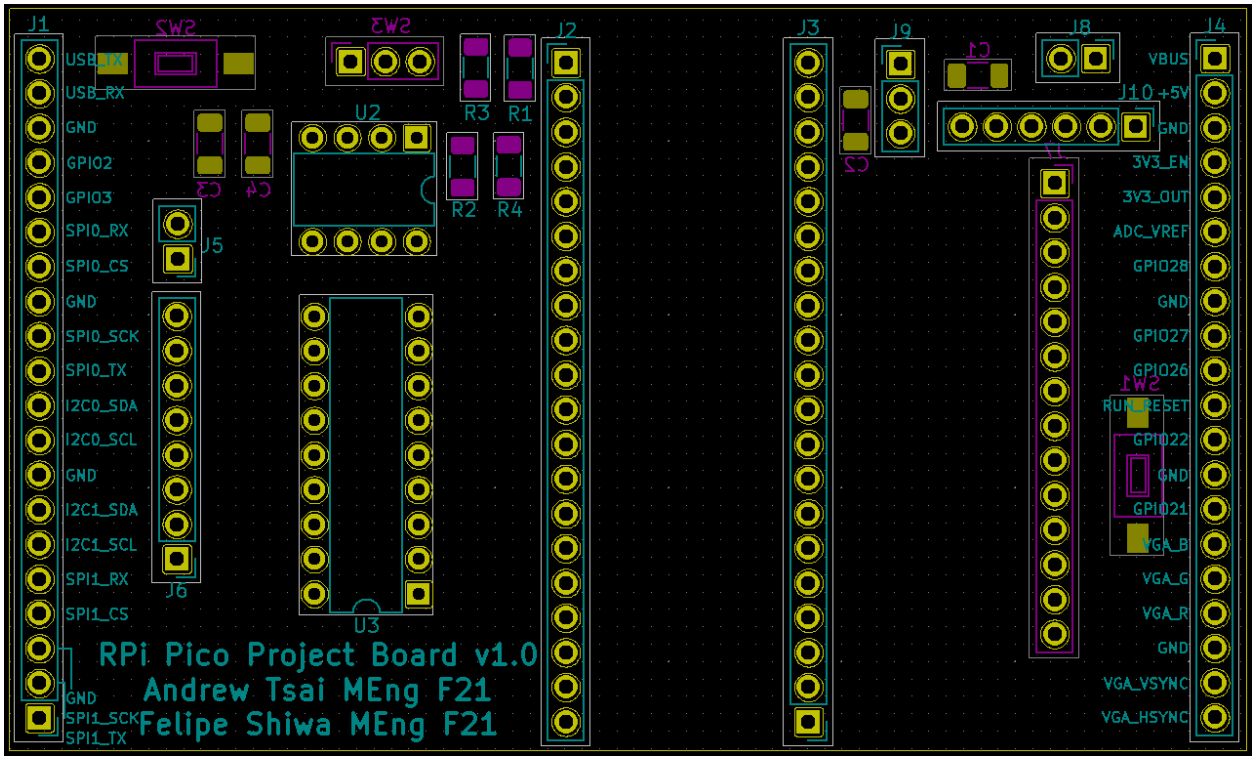


Figure 19: Layout Diagram. This view highlights the silkscreen and solder mask layers, with blue denoting the front side of the board and purple on the back. Note that the height of the board is barely taller than the row of headers for the Pico, demonstrating the matching height.

For the copper stackup of the board, we decided that 2 layers would be sufficient. This is due to the fact that even with the peripherals and headers on the board, there was ample room to connect signals from the Pico attachment headers to the breakout headers on the sides of the board and to the intended peripherals while also supplying power to said peripherals. Furthermore, the cost differential for manufacturing between 2-layer and 4-layer boards is often quite steep, so we wanted to make every effort to avoid paying that cost if possible. In the final layout, we placed the input power/ground signals on the top right of the board (header J8). The intended input voltage is 5V, which is the voltage connected to VSYS used to power the Pico and the IMU. We used a 3.3 volt linear dropout regulator (LDO, header J9) component to bring the intended input voltage of 5V down to the level used to other board components. That 3.3V line would extend to either side of the board to power the port expander, DAC, and TFT display. Equally as important as the power inputs, the ground net also needed to connect to each of the peripherals as well as multiple pins on the Pico and therefore the pin breakout headers. Figure 20 shows the regulated 3.3V and ground nets highlighted on the layout.

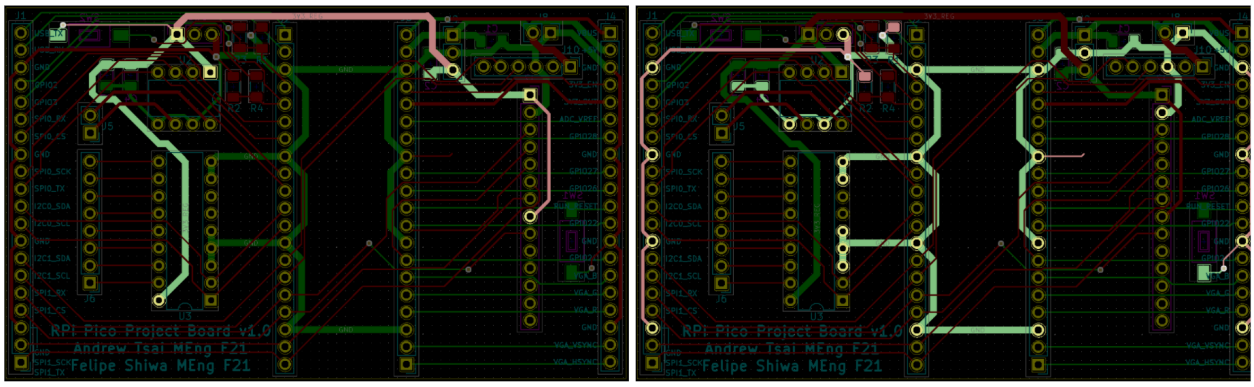


Figure 20: 3V3_Reg (left) and GND (right) nets. This view shows the power distribution across the board, with red nets corresponding to the top layer and green nets for the bottom layer. Note the usage of thicker traces along these nets to avoid any current bottlenecks.

There are several notable considerations we made when designing the layout for the board. We tried to keep the attachment headers vertical wherever possible so as to minimize the board footprint, since the height of the board would be fixed while the width was based on how we laid out our components. The main exception to this is the header for the IMU, which we placed horizontally so as not to risk a potential collision with the Pico. The power input (header for 5V/ground) is on the top right of the board so as to be in close proximity with the VSYS pin on the Pico. This placement also motivated the placements of several of the other components such that their input power pins would be as closer to the top of the board. This can be seen in the 3.3V line in Figure 20, where most of the connections are concentrated toward the upper half of the board except for the port expander's input voltage line. To improve the board from a user friendliness perspective, we labeled each of the header pins on the edge of the board with our assigned function if there is one, or the GPIO pin number otherwise. We also placed any user interactive components, such as the switch and buttons, on the back side of the board since that would be where the displays are attached, while the remaining components are on the front. This removes the need for users to continuously flip between the front and back of the board when in use, since the front of the board offers very little debugging information and thus a majority of the user interaction would be through the back.

With regards to design specifications, we used three different trace widths in this design: 0.25mm, 0.5, and 1mm. The thicker traces are primarily used on paths with higher expected current draw while the thinner traces are used for logic signals. For example, the power delivery traces for 5V/3.3V/GND are 1mm wide where possible, while all of the traces connecting the Pico GPIO pins to the corresponding breakout headers on the edges of the board are 0.25mm. 0.5mm traces are used where the component/trace density is higher and 1mm traces cannot easily be routed. Although the individual peripherals do not draw high amounts of current, the Pico can draw up to ~100mA, 2.5x that of an Arduino, so we wanted to avoid any risk of current bottlenecking in our design [20]. Using just one via size was deemed sufficient, with a size of 0.8mm and a drill size of 0.4mm. All of the footprints used are pulled from the standard KiCAD library, with one exception being the SW_SPST push button footprints. These were pulled from

the library used by Cornell's Autonomous Underwater Vehicle subteam, since we could not find the proper footprint for the component we wanted to use within the standard libraries. The resistors and capacitors use a 1206 footprint so as to match the size of the surface mount components used on the SECABB.

Testing and Revision 1.1

After placing and routing all of the components as well as passing the KiCAD PCB design rules checker, we finalized the design for the first iteration of our prototype board. We sent out the board to manufacture through JLCPCB, which has a fast turnaround time even internationally (~1.5 weeks from order placement and using express shipping). In the meantime, we continued with developing the software for the Pico in order to be able to test the board immediately upon its arrival. To do this, we put together a new C project called pico_master which would allow us to do a full systems test for the on-board peripherals. This would combine all of the test programs we used for the individual peripherals into a single C program, which would also be a baseline to help us with developing the software library that was one of our other main project objectives.

