THREADING AND NETWORKING DEVELOPMENTS FOR THE RASPBERRY PI PICO W

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

Liam Kain, Harris Miller

MEng Field Advisors: Hunter Adams, Bruce Land

Degree Date: May 2024

TABLE OF CONTENTS

ABSTRACT

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Threading and Networking Developments for the Raspberry Pi Pico W

Authors: Liam Kain, Harris Miller

Abstract:

The course ECE 4760 – Digital Systems Design Using Microcontrollers recently switched from the Microchip PIC32 to the Raspberry Pi Pico W, leading to a whole new set of learning opportunities by offering more advanced hardware capabilities. While open source examples exist that use FreeRTOS to implement real-time process scheduling and lwIP for TCP/IP networking capabilities, few resources detail their implementation and even fewer utilize both libraries at once. This report details how the implementation of a WiFi “walkie talkie” on the Raspberry Pi Pico W platform served as a culminating project for the integration of the FreeRTOS with lwIP, the infrastructure we developed for future student work, and the future work to be done to convert this proof-of-concept into an ECE 4760 laboratory exercise.

EXECUTIVE SUMMARY

In our project, we set out to accomplish the task of integrating the real-time operating system kernel FreeRTOS with the embedded TCP/IP stack lwIP on the Raspberry Pi Pico W platform. Usage of both of these libraries is not novel, but the integration on this microcontroller platform is and has major ramifications for the exploration and types of projects that can be accomplished with it. We have developed a library that allows for any person to perform this integration without needing to understand the underlying complexities of the individual packages alone and just focus on the general concepts of threading and networking on embedded systems. The aim for this library is for use by students in ECE 4760 for culminating design projects and potentially any laboratory exercises that become available once it is deemed suitable for the teaching environment.

One project we created using our newly developed software was a new form of “walkie talkie”, or wireless voice communication device, that takes advantage of the existing wireless networking infrastructure in buildings and especially on a university campus. While phone-based voice-over-IP implementations are a proven technology, our accomplishment is the transmission of audio over a network at an extremely cheap cost. Lastly, we detail the steps that should be taken before deploying our walkie talkies to a real, non-isolated network. In particular, we identify parts of the TCP/IP suite that can be used to efficiently route packets between all devices and avoid network congestion.

The associated poster for this report that was presented at the Spring 2024 ECE M.Eng. Poster Session won Best in Category under Communications.

ACKNOWLEDGEMENTS

We would like to thank Hunter Adams and Bruce Land for their guidance and support throughout our project. We must also thank Pickles for her emotional support, for which this project may not be possible.

CONTRIBUTIONS

In the completion of this report, Liam Kain completed the following sections:

- Integration
- Background: Raspberry Pi Pico
- Planning: Issues
- Integration
- Development: Phases, diagrams
- Infrastructure

In the completion of this report, Harris Miller completed the following sections:

- Abstract
- Executive Summary
- Background: Wireless Capabilities, Walkie Talkies
- Planning: System Development, diagrams
- Development: Post-Development Issues
- Future Work
- Conclusion
- References
- Appendix

Both authors worked jointly in the completion of the project and both met for a majority of work sessions. In terms of independent project work, Liam Kain worked on the library portion while Harris Miller worked on the hardware portion.

BACKGROUND

Raspberry Pi Pico

The project at hand sprung from the development of materials for ECE 4760: Digital Systems Design Using Microcontrollers. An important aspect of this class is developing real-time systems, which is currently accomplished through a threading library called Protothreads. This is a very lightweight threading library developed by Adam Dunkels that uses no stack, lacks thread priorities and preemption, and uses very simple structures to control thread execution. With this lightweight structure, Protothreads lack extensive functionality with thread control. Also of note, the Cornell versions of Protothreads were ported to the PIC32 and RP2040 platforms by Bruce Land.

Since the first use of Protothreads in the class, the microcontroller used has also become more powerful. For the Fall 2022 semester, ECE 4760 switched from the PIC32 to the RP2040. The RP2040 is a dual-core ARM Cortex-M0+ microcontroller and is housed on the Raspberry Pi Pico development board, which costs only $4. The original PIC32 is a single MIPS32 M4K core with a custom printed circuit board developed to make using peripherals easier with the course. The PIC32's base clock is 80 MHz but the RP2040's base clock is 133 MHz. Part of the course involves overclocking the core, and the RP2040 can be clocked stably to 250 MHz with a single line of code. With more processing power, the RP2040 can execute more complex tasks than the PIC32, and therefore can benefit from a heavier, but more helpful threading library.

This is where we introduce FreeRTOS, a real-time operating system (RTOS) kernel developed for use on embedded devices and written in C. A real-time operating system names its multiprocessing threads, tasks instead. With these tasks, the RTOS introduces extra complexity

and overhead to the systems, but is also able to add extra functionality. For example, instead of just using semaphores to control tasks, in FreeRTOS, data from one task can be queued up and be accepted by another task. Tasks can also notify each other to start a section of their code. Another improvement is task priority. Unlike Protothreads, tasks can have different priorities and thus get priority over other tasks.

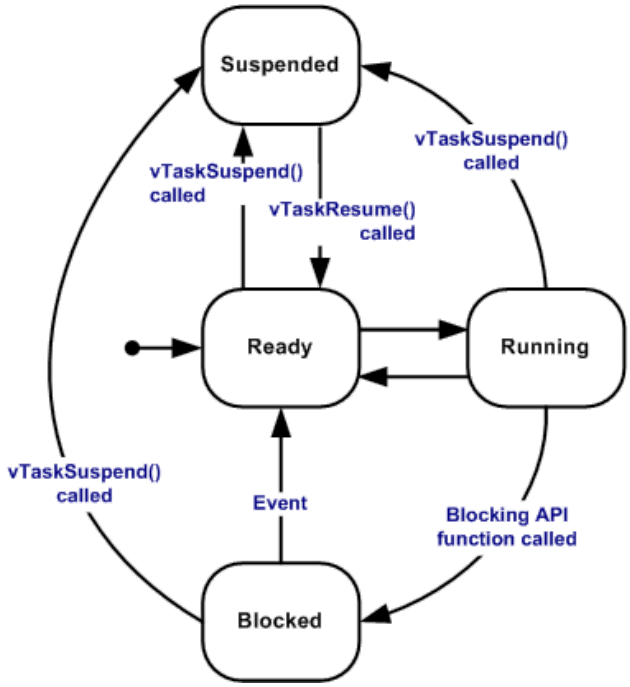

Figure 1: FreeRTOS method description [9]

FreeRTOS is also widely used in industry. From Tesla, to Stanley Black & Decker and Bosch, FreeRTOS is used by many companies and thus prepares any students using it for working in industry. While originally developed by Richard Barry, stewardship of the project was passed on to Amazon Web Services but it still remains open-source and distributed under the MIT license.

Wireless Capabilities

A variant of the Pi Pico also exists that has added network capability, the Pico W. This is enabled by the Infineon CYW43439 (henceforth abbreviated to cyw43), a 2.4 GHz radio IC that supports both IEEE 802.11n and Bluetooth 5.2, and is complemented by an on-PCB antenna. The Pico and Pico W have open-source development of hardware and software, with schematic and layout files available besides the antenna which is licensed. A common library used for this is the “Light Weight IP“ library or *lwIP*. This is a great application to use lwIP with since internet functions require a great amount of processing that is separate from other tasks in the processor. The networking also needs to manage the data it's sending and receiving between processors.

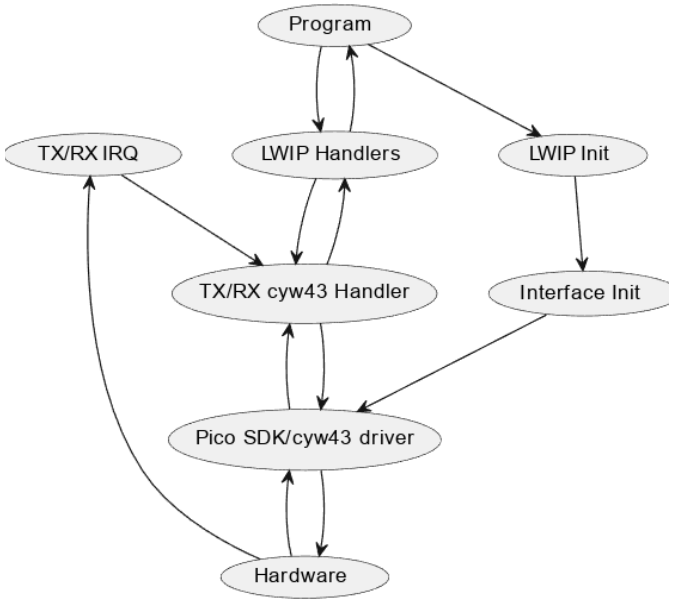

Figure 2: lwIP hierarchy with cyw43 driver

The library works in tandem with the Pico W’s network chip by interfacing with the cyw43’s driver. This figure shows after initializing the library, whenever the cyw43 receives a

network packet, an interrupt will get called thus calling the cyw43 driver's ISR (interrupt service routine) which then calls lwIP's handler which will then actually decipher the network packet.

In summary, lwIP is essentially a translation library from data to network packets rather than a library that actually controls hardware. There are several API's used to use lwIP. There is a sequential API, for blocking, thread-safe actions. There is also the socket API which is more geared towards higher level OS's. Then there is the raw-API. It's a very low-level, thread-unsafe API, but it has very low latency and is non-blocking. This was a good candidate to use for FreeRTOS so that we could achieve the low latency needed for our walkie-talkie as well as have an exercise in enforcing concurrency.