This program used I^2C on both channels to communicate with the port expander and the IMU as well as SPI for the DAC. Two sine waves would be generated at different frequencies using two different timer callbacks, one on each core, with the resulting output sent to the DAC's A and B channels. The two I^2C lines would be used in the main loop, where both the port expander, as well as the x, y, and z accelerations, would be continuously polled and printed out through serial. Upon attaching the components via breadboard and running the program, we immediately discovered an error with our PCB design, which was that in the schematic we shorted the INT pin on the port expander to ground, not realizing that that pin was an output rather than an input on the port expander. However, the rest of the peripherals were connected as we had them in the schematic and layout, and after leaving the INT pin ungrounded on the breadboard the program performed how we expected it to with all of the peripherals. The next test was to simulate the 5V input to power the Pico and the IMU, along with a 3.3V regulator for powering the other components. We obtained a 5V power supply from the ECE4760 lab which is also the supply currently used by the SECABB. We demonstrated that the Pico would accept 5V through the VSYS pin as the power source; however, the IMU proved to be more of a problem. We realized that connecting 5V as the power input, as we had done in the layout, would cause the IMU to not read the I^2C signals coming out of the Pico which were at a logic level of 3.3V. Thankfully, this would still allow us to test the IMU on the board itself simply by using 3.3V instead of 5V as the input since the VSYS pin has a range of 1.8V to 5.5V.

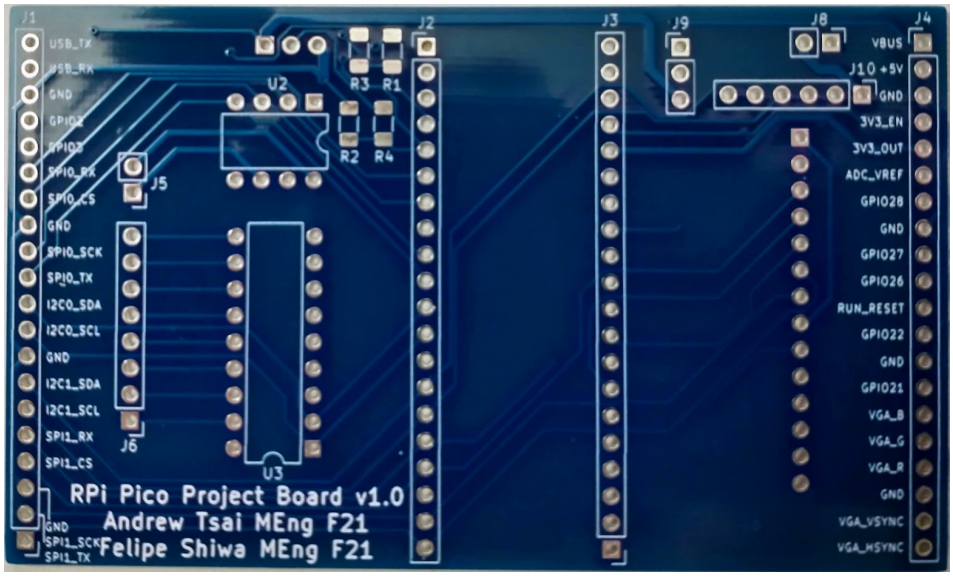


Figure 21: Prototype Board Photograph.

Figure 21 shows the first iteration of the board after arriving. The first step towards testing the board was to first populate what we planned on testing. This included the headers for attaching the Pico as well as for the power supply and IMU, the socket for the DAC, and the various resistors and capacitors associated with the components. After population, we attached the Pico and the rest of the components, and tested the board by powering the input with 3.3V. Although there were no issues through powering the board this way and the peripherals worked as expected, another issue that we discovered was that the pin assignments for the voltage regulator did not match the regulator we planned on using. The component we planned on using was the MCP1702 LDO regulator [21], and Figure 22 shows the footprint compared to the component pinout, where it can be seen that the input voltage needed to be on pin 2 while GND needed to be on pin 1. After realizing that the IMU's physical footprint extended over the headers on the right edge of the board and that we would have to slightly widen the board to the right, we had a majority of the list of changes we needed to make for the second design iteration of the board. Hunter also suggested that we place mounting holes on the board for standoffs, so that the board could be placed on a surface without resting on any of the onboard components. The last set of modifications came from usage of the 1.14 LCD display. This was a display that attached directly to the back of the Pico through sockets directly mounted on the display board [Figure 23].

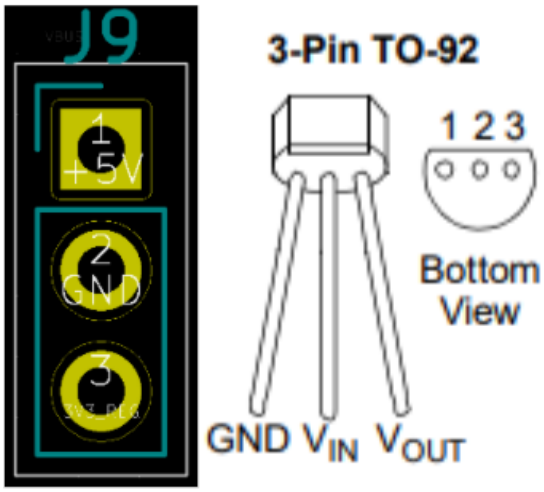


Figure 22: Erroneous Voltage Regulator Footprint. This view shows an error in the nets assigned to the header footprint. The solution we chose was to swap 5V with GND, so that GND would be on pin 1, 5V on 2, and 3V3_Reg on 3.

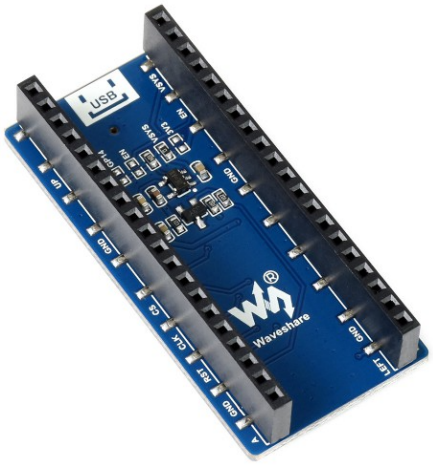


Figure 23: Underside View of 1.14 LCD Display. This view shows the Pico attachment sockets and how the pins used by the display are labeled [18]. We had to figure out a way to preserve this attachment method with our own board between the two devices, and the solution was to use double-sided headers with a socket for the Pico and pins on the other side for the display.

As such, it made use of several connections that we had already allocated for other functions. Namely, it used GPIO pins 2, 3, 15, 16, 17, 18, and 20 for the buttons and control stick on the front of the display, as well as the SPI1 channel. It used GPIO 8, 9, 10, 11, 12 and 13 for SPI-related functions. Although we had already allocated GPIO 12, 13, 14 and 15 for SPI1, the display still clashed with our assignments for the port expander, and IMU while the input components clashed with the user button/switch on GPIO 2 and 3 as well as the VGA display pins. Despite these issues, we figured that both the display and a VGA display could be driven if the inputs are disabled on the display, and that we could retain the functionality of either the

port expander or the IMU by moving one of the I²C channels to GPIO 26 and 27. For demonstration purposes, we prioritized the IMU since we figured the sensing capabilities would allow for a wider range of interactive display-based projects, and thus we reassigned I²C1 to 26 and 27. Furthermore, we realized that the buttons and control stick on the display were initialized to have the Pico designate their respective pins as internal pull-ups. To match our schematic with this behavior, we changed the user button/switch circuits to have an external 3.3V pullup circuit instead of a pulldown circuit which would otherwise cause power leakage with internal pullups set; this way our own buttons and switch could be connected to the same GPIO pins as ones used by the display without any negative consequences [Figure 24]. For board parts, we ordered special double-sided headers that had sockets on one side and pins on the other, in order to allow the Pico to plug in on the socket side and the display to be connected on the pin side. Refer to the appendix for a parts ordering list.

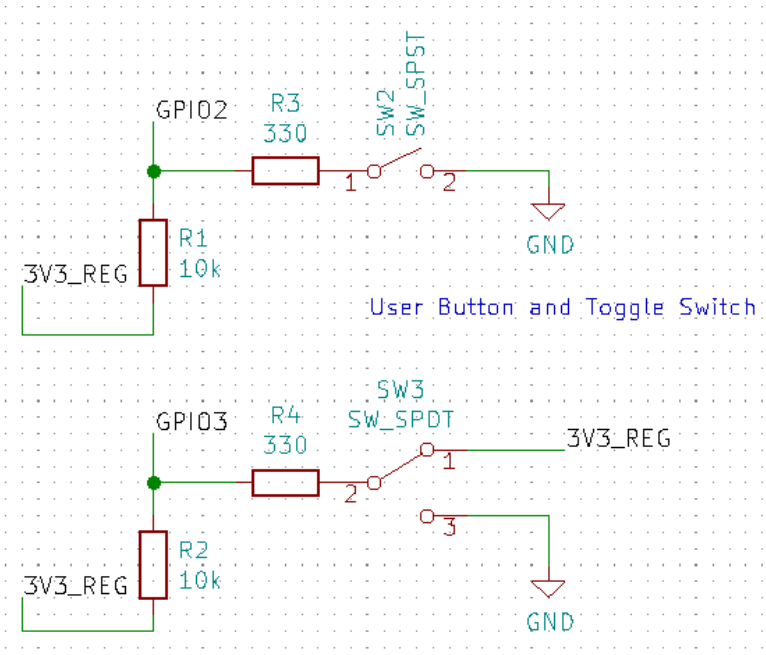


Figure 24: Revised Wiring Diagrams. Contrast the circuits with the ones shown in Figure 18 where now pullup circuits are used instead of pulldowns to accommodate the GPIO wiring on the display.