Walkie Talkies

The more-real world rationale for the development of such devices is as follows: typical walkie talkies that operate within the VHF (very high frequency) band of the RF spectrum, typically around 462-467 MHz, with little or no underlying infrastructure. While the upsides of these systems are that the used spectra is typically allotted for these devices only by the FCC (Federal Communications Commission) and the lack of intermediate devices or repeaters allows for cross-brand functionality, they are limited by the media they transmit in. Typical building and construction materials such as steel, glass, concrete, brick, and even insulation dampens the power of the transmitted signals and diminishes the range of communication in offices and campuses.

Almost all modern structures have wireless networking systems. Most of these comprise of the typical wired devices such as routers, switches (L2 or L3), and cabling, with wireless access points added on top. By utilizing this existing infrastructure and typical protocols (such as

from the TCP/IP suite), we can provide voice communications across an extremely long range. As proven by cellular VoIP (voice over IP) technologies, this can be accomplished with low enough latency, bandwidth, and frame loss to provide a good and consistent experience for the user.

If most modern people already have cellphones that can take advantage of both cellular networks and VoIP, then why do we need walkie talkies? Most cellphones that we have in our pockets cost hundreds of dollars and sometimes become prohibitively expensive for organizations to both deploy these devices to users, and could still pose security threats with malware and software exploitation being ever present in this digital age. A “walkie talkie” provides a direct line of communications between personnel, without requiring a phone to ring and be answered, and can be developed openly with the potential to use whatever modern encryption technology the organization wishes to apply.

Such a device also has a personal application to the Maker Labs within Phillips Hall. The Maker Club operates two lab spaces, one in Room 201 by the northern entrance facing Campus road, and another in Room 217, approximately 40 meters down a hallway. When cleaning in the labs or working on projects, we experience plenty of times where we need to look for materials in one lab or check up on the status of a 3D printer in another. Having such a large community means that it is easy to talk to a lot of people, but you do not necessarily know who is in the other spaces which makes it difficult to call them or send a quick message over Slack. This leads us to walking up and down the hallway many times each day, a problem we have experienced throughout our undergraduate and graduate educations and one that has become tiresome but solvable by our device.

PLANNING

System Level

At a high level, it was evident that we wanted to have multiple Pico Ws connected to a single access point rather than a larger segmented network so we could start out with the simplest network for our UDP packets to navigate. Each Pico W would act as both a transmitter and receiver, and as such each would have a microphone for sampling audio and a speaker for transmitting audio back to the user. As an arbitrary number of users could be connected to the network, the users could perform a discovery of all other users' IP addresses, but it is easier to use UDP broadcast such that the packets are directed towards all users and the work is then performed by the network rather than the microcontrollers.

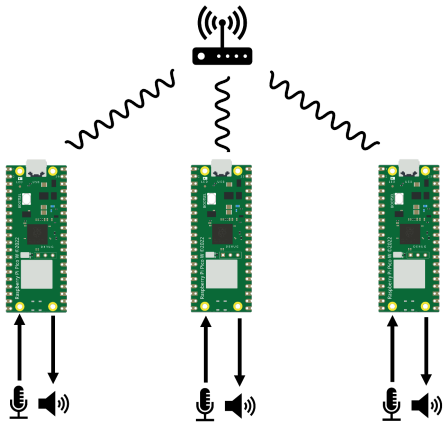

Figure 3: High level connectivity structure

Before starting to breadboard out a prototype of a single walkie talkie, we created a wiring diagram to organize our thoughts and map pin functionality to the hardware we plan to include. Most of this hardware was already in use and vetted through ECE 4760 when the class transitioned from the PIC32 to the Pi Pico. The microphone we used was a breakout board from Adafruit that houses an electret microphone and MAX4466 amplifier, so we did not need

additional hardware besides wiring to connect it to the Pico W. Since the board accepts an input voltage between 2.4 and 5.5V, the 3.3V rail regulated by the Pico W could safely be provided to the power rail. The only consideration was that the analog signal must connect to an ADC-capable pin (analog-to-digital converter). Since the Pico series does not have a DAC (digital-to analog converter) on-board, we must add hardware to support the decoding and transmitting of audio to a speaker.

For this, we used an MCP4822 which similarly accepted a 2.7V to 5.5V range and could safely be used. As a digital device, the DAC has SPI (serial peripheral interface) and acts as a slave device by which the master (Pico) transmits data unidirectionally. The constraint posed is that a limited set of pins are connected to different SPI channels on the Pico but you must select all from the same channel (e.g. must use SCK0, MOSI0, and CS0, and not a mix such as SCK0 and MOSI1, etc.) The last device critical component we added was a button to any GPIO-capable pin (general purpose input-output) that we can press to enable audio sampling and transmit it to the network.

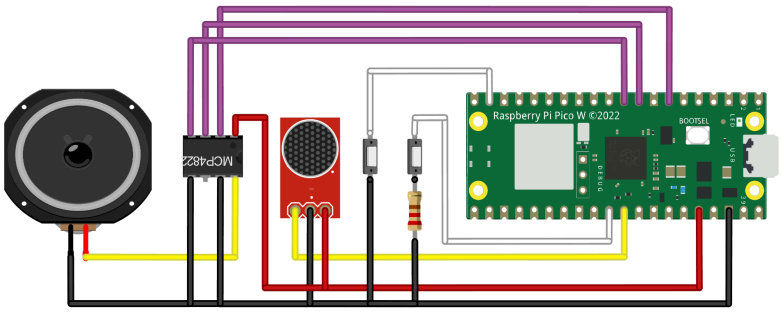

Figure 4: Walkie talkie wiring plan

While we did have some mental plans about the software architecture, they changed drastically across the course of the project as we determined the capabilities and supported features of both lwIP and FreeRTOS. Most of these discoveries were opposite of what we

anticipated and required us to change the pathway of how audio samples were transmitted and received. Here we will present the final high-level pathway even though it was not in this form until further in the development phase of our project:

Audio will be continuously sampled by an ADC channel from the microphone at a rate of 44 kHz. A DMA (direct memory access) channel is then configured to grab samples from the ADC FIFO and form packets by placing the data into its payload. While in the DMA handler, we notify a task from the IRQ context that a new packet is ready for transmission, and then swap buffers such that the next ADC sample will get placed into a different packet's payload. When FreeRTOS is running the transmission task and it receives this notification, it will proceed to call lwIP to transmit a new UDP packet into the network. As lwIP is the highest layer in the network stack that the programmer will interact with, it will manage the network driver to perform the transmission.

On the receiver side, the radio hardware generates an interrupt when a packet is received which then calls the lwIP receiver handler. This handler then grabs the packet payload and sends it to the back of a FreeRTOS queue from this IRQ context. Another timer also running at a rate of 44 kHz checks for queue samples, and when available performs a blocking SPI write. From here, audio is received by the DAC and converted into an audible signal.

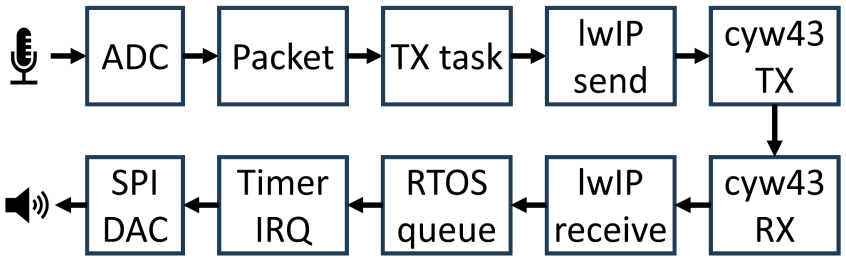

Figure 5: Data transmission and networking block diagram

Issues

The first issue that will be addressed with the project is the lack of documentation of FreeRTOS with lwIP. There are two known example files using these libraries together in the pico-examples repository, but no other known examples. These examples are maintained by just two open-source contributors, so there is not much future development with these examples. With this lack of documentation is also the lack of performance data. Protothreads is a more lightweight library than FreeRTOS, but it's unclear how much of a slowdown the overhead of tasks causes compared to protothreads.

The biggest issue to address is the ease of use of these examples. FreeRTOS and lwIP require a large amount of setup from the engineer to work. The RTOS must be linked into the executable, and various macros need to be defined in order to link in the correct header files. The same things must be done for lwIP. There are also various methods that need to be set up to receive and send data from the Pico as well as to initialize FreeRTOS tasks. ECE 4760 students would spend a significant amount of time setting up these tools, so there needs to be a way to simplify this process and get started with labs more quickly.

INTEGRATION

FreeRTOS

To integrate FreeRTOS, the first step was to work through an example. Thankfully, FreeRTOS has an example in its repository for a lwIP-FreeRTOS demo, a PING example to query another IP address. This gave both the CMake as well as header file setup we needed for getting FreeRTOS working. Essentially, we needed to include a `FreeRTOSConfig.h` file which specified various FreeRTOS parameters and macros. This is standard on all platforms. We also needed a `FreeRTOS_Kernel_import.cmake` file which, like the `pico_sdk_import.cmake` file from ECE 4760, imports the FreeRTOS kernel. And then, for the purposes of most of our examples, a few CMake variables were specified, `WIFI_SSID`, `WIFI_PASSWORD`, `SERVER_IP`, and `NO_SYS=0`. This is because making `NO_SYS=0` tells lwIP which mode FreeRTOS is operating in. A few things that have to be taken into account, firstly, FreeRTOS has multiple modes. There is the standard mode where FreeRTOS runs on an individual core. To do what is required on the pico, FreeRTOS would have to be initialized on each individual core. The other mode is Symmetric Multiprocessing (SMP) mode which will run FreeRTOS on both cores. The user can then select which core each task runs on using `vTaskCoreAffinitySet()`. SMP mode makes using FreeRTOS with the coprocessors much easier, but there's a catch. Any interrupt functions that FreeRTOS provides still only operate on one core just as the rest of the pico sdk functions. Any interrupt functions still need to be run on the second core. Functions like `DISABLE_INTERRUPTS` need to be called on both cores to disable all interrupts.