The changes brought about by working with the display were the last set of fixes that we needed to make for the second board iteration. To summarize, the fixes we made to the schematic and layout are as follows: leaving INT on the port expander unconnected, widening the board to accommodate the IMU footprint, changing the IMU power input to 3.3V, revising the pins for the MCP1702 voltage regulator, moving I²C1 to GPIO 26 and 27, switching the button circuits, and lastly adding mounting holes for standoffs. The full revision 1.1 pinout assignment and schematic can be found in the Appendix [Figures A4-A5]. The revised layout, labeled version 1.1, is shown in Figure 25. The previous dimensions were 3.57x2.14 inches, while the new width increased the dimensions to 3.75x2.14 inches. The component placement remains

largely the same. The voltage regulator was moved closer to the power input header and changed from a vertical orientation to horizontal, while the user button on the left was shifted down to accommodate the standoff mounting hole. The mounting holes are designed for M3-size screws.

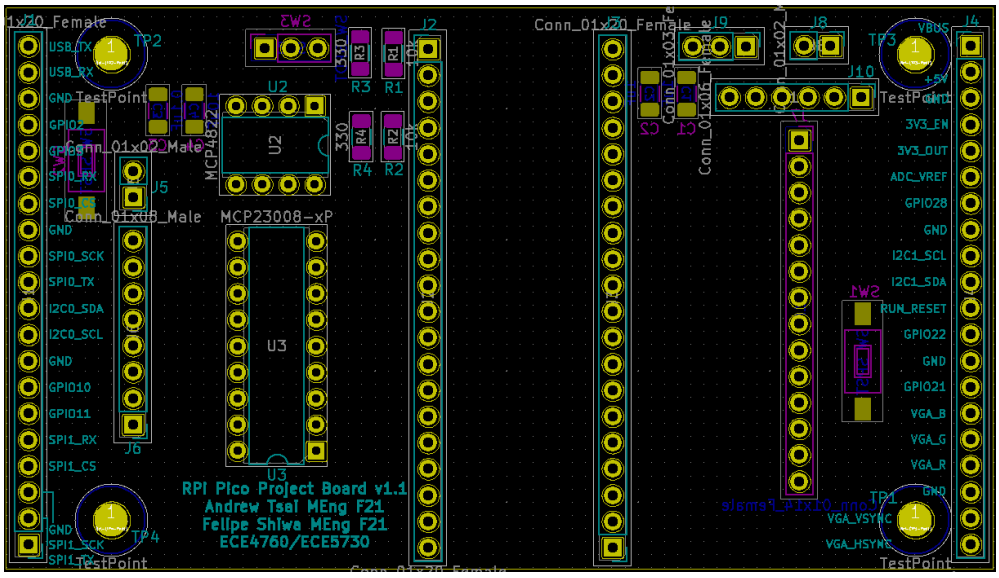


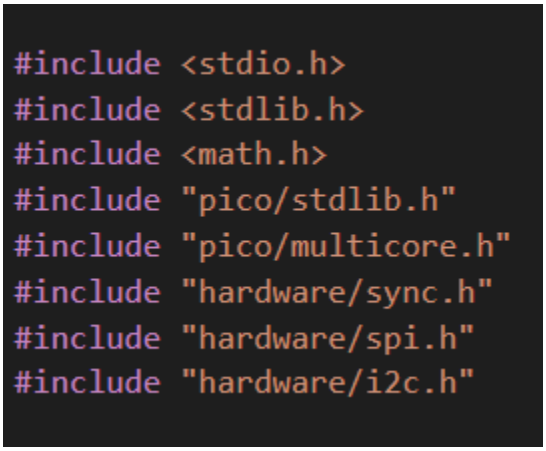
Figure 25: Revision 1.1 Layout. A layout view again highlighting the silk and solder mask layers on the front and back sides; note the changed positions of the voltage regulator and left-side user button as well as the added mounting holes for standoffs. More detailed views are contained in the Appendix [Figures A6-A7].

Developing the Software Library

Now that we had our hardware in a more finalized state, we switched our focus to the software. To achieve our goal of developing a proper software library for the board, we had two main areas of focus: a pico_board_lib library as well as the library of code for the 1.14 LCD display. The pico_board_lib would serve two main functions: consolidating the libraries to be included in projects using the board, as well as offering useful helper methods for initializing and communicating with the peripherals on the board as well as any external peripherals. The library for the display, on the other hand, would solely provide display functionality.

We based the code in the pico_board_lib largely off of the work we had done for the pico_master project, where we initialized and tested our peripherals. Currently, the pico_board_lib contains only a handful of functions: I^2C initialization, read and write, along with an initialization function for the complex startup procedure for the IMU. However, if development for the board is to continue, this library would serve as a flexible springboard for facilitating further hardware and software work, as key helper methods would continue to be added in the future. The library header file provides the other key functionality of this library, which is consolidating the other included libraries. This is shown in the code snippet in Figure 26. By consolidating the inclusions in this library, C programs that use this library would not

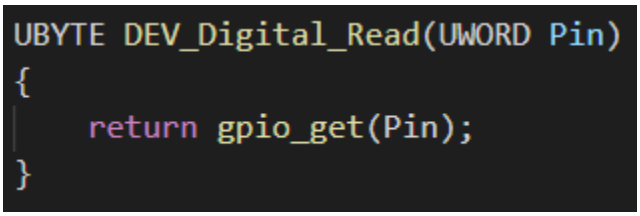
need to have the long list of inclusions at the top of the main project files, cleaning up code as well as potentially alleviating library path headaches during program compilation.



```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pico/stdlib.h"
#include "pico/multicore.h"
#include "hardware/sync.h"
#include "hardware/spi.h"
#include "hardware/i2c.h"
```

Figure 26: List of Included Standard Libraries.

Putting the display library together, on the other hand, was a much more involved process. We started with the functions in the software library given for the 1.14 LCD display [18, Resources - demo codes], and our goal was to modify its contents to be as transparent as possible. This meant students should very easily be able to take a function provided within the library and trace the code back to the standard Pico or C library calls. Initially, this was not the case, as there were multiple levels of files containing helper functions abstracting the library functionality to a higher level than we wanted. The given demo code used custom data types, renamed basic GPIO function calls, and had layers of encapsulation that we deemed unnecessary and which would make it difficult for students to trace the logic behind the code. An example of this is seen in Figure 27, where a gpio_get function call is renamed to "DEV_Digital_Read", and the custom UWORD and UBYTE data types (corresponding touint16_t and uint8_t) are specified as the input and returned data types. For our purposes, it would simply be clearer to directly use the gpio_get function.



```
UBYTE DEV_Digital_Read(UWORD Pin)
{
    return gpio_get(Pin);
}
```

Figure 27: Obfuscation of Basic GPIO Functions. We removed this and similar functions that unnecessarily wrapped basic methods that should be directly called.

There were numerous similar functions in the library which we deemed wholly unnecessary, as well as sub-directories with obscure functions that we also figured would not need to be included. We decided to incrementally work through the functions to gradually replace the functionality from the original library with our own functions utilizing the base

variable types and SDK functions for GPIO control. While the approach of gradual replacement of variables is generally encouraged to allow for incremental development, one of the issues we ran into was the setting of global variables used across the different files in the original libraries. Working around and gradually removing the global dependencies significantly increased the development time required to rework existing libraries, since to incrementally test functions that we replaced we needed to maintain the expected environment state and global variable values. After the initial hurdle of replacing the initialization steps for the display with our own library functions, the remaining functions were more simply adapted to no longer depend on the global variables and instead utilize defined constants from our own library.

At the end of the refactoring process, we condensed our code into a Config_Rewrite and a GUI folder, each of which only contains one C source file, one header file, and a CMakeLists.txt file. The Config_Rewrite folder contains the methods for initializing and sending commands to the display, such as SPI_Display_Init(), Display_Reset(), Display_SendData_Xbit(), etc. The GUI folder contains methods for modifying pixel arrays which correspond to the visuals drawn onto the screen, such as Image_DrawPoint(), Image_DrawRectangle(), Image_DrawString(), etc. We considered the combined functionality of the Config_Rewrite and GUI programs to be comprehensive enough for the intended purpose of the display, which is to offer a small visual interface for simple shapes and which could also print characters and strings for debugging information.