lwIP

Taking a step back, lwIP had to be implemented separately in order to understand the functionality of the library. To do this, we used Bruce Land's UDP example (reference 1). By stripping down the serial interface and restricting the program to either sending or receiving, a basic UDP test file was created that could either continuously receive messages or if it was set to a sender, send "hello" repeatedly.

The basic structure of lwIP on the Pico is that of course the network driver has to be initialized with `cyw43_arch_init`. Then `cyw43_arch_enable_sta_mode` is called to enable the driver in "Software Station" mode rather than AP (access point) mode which would act more like a router. Next, calling `cyw43_arch_connect` with the wifi credentials will actually connect to the network. After by initializing a network PCB[1] (protocol control block) and packet buffer (`pbuf`), these both can be binded to a UDP port using `udp_bind`. Lastly, by calling `udp_recv` with the PCB and a custom callback function, the callback function will execute every time a UDP packet is received. The callback function takes in a few pieces of info about the received packet, but mainly contains a `pbuf` containing the received data. On the sending side, the step for this is simply to take the `pbuf` that was previously created, fill the "payload" member with the desired data and call `udp_sendto`.

[1] While PCB typically is an acronym for "printed circuit board" in electrical engineering, in low level embedded systems development it can mean "process control block" and in reference to lwIP it means "protocol control block." Confusing, huh!

Combining FreeRTOS and lwIP

Combining the two packages was a very simple affair on the programming side, looking at this figure, lwIP as a translation layer works on top of FreeRTOS. FreeRTOS and lwIP also reserve separate sections of memory, so there shouldn't be any issues with allocating memory using both of the systems.

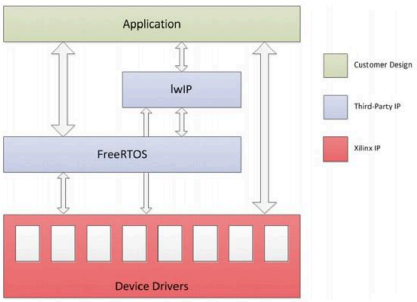

Figure 6: Software layering with lwIP and FreeRTOS, example from an analogous platform [3]

To actually modify these two systems, the CMake files were essentially merged together. There weren't any conflicts between the two except for a major change. When using lwIP for protothreads, its best to use the pico-sdk library `pico_cyw43_arch_lwip_threadsafe_background`, but for use with FreeRTOS, the pico_sdk actually has a FreeRTOS port to work nicely with FreeRTOS. This CMake library is instead `pico_cyw43_arch_lwip_sys_freertos`. With this change, lwIP and FreeRTOS can be used together.

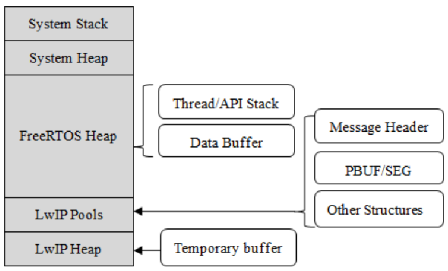

Figure 7: Memory layering with lwIP and FreeRTOS [1]

DEVELOPMENT

Loopback

The first thing to do was to see if audio input and output could work at the same time, avoiding the networking entirely. As seen by Figure 8, the structure was to take ADC measurements from the microphone, use a DMA channel to load the samples from the ADC FIFO into two alternating packets. While the DMA channel is transferring ADC data into the packet, the second packet will send its data to the speaker. Once the DMA transfer is done, the DMA channel has its output switched to packet 2 while packet 1 sends its data to the speaker. This alternates forever.

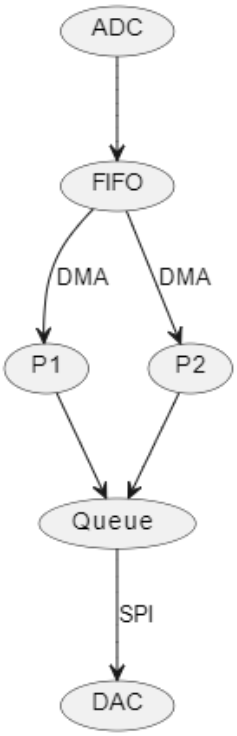

Figure 8: Loopback state machine diagram

The reason for only using two alternating packets was for the assumptions when networking is used. Since the rate of transmission of the packets when networking was assumed

to be faster than the ADC sampling rate (44kHz), then the DMA transfers should take longer than the packet sending, thus ensuring that the DMA transfer won't complete before the second packet finishes sending. For the echo example, the sending to the speaker occurred at about the same sampling rate as the ADC collection, so the assumption was not true, but was ignored for now.