Results - Fall 2021

Now that we had our software library put together, we were able to test the revised 1.1 PCB as soon as it arrived from JLCPCB. The results were even better than we had expected, with the fixes that we made for the revision proving they were worthwhile. The appendix contains a side-by-side comparison between the rendered 3D-view in KiCAD and the manufactured board photo [Figure A8]. Although we were initially afraid that attaching the display would cause issues with the rest of the board, or vice versa where running peripherals could damage the display, it turns out that neither was the case. Even when the display was attached, the other peripherals still worked fine as long as they were initialized properly. The only drawback, which we previously discussed, is the issue of having to choose between the display or the port expander, and between the GPIO pins on the display or VGA functionality. However, we consider the design of the board a success in the context of our design goals. It can simultaneously accommodate the same peripherals as the SECABB, offers multiple display options, and provides full flexibility with the breakout headers on the board edges allowing users to completely bypass the hardware on the board if so desired. Shown in Figure 28 are the top and bottom views of the populated board, which demonstrate how the board matches the form factor for the Pico and integrates the peripherals, how the display attaches to the back side, and the effectiveness of the mounted standoffs at properly elevating the board platform.

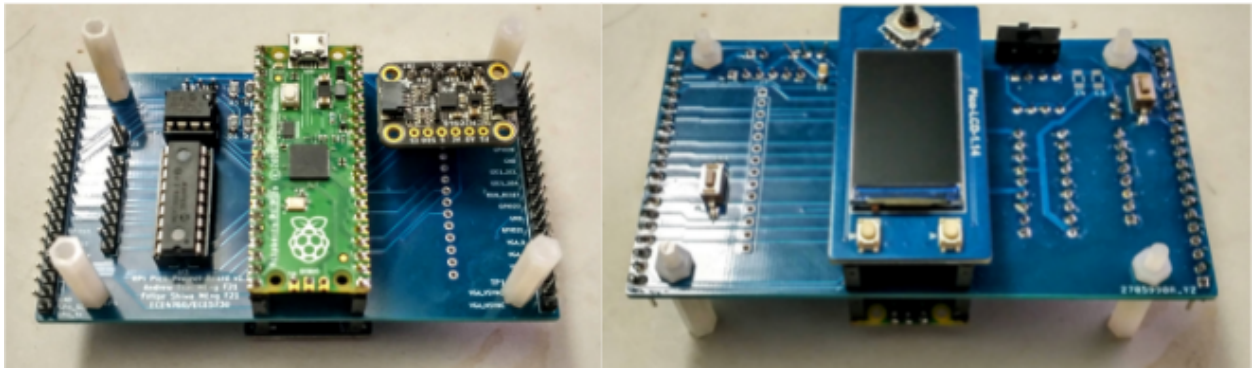


Figure 28: Bottom and Top Views of Populated Prototype Board.

With this board, students can be introduced to the same topics currently presented in ECE4760, such as the SPI and I^2C communication protocols, driving graphics, and embedded sensing. However, the board and the Pico in conjunction offer much more. The graphics capabilities are greatly expanded compared to the PIC32 with the possibility of using both the display and an external VGA interface. The IMU provides motion-based sensors directly on the board, adding potential for sensor-based control loops. The smaller form factor compared to the SECABB can also be a great benefit for student projects since the development platform is more compact and wiring can be easier to manage. The combination of this new lab board and the software library should serve as useful teaching tools for the course, while retaining a high degree of familiarity since the peripherals match what were previously used in the curriculum.

To further demonstrate the effectiveness and potential for the board to be used in the course, we designed a sample project to showcase some of the improvements this board brings to the table, including the multiple display methods, better processing power, the dual-core nature of the processor, and sensing capabilities. Building off of the N-bodies simulation program we developed last semester, we made an asteroids-like game where a ship navigates through a cluster of planetary masses, attempting to survive as long as possible without collision. We use the first core to run the simulation, animate the ship, and send image information to the programmable I/O state machines to display the game screen through VGA. Simultaneously, we use the second core to poll IMU sensor readings to control the ship and drive the mini display which acts as a HUD, showing the current control orientation, frame rate, survival time, and record survival time. While maintaining a stable 30 FPS, our simulation can contain up to 40 bodies while performing the mathematical operations for the simulation with the double data type, taking advantage of the double-supported floating point unit in the RP2040. Very few explicit optimizations are performed in our program; in other words, even with significant room for optimization, the demonstrated performance through the raw capabilities of the RP2040 is still impressive.

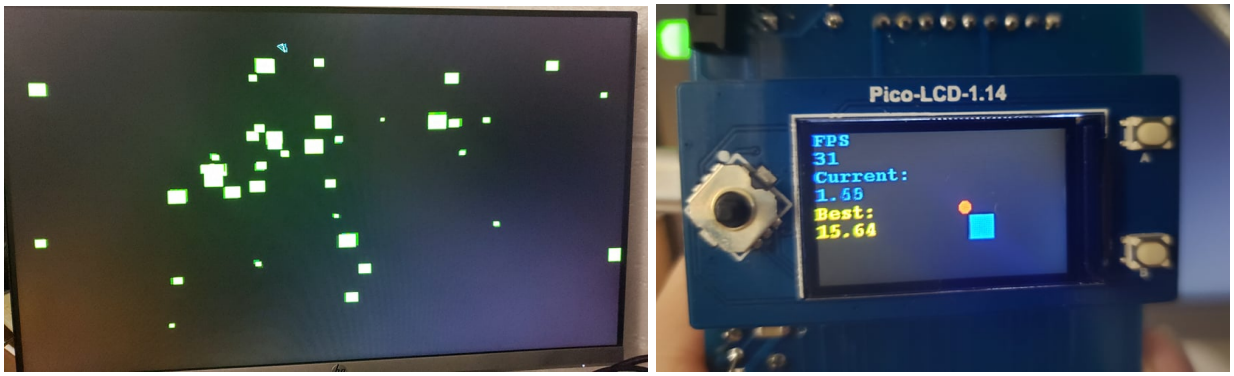


Figure 29: Dual-Display Demonstration, showing the n-bodies simulation on the left and the mini-HUD on the right, with the blue square and red circle showing the control orientation and other game state information shown on the left.

Conclusion

At the beginning of our project, we were tasked with developing a prototype board for the Pico so that we could subsequently assess its potential for ECE4760. After our development and testing efforts across the Spring and Fall 2021 semesters, we have finished developing such a board and have gained significant familiarity with the new microcontroller platform. Consequently, we believe we are sufficiently prepared to address the issues we initially laid out.

Programmability and Ease of Interfacing

Our work with the Pico began with setting up the development environment from scratch. Through this setup process, which we went through for both Windows and Raspberry Pi systems, we discovered the potential difficulties with properly configuring the C SDK, modifying CMakeLists files, and library linking. If these steps are not properly understood and done properly, it can be quite difficult for a new user to compile and run an out-of-the-box C project. That being said, there are numerous ways to alleviate this problem; our software library is the first. We consolidate a majority of the commonly used libraries within our pico_board_lib, and very clearly demonstrate how it can be linked to a project as well as how it links the remaining libraries. This resource is also flexible, because our intent with its design was extensibility such that any further development with our board or software can simply add on to the existing functionality within the library. Furthermore, many of the issues with setting up the C SDK can be taken care of simply through course setup. We have had discussions with Hunter and established that it would be beneficial to have the C SDK set up and running in the ECE4760 lab. We also include steps for setting up the SDK for Windows in the Appendix (see "Software User Manual"), and link to the starter guide which details setting up the environment for other devices. These resources should be more than sufficient for students to be able to set up the environment, write their own project software, and program the Pico both in and outside of the lab.

With regards to interfacing, the Pico standard software libraries already include a wide variety of functions for interfacing with both internal and external hardware. Through our experiences with using the programmable I/O, the dual-core system, internal timer interrupts, and other internal hardware on the RP2040, we conclude that the ease of internal interfacing is actually rather impressive. Many of the hardware functions can be invoked with a single function call, and the more complex operations are well-documented. For external peripherals, the host of communication protocols that the Pico can use, as well as the functionality of our own software library, can effectively cover a majority of the peripherals that would belong in a microcontroller ecosystem. Thus, we have ensured sufficient programmability and ease of interfacing with the Pico, the first of our three main design goals.

Hardware and Software Limitations

The next design goal we set out to accomplish was to discover the hardware and software limitations, and through the process of doing so establish a comparison between the RP2040 and the PIC32MX. In the case of hardware limitations, the first and foremost to be examined is GPIO. As thoroughly discussed as part of our design work for the Fall semester, a significant design issue we encountered was GPIO pin assignment. The full range of intended functionality for our board could not be realized since some pins needed to serve multiple functions at once, which is simply not possible. However, we still deem our current board configuration and pin assignments to be quite effective when examined through functionality subsets. For example, the ability to drive two different displays is extremely desirable, even if it comes at the expense of the on-board port expander and some additional GPIO. In fact, if additional GPIO is truly needed, the flexibility of our design allows for off-board peripherals, such as an additional port expander, to be connected and used simply by reassigning the I^2C channel pins. We decided to make tradeoffs like these in our design to aggressively implement as much with the Pico as we could while doing our best to avoid restrictive decisions.