To actually send data to the speaker, the repeating timer from the chirp example was used. Since the DAC that outputs to the speaker had to be the same sampling rate as the ADC for time consistency, the period of the timer was changed to 22 μs (approximately 45 kHz). This sampling rate is slightly too fast for the ADC, but this actually worked in our favor regarding the assumptions we made previously. Due to the faster sampling rate of the SPI send, the DMA transfer didn't stop alternate packets too quickly.

Unidirectional

The two-way communication was the major stepping stone to finishing the walkie-talkies. Since we were using FreeRTOS, then if implemented both the receiving and sending tasks in individual RTOS tasks, then running both tasks at once should be simple. As seen from Figure 9, instead of queuing the packets directly to the SPI, the packets are sent over the network using `udp_sendto`. Then, on the receiving side, the packets are received and then queued just as in the loopback system.

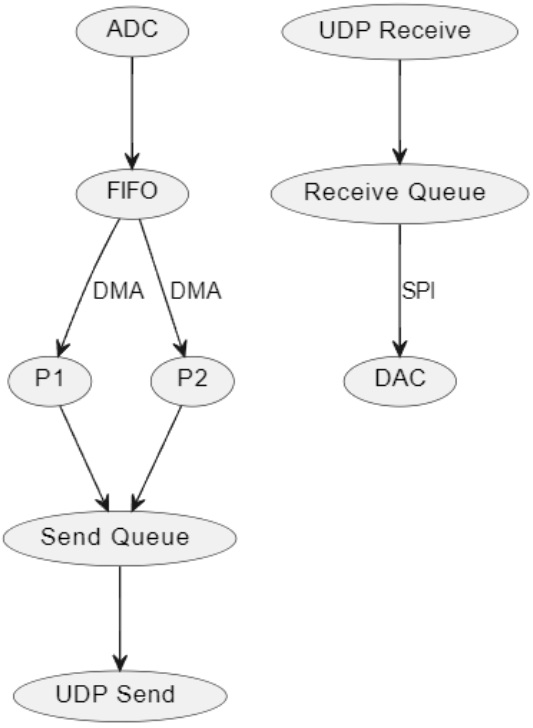

Figure 9: Two-way communication state diagram.

Broadcast

For the final walkie talkie, we went for a broadcast approach. Each Pico would be running the two RTOS tasks in parallel. The power of FreeRTOS showed in this step as very little changes had to be made to the code. The tasks simply both had to be started instead of one task per pico. We then connected a button to one of the GPIO pins and only executed the transmit task when the GPIO was high, or else we executed the receive task code.

A major change we had to make was to prevent anarchy in the walkie-talkie system. We had to prevent multiple devices from trying to transmit audio at the same time. To fix this, we implemented a channel acquisition system similar to the programming construct of locks. If the channel is clear, which is defined as receiving no packets within a 100 milliseconds span, and the user wishes to broadcast, the program will allow the walkie talkie to transmit packets into the network. Now that all other receivers are getting these packets, they see the channel has been acquired and are blocked from transmitting back to the network. If any user attempts to press the button, their audio will not be transmitted.

While this system does suffice, it does not ensure fairness like modern walkie talkie systems. If you have a real radio and someone is transmitting, any other user can begin transmitting and take control of the channel, stopping the original transmission. This allows for higher priority messages, such as those from emergency services, to get their messages out to all receivers.

Post-Development Issues

The first issue we had to deal with during development was with `pbuf`. The packet buffers are designed by lwIP as nodes in a linked list of packets. The buffers also have various options for how the memory should be allocated for the buffer. It can be allocated in RAM, on ROM, or in a "pool" of memory that future packets can dynamically grab from. This causes a number of issues when we only want to send one packet at one time.

One of which is that only having a single packet storage element would bottleneck the DMA process, as we cannot save the ADC data into a packet while that packet is being read by TCP/IP layer for transmission. To solve this, we have two statically allocated packet buffers, one of which at a time is the destination address for the DMA channel. When the DMA has successfully finished a single transaction, it flags the filled packet as ready for transmission and then changes its destination address to a second packet buffer. On the next successful transaction, the same thing happens and the buffers are reversed. This process relies on the transmission of a packet taking much shorter than the period of a DMA transaction.

Another `pbuf` related issue was due to the "reference" count of a packet buffer. It acts like a shared pointer such that the responsibility of "freeing" is not solely the responsibility of the creator. Since each call to `udp_sendto` decrements the reference count, calling `pbuf_free` to clear it out would result in the entire `pbuf` being free'd even though we still need to keep it in scope. To bring back up the reference count, we add in an additional call to `pbuf_ref` after the UDP transmission but before the free call.

The second issue we dealt with was with the placement of our RTOS receiving queue. Initially, to use FreeRTOS to its fullest, the receiving task would take queued packets, and then insert the queued packet in a buffer that could be sent over SPI. The problem with this was the

context switch from any other task to the receiving task was too long to grab every packet in time. FreeRTOS sets the context switch timer to about 100 ticks. This context switch time is too high for the rate at which the audio packets need to be received. The audio packets are around 1000 bytes, so there are about 44 packets per second, therefore, if there were any delay from the context switch, there would be a 44 Hz harmonic present in the output signal. This is exactly what we saw in our oscilloscope output. Moving popping packets from the queue from the receiving task to the actual 44kHz IRQ greatly reduced this delay eliminating the 44 Hz harmonic.

INFRASTRUCTURE

A big part of the project was in using the knowledge we gained for ECE 4760 students. We did this by creating an easier way for students to build up projects using lwIP and FreeRTOS. To do this, we created a CMake library that students can import into their own pico project and have wifi and FreeRTOS available without the CMake hassle. A difficult part of making this library was to make sure not to hold the hand of the student, but make sure the students understands the library they are using while also not having to build their program from the ground up and do the rediscovering work that we've done.

The library is separated into 4 sections: `udp_common`, `udp_recv`, `upd_send`, and `udp_status`. Udp_common of course contains common variables and functions used in the other sections, most importantly, `rtos_udp_common_init` which will initialize lwIP in FreeRTOS. `udp_status` contains an initializable task that will blink the Pico's LED while it's trying to connect to Wi-Fi. `udp_recv` contains a method to initialize a receiver callback when UDP messages are received. It also contains an example callback function as well as an example receive task that the user can add to the FreeRTOS task list. This example callback and task work together as the callback will notify the receive task when data is received. The task will then print the received data out. This is a good example to use when ensuring data is being received by the Pico. `udp_send` simply contains an example task to send data to the address `SERVER_IP`. Students can build off of this task to send whichever data they wish to send.

This structure of the library allows the student to be able to set up UDP, send, and receive packets simply by calling a few functions and setting up a few tasks. They can easily plug in

their own tasks and interrupts based off of the examples fairly easily. This will make it easy for students to start their own UDP projects.

Now the hardest part of using this library was in how students should include it. In ECE 4760, files for a given program are simply kept in one directory. We wanted the same behavior, but wanted to keep the library as a separate directory. We therefore wanted to make the library an actual CMake library rather than just a folder of source code files. By doing this, the project can link the library and be able to use the library headers. They don't have to physically add the library source files to the executable or include the files using a relative path. There was another issue though and we needed to use the `INTERFACE` option when creating a CMake library. When making a CMake library in any other way, the library is generally compiled into a separate library file. This file is then compiled with the source code. The upside to this is the file is the only thing necessary to compile the library into the project without the library's source code. The downside to this though is that if a student wants to modify any part of the library, they have to recompile the library separately. The examples we used would set the WiFi credentials through CMake variables, so these variables needed to be propagated to the library source code. By making the library an interface, the library source files are simply treated as another directory in the project. Therefore, recompiling the project will recompile the library thus propagating the CMake variables into the library files.

With this CMake work done, a student simply has to copy the library directory into their project, call `add_subdirectory()` on the library directory, and then link the library into their project.

FUTURE WORK

Now that we have developed a working proof-of-concept and provided libraries that perform the dirty work, the only thing remaining for students to take this on as an exercise in a laboratory environment is a proper document or website, such as those from Hunter Adams on https://ece4760.github.io/. Having additional students or instructors replicate our project would help to identify what steps and stages make for good checkpoints in a multi-week lab, any details must be additionally provided for those without a background in computer networking as ECE 4450 is not a requirement before this course, and any information that should be purposefully left out and discovered by the student that would make for a good learning experience.

While we have developed our walkie talkie to a mature state that functions on a standalone network, there is still additional functionality and modifications that should be completed before deployment on a large network such as one at Cornell. Right now, the UDP packets are being transmitted to all hosts that match `IP_ADDR_ANY`, so the moment the packet hits a network switch it will be duplicated and sent towards all available hosts on the subnet. This could cause unnecessary congestion on the local network.