The rest of the hardware and software limitations, especially pertaining to performance, have to be examined in the context of the existing course hardware. The software used with the Pico is by no means drastically different from the C software structure currently implemented in the course for the PIC32. In fact, we demonstrated that protothreads can also be used effectively on the Pico, and for the interrupt service routine functionality commonly used with protothreads on the PIC, the Pico's timer callbacks can be equally effective. Holistically, the much more powerful hardware on the Pico, especially when the dual-core processor is taken into account, clearly places its capabilities for both hardware and software above those of the PIC32.

Integration into ECE 4760

With our two previous questions answered, the last that remains is whether the Pico should be integrated into ECE4760. After all, just because it offers improvements over the current hardware does not necessarily mean it may be a good fit for the course. The cons of

modifying the existing curriculum need to be carefully considered. Of these, the most significant change is the overhaul of the existing code base. The current labs, and the course structure centered on the PIC32, rely on an extensive collection of existing sample programs and examples. Although they would still offer significant value for a new Pico-based course structure, much of it would need to be explicitly refactored to run on the Pico and demonstrate the same concepts. The course also currently uses the MPLab X IDE to compile programs and upload to the microcontroller; this would have to be replaced by another IDE compatible with the C SDK, such as Visual Studio Code, which requires additional setup and effort on the part of the course staff. Lastly, as is common when considering a new architecture, the planned course lab exercises would need extensive testing after being ported to the Pico since there may be many potentially new sources of error that would have to be documented and debugged to ensure a smoother student laboratory experience. Lastly, it must be considered that newer and better technologies will continue to be released for the foreseeable future, which can have an impact on whether the timing is right to switch to an improved microcontroller.

Though there are a number of hurdles that need to be overcome before the Pico can be properly integrated into the course, our work demonstrates that the pros of doing so can outweigh the cons. The capabilities of the Pico with the RP2040 microcontroller outshine those of the PIC32, and if used within the course, can be an effective modernization of the course content. The educational opportunities will also be broadened, with the platform able to introduce more advanced concepts within graphics, processing, and embedded communication protocols. Even still, there are many features on the microcontroller that the course curriculum simply will not have the scope to cover in depth, but students can take advantage of regardless. Examples such as programmable I/O and multiprocessing, which may fall outside the scope of current course material, still offer students greater flexibility when it comes to their course design project. There are currently many examples of students pushing various hardware capabilities of the PIC32 which are not taught explicitly in the course for their projects, and so expanding that realm of possibility is an exciting prospect.

We originally set out to answer the question of whether the RP2040 microcontroller can be a suitable upgrade to the PIC32MX. After working with and designing around the Pico board for two semesters, we can conclude that it has all of the requisite features for the course, and still offers much more. It is a definite upgrade to the PIC32MX in terms of performance, user-friendliness, and capability range, and therefore can bring significant educational value to the course's lab exercises and for student design projects.

Appendix

Diagrams

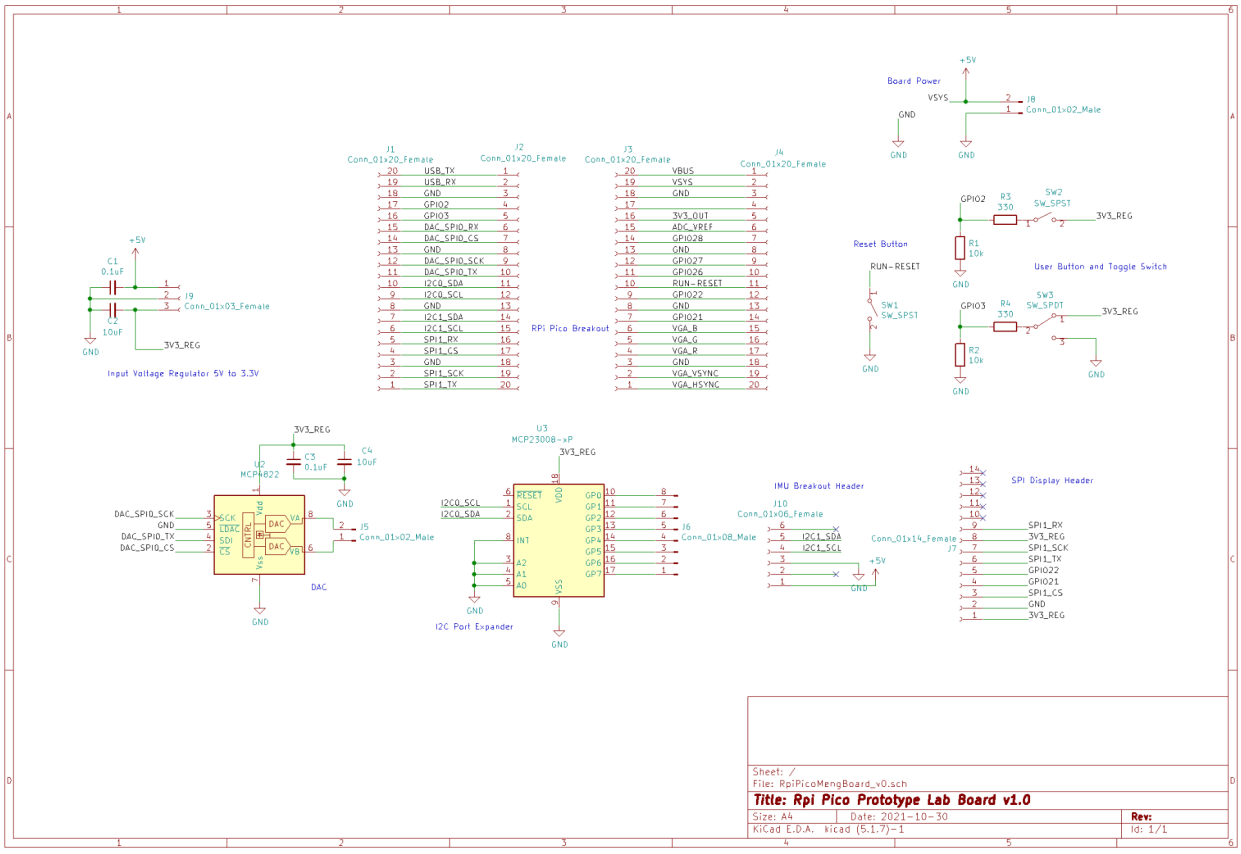


Figure A1: Version 1.0 Full Schematic

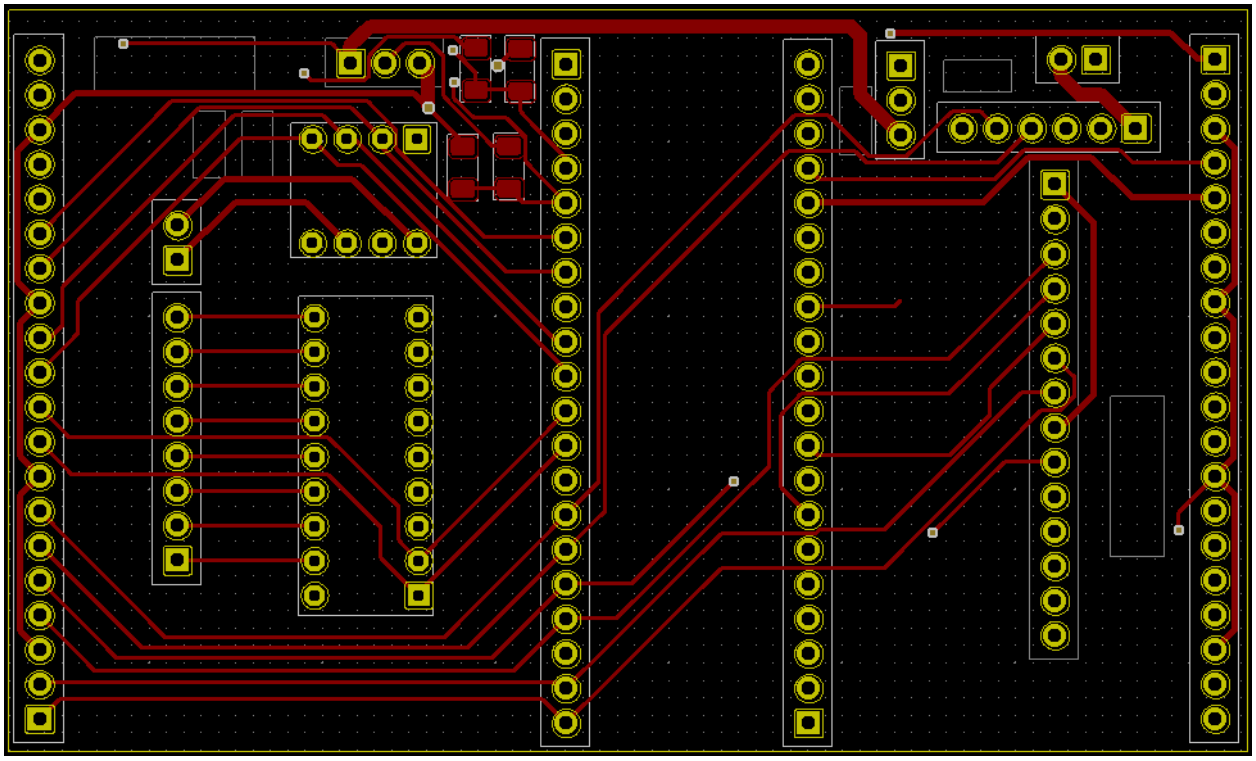


Figure A2: Version 1.0 PCB Front Copper Layer

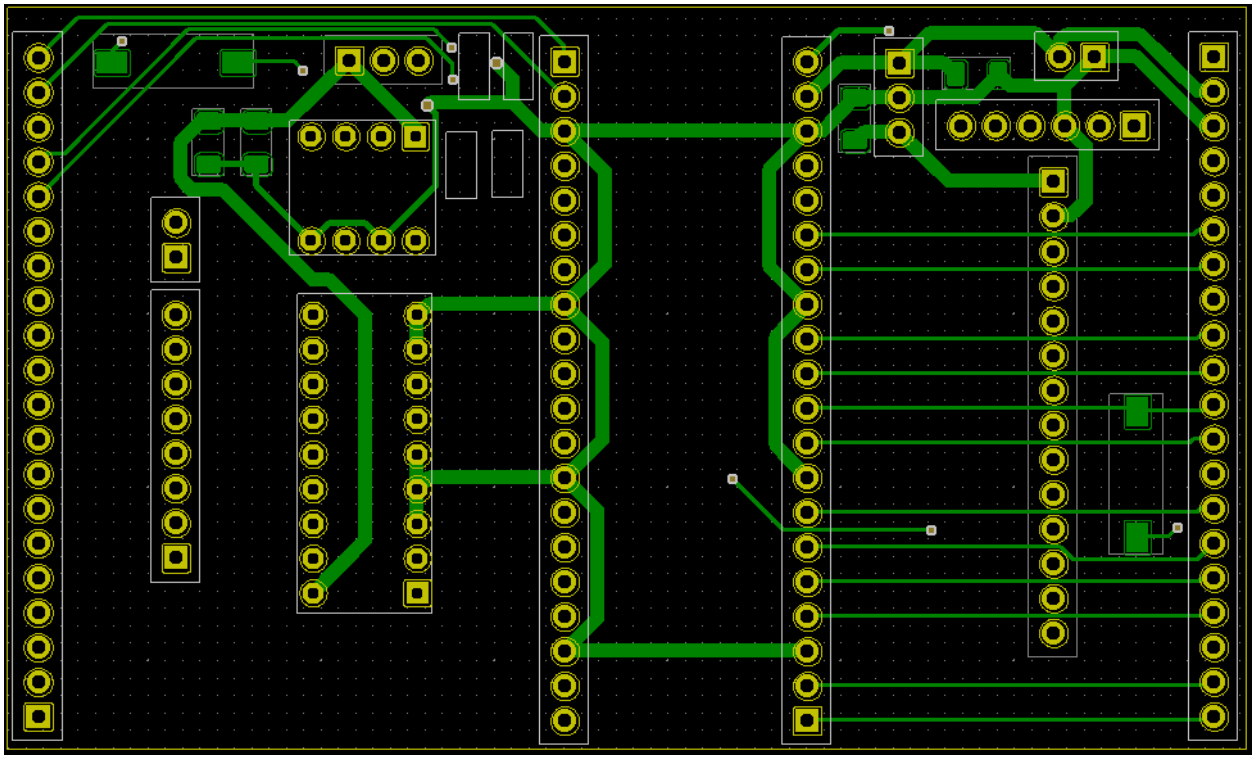


Figure A3: Version 1.0 PCB Back Copper Layer

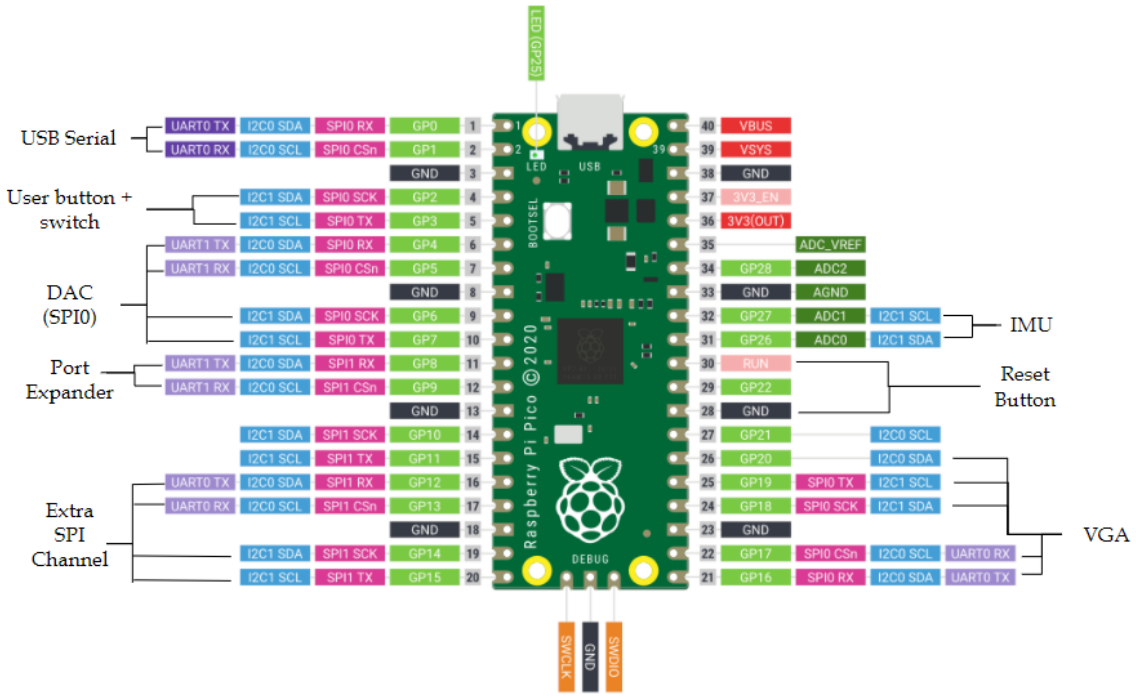


Figure A4: Revision 1.1 Pin Assignments

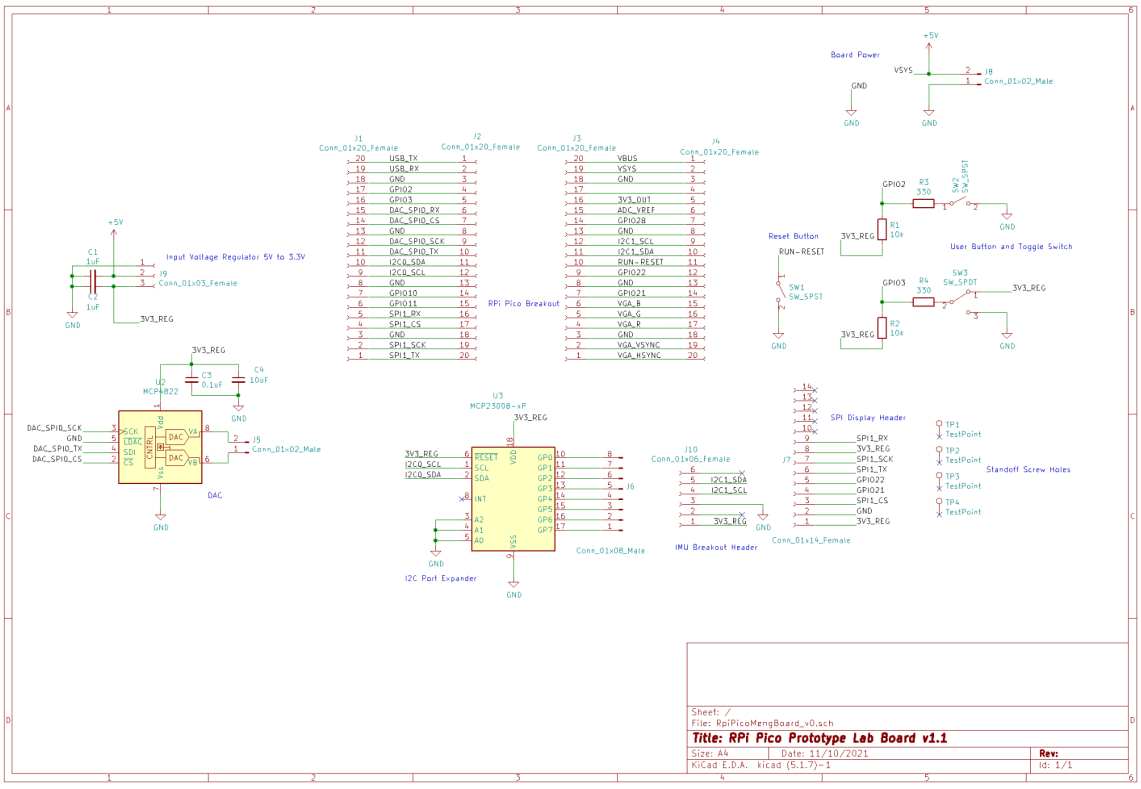


Figure A5: Revision 1.1 Full Schematic