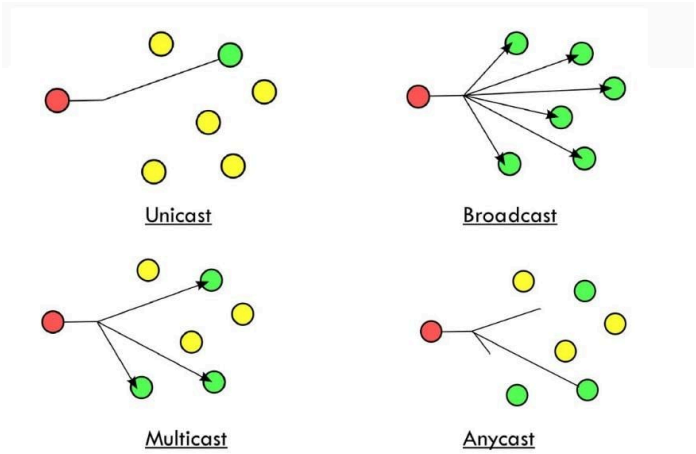

Figure 10: Forms of single to multi-user network transmissions [2]

This can be solved by taking advantage of the Internet Group Management Protocol (IGMP). This is the more appropriate method to multiple devices in a one sender, many receivers scenario. When a device powers on and connects to a network, it will send a message to join an IGMP group. Each group is tied to a multicast address, and when a sender sets the destination address to that of the group, the router will duplicate and forward the message to all users that have joined the group.

Additional work that we were unable to explore in the constraints of our project was towards the Bluetooth capabilities of the cyw43 module. The hardware supports both IEEE 802.11 b/g/n and Bluetooth 5.2, but Bluetooth support within the pico-sdk v1.5.0 is still in beta. The Bluetooth stack has a much higher level of complexity and is deserving of its own independent studies on the matter.

CONCLUSION

In conclusion, we developed a library that successfully integrated lwIP and FreeRTOS features for the Raspberry Pi Pico W, allowing for the future expansion of laboratory exercises for ECE 4760 and additional explorations by students and instructors alike. While most testing is needed to determine the maturation of our software packages, our preliminary results are promising and serve as the basis for future course development in the School of Electrical and Computer Engineering. The development of a “walkie talkie” acts as a proof of concept for the deployment of the library in a project that utilizes more advanced properties of a real-time operating system in conjunction with live networking. While other projects that were part of master’s and independent design projects involved networking, building on top of FreeRTOS allows for true thread safety, thread priorities, and better static and dynamic memory management for independent processes.

REFERENCES

[1] L. Liu, N. Li, and L. Feng, “Improvement and optimization of LWIP,” 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Oct. 2016. doi:10.1109/imcec.2016.7867431

[2] O. Ergun, Unicast Multicast Broadcast Anycast and Incast Traffic Types, https://orhanergun.net/unicast-multicast-broadcast-anycast-and-incast-traffic-types.

[3] “Runtime Software,” Zynq, https://www.amd.com/en/training/customer/adaptive-computing/zynq/runtime-software.html .

[4] hwm44/pico-meng, https://github.coecis.cornell.edu/hwm44/pico-meng.

[5] raspberrypi/pico-examples, https://github.com/raspberrypi/pico-examples.

[6] raspberrypi/pico-sdk, https://github.com/raspberrypi/pico-sdk. hash:`c9cce7a`

[7] FreeRTOS/FreeRTOS, https://github.com/FreeRTOS/FreeRTOS. hash:`7372519cb`

[8] B. Land, “UDP from Pico W to Laptop,” ECE4760, https://people.ece.cornell.edu/land/courses/ece4760/RP2040/C_SDK_LWIP/UDP_send_recv/index_udp.html.

[9] “FreeRTOS API Reference,” FreeRTOS, https://www.freertos.org/a00106.html.

[10] LwIP: Overview, https://www.nongnu.org/lwip/2_1_x/index.html.

APPENDIX

Our shared repository is accessible at the following link, hosted on the Cornell University COECIS GitHub Enterprise Server: https://github.coecis.cornell.edu/hwm44/pico-meng. The most recent commit hash at the time of writing is `3d297b8`. After cloning the repository, the submodules should be initialized and updated.

This includes the `FreeRTOS` repository (which contains the FreeRTOS kernel) and a copy of the `pico-examples` repository. The Raspberry Pi Pico SDK repository `pico-sdk` is NOT included and must be installed separately, and checked out to commit hash `c9cce7a` on the `develop` branch. At the time of writing, SDK release v1.5.1 is incompatible with our work and the pull requests that make the necessary changes are targeted at v2.0.0, hence the use of a commit on a development branch.

For users that followed a guide such at this one provided by Hunter Adams at https://vanhunteradams.com/Pico/Setup/PicoSetup.html or this automated installer hosted by the Raspberry Pi Foundation and developed by Nikhil Dabas at https://github.com/raspberrypi/pico-setup-windows may need to determine the correct installation directories and checkout the proper commit from there.

It may be desirable, but at the user's own discretion, to determine whether to modify the state of the existing copy of the `pico-sdk` or clone a second copy just for usage of our library and programs developed using the library. Be warned of `path` issues regardless of the path you take.