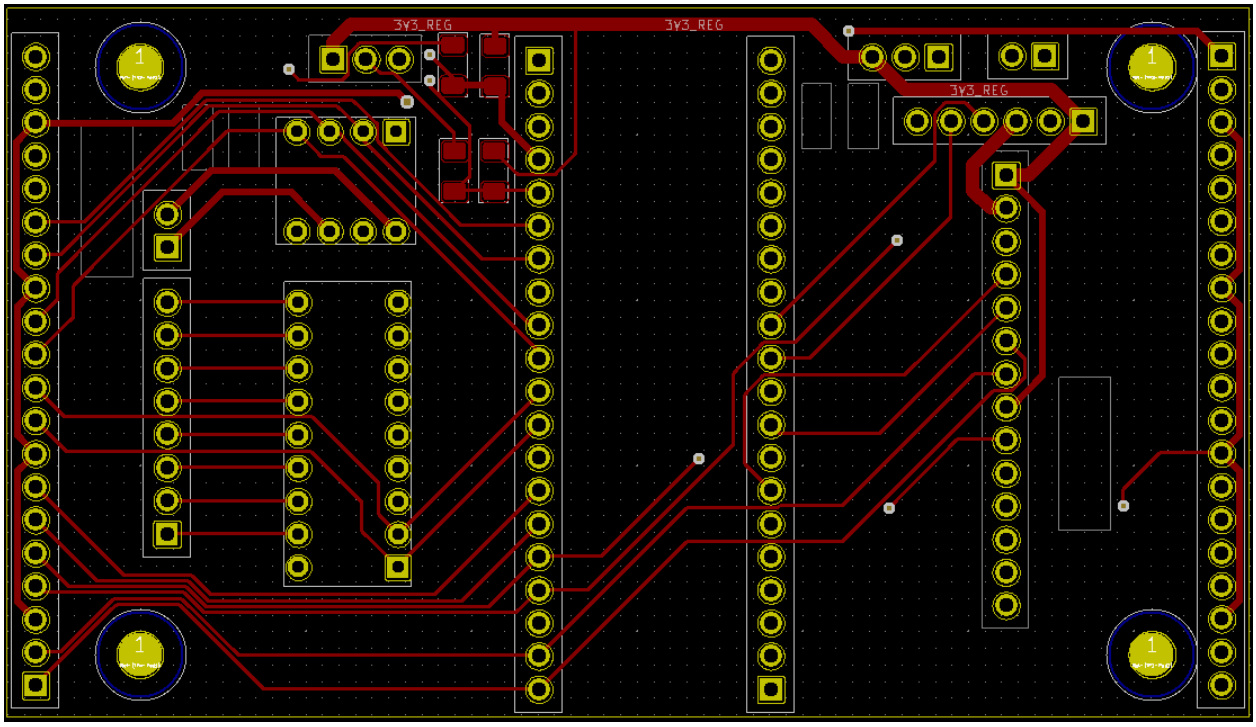


Figure A6: Revision 1.1 PCB Front Copper Layer

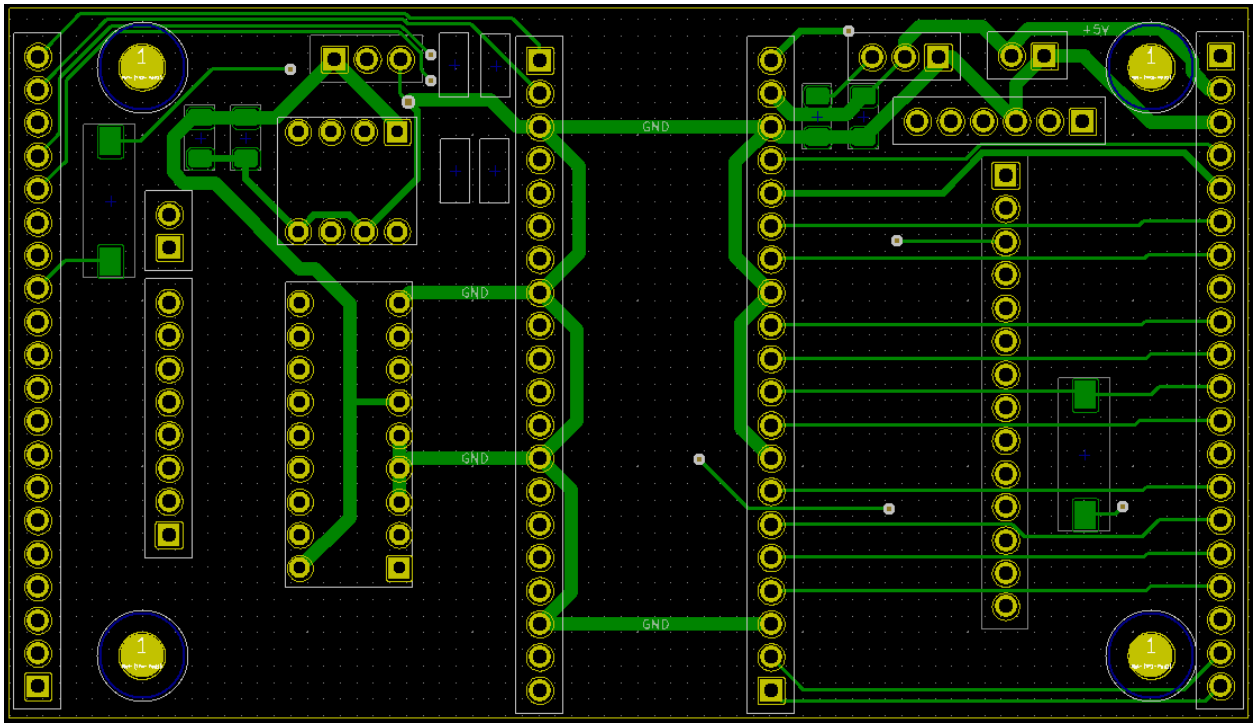


Figure A7: Revision 1.1 PCB Back Copper Layer

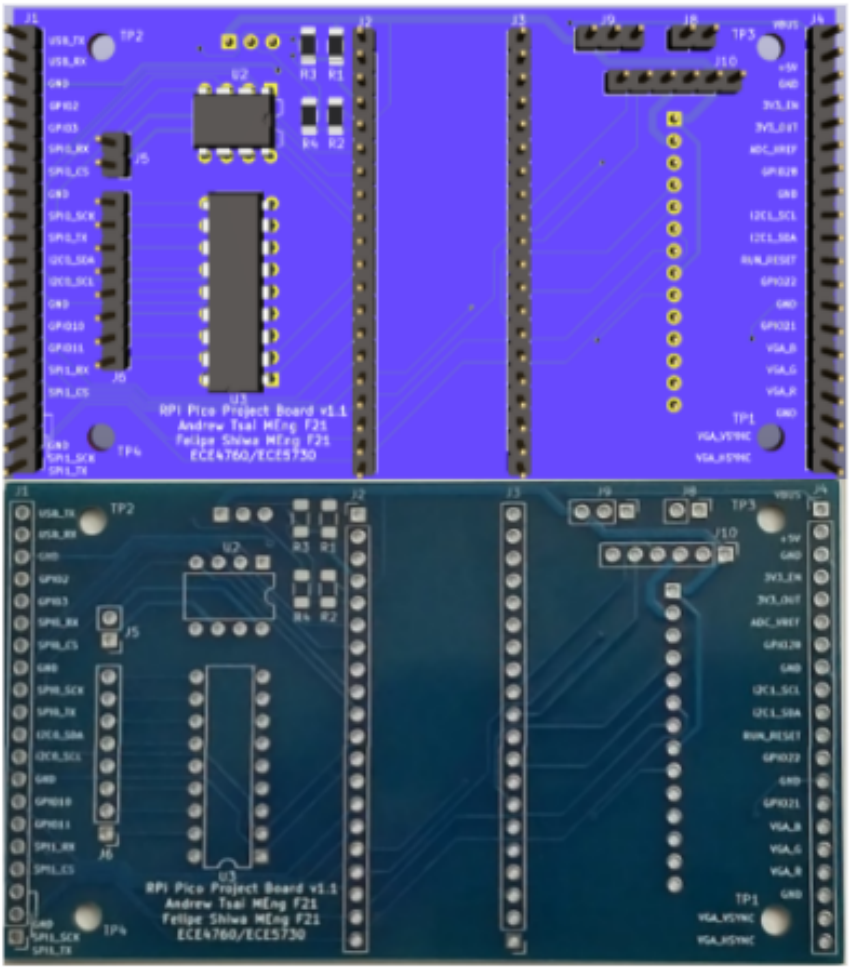


Figure A8: Revision 1.1 KiCAD 3D View vs. Manufactured Board Photo

Software User Manual

A brief overview of the method for installing the C SDK on Windows is given below, with more details contained in [3].

1. Installations required: Arm GCC compiler, CMake, Visual Studio Build Tools, Git
2. After performing the installations, run the following Git commands:

```
C:\Users\pico\Downloads> git clone -b master
https://github.com/raspberrypi/pico-sdk.git
C:\Users\pico\Downloads> cd pico-sdk
C:\Users\pico\Downloads\pico-sdk> git submodule update --init
C:\Users\pico\Downloads\pico-sdk> cd ..
C:\Users\pico\Downloads> git clone -b master
https://github.com/raspberrypi/pico-examples.git
```

3. Set the path to the Pico SDK (this will usually have to be done if building from new project directories that exist outside of where the path variable has been set):

```
C:\Users\pico\Downloads> setx PICO_SDK_PATH "..\..\pico-sdk"
```

4. Then navigate to the directory for the project you want to compile

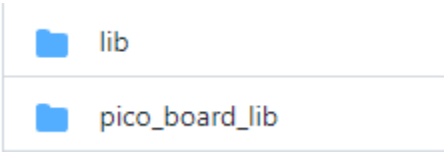
```
C:\Users\pico\Downloads> cd pico-project
C:\Users\pico\Downloads\pico-project> mkdir build
```

```
C:\Users\pico\Downloads\pico-project> cd build
C:\Users\pico\Downloads\pico-project\build> cmake -G "NMake Makefiles" ..
C:\Users\pico\Downloads\pico-project\build> nmake
```

5. After calling the nmake command, a .uf2 file should now exist in the build directory, which can be dragged and dropped to the drive corresponding to the RPi Pico if it was plugged into the computer with the program button held down.

An alternative to this build method is to use Windows Subsystem for Linux (WSL) through VSCode; the steps for doing so are contained in [22].

Our software library is divided into two parts: the display library (lib) and pico_board_lib:



Our demo program (from the DS_demo directory on the project Github repository) shows how these libraries can be included in the CMakeLists.txt file. The following is the associated lines of code:

```
add_subdirectory(./lib/Config_Rewrite)
add_subdirectory(./lib/GUI)
add_subdirectory(./lib/Fonts)
add_subdirectory(./pico_board_lib)

include_directories(./lib/Config_Rewrite)
include_directories(./lib/GUI)
include_directories(./lib/Fonts)
include_directories(./pico_board_lib)

# must match with executable name
target_link_libraries(vga_pio PRIVATE hardware_pio pico_board_lib pico_time
  hardware_dma hardware_i2c hardware_pwm Config_Rewrite GUI Fonts)
```

The `add_subdirectory` and `include_directories` commands are needed to explicitly include the Config_Rewrite, GUI, Fonts and pico_board_lib folders, and the libraries then need to be linked using the `target_link_libraries` command, as shown. The pico_board_lib contains the necessary library consolidations to interface with the on-board peripherals, while the code contained in /lib/ is used to drive the LCD display.

Prototype Board User Manual

Following is a recommended order of steps for populating the Prototype Board:

1. Surface mount components - R1-4, C1-4. Once headers are soldered, these components become difficult to reach.
2. Central header sockets - J2, J3. This will allow you to attach the Pico and test that there are no shorts introduced by the board if it can turn on.
3. Voltage regulator (MCP1702) and power input header- J9, J8. The voltage regulator can be directly soldered onto the board, with GND on pin 1. This will allow you to test that the board powers on from an external 5V power input.
4. IMU header, PDIP sockets for DAC and Port Expander - J10, U2, U3. These peripherals can be tested as soon as they are connected.
5. TFT display header - J7, populate if planning to use the large TFT screen instead of the 1.14 LCD display.
6. Reset button, user button & switch - SW1, SW2, SW3.
7. Outer edge headers - J1, J4, in the desired orientation for convenience.

Bill of Materials

Part/Part Number	Description	Quantity (per board)	Unit Cost	Total Cost
https://www.cytron.io/p-raspberry-pi-pico	Raspberry Pi Pico	1	$4.98	$4.98
EG4582CT-ND (DigiKey)	SWITCH TACTILE SPST-NO 0.05A 12V	2	$0.66	$1.32
450-1634-ND	SWITCH SLIDE SPDT 100MA 12V	1	$1.65	$1.65
36-25505-ND (DigiKey)	HEX STANDOFF M3 NYLON 20MM	4	$0.57	$2.28
36-4688-ND (DigiKey)	HEX NUT 0.217" NYLON M3	4	$0.16	$0.64
A120350-ND (DigiKey)	CONN IC DIP SOCKET 18POS TIN	1	$0.33	$0.33
A120347-ND (DigiKey)	CONN IC DIP SOCKET 8POS TIN	1	$0.20	$0.20
MCP4822-E/P-ND	IC DAC 12BIT V-OUT 8DIP	1	$3.92	$3.92
MCP23008-E/P-ND	IC I/O EXPANDER I2C 8B 18DIP	1	$1.40	$1.40
1568-PRT-16279-ND (DigiKey)	ARDUINO NANO STACKABLE HEADERS	4	$1.50	$6.00
RMCF1206JT330RCT-ND (DigiKey)	RES 330 OHM 5% 1/4W 1206	2	$0.10	$0.20
13-RE1206FRE0710KLC	RES SMD 10K OHM 1%	2	$0.10	$0.20

T-ND (DigiKey)	1/4W 1206			
1292-1603-1-ND (DigiKey)	CAP CER 0.1UF 25V X7R 1206	1	$0.15	$0.15
720-1808-1-ND (DigiKey)	CAP CER 1UF 16V X7R 1206	2	$0.36	$0.72
478-CM316X5R106M10ATCT-ND (DigiKey)	CAP CER 10UF 10V X5R 1206	1	$0.34	$0.34
2553-2011-1X36G00SB-ND (DigiKey)	PIN HEADER, SINGLE ROW, 36 PIN	3+	$0.76	$2.28
https://www.amazon.com/Waveshare-1-14inch-Raspberry-Embedded-Interface/dp/B08XK49TWM	Waveshare 1.14inch LCD Display Module	1	$13.59	$13.59
4554 (Adafruit Product ID)	Adafruit TDK InvenSense ICM-20948 9-DoF IMU	`	$14.95	$14.95

Total price of all components: $55.46
Total price excluding LCD display and IMU: $26.92

References

[1] "Raspberry Pi Pico Python SDK". [Online] Available: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-python-sdk.pdf
[2] "Raspberry Pi Pico C/C++ SDK". [Online] Available: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf
[3] "Getting Started with Raspberry Pi Pico". [Online] Available: https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf
[4] Raspberry Pi, "pico-examples", GitHub repository. Available: https://github.com/raspberrypi/pico-examples\
[5] A. Tsai and F. Shiwa, "Multicore Example", GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/ex_multicore
[6] A. Tsai and F. Shiwa, "Testing Shared Memory", GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/ex_count
[7] A. Tsai and F. Shiwa, "Testing Shared Array", GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/ex_array
[8] A. Tsai and F. Shiwa, "Protothreads Example on Raspberry Pi Pico", GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/ex_protothread
[9] A. Tsai and F. Shiwa, "N-Bodies Simulation Multicore on Raspberry Pi Pico", GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/n_bodies_multi

[10] V.H. Adams, “Dual-core Direct Digital Synthesis (DDS) on RP2040 (Raspberry Pi Pico)”. [Online] Available: https://vha3.github.io/Pico/Multi/MultiCore.html
[11] A. Tsai and F. Shiwa, “N-Queens Multicore on Raspberry Pi Pico”, GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/n_queens_multicore
[12] A. Tsai and F. Shiwa, “N-Bodies Simulation on Raspberry Pi Pico”, GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/n_bodies
[13] A. Tsai and F. Shiwa, “N-Bodies Simulation with VGA Display”, GitHub repository. Available: https://github.com/Luminous22213/RPi-Pico-MEng-Project/tree/main/vga_n_bodies
[14] V. H. Adams, “PIO Assembly VGA Driver for RP2040 (Raspberry Pi Pico)”. [Online] Available: https://vha3.github.io/Pico/VGA/VGA.html
[15] B. Land, “ECE4760 Development Boards”. [Online] Available: https://people.ece.cornell.edu/land/courses/ece4760/PIC32/target_board.html
[16] “MCP4802/4812/4822”. [Online] Available: https://ww1.microchip.com/downloads/en/DeviceDoc/20002249B.pdf
[17] “MCP23008 8-Bit I2C I/O Expander with Serial Interface”. [Online] Available: https://www.microchip.com/en-us/product/MCP23008
[18] “Pico LCD 1.14”. [Online] Available: https://www.waveshare.com/wiki/Pico-LCD-1.14
[19] TDK Corporation, “ICM-20948 World’s Lowest Power 9-Axis MEMS MotionTracking™ Device”. [Online] Available: https://invensense.tdk.com/wp-content/uploads/2016/06/DS-000189-ICM-20948-v1.3.pdf
[20] “Raspberry Pi Pico Datasheet”. [Online] Available: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf
[21] Microchip Technology, “MCP1702 250 mA Low Quiescent Current LDO Regulator”. [Online] Available: https://ww1.microchip.com/downloads/en/DeviceDoc/22008E.pdf
[22] “Easy Raspberry Pi Pico Microcontroller C / C++ Programming on Windows”. [Online] Available: https://paulbupejr.com/raspberry-pi-pico-windows-development/