Introduction - The Problems

I have a neurological disease which has required a tracheostomy to which I attach a portable ventilator and carry on my power wheelchair. The trach makes it painful and difficult to turn my head to look behind me. Therefore, I have problems backing up since I cannot see behind me.

I did purchase a backup camera which is a huge help. Howsever, it is a wide angle camera so I cannot determine distance. Even worse, I was constantly backing into things and people.

To make matters worse, I live in a high rise retirement center and need to take the elevator. This is a challenge since some of the elevators have a small distance front-to-back from the door to the back wall. The ventilator tray on the back of the chair makes the problem worse since it protrudes back several inches. On top of that, as I'm backing out of the elevator in my chair, some of my senior toddler neighbors are in a rush to get on as I'm getting off and so they walk in the path of the chair as I try to exit - since I cannot see behind me. At over 700 pounds, the power chair is a danger to these seniors.

The last automobile I had before I could no longer drive had these super helpful proximity sensors that would sound when the distance between the car and outside objects crossed a given threshold. When I received the chair, I thought: "surely there will be proximity sensors for power wheelchairs". I asked the chair supplier and was told that there were two types: one was $3000 and the other $5000. As a career geek, I set out to find a better way.

Introduction - The Solution

I've had forty years of programming experience but zero experience with microcontrollers, breadboards, resistors, etc. But with $5000 in incentive, I started looking for alternatives and landed on the Raspberry Pi Pico W with bluetooth.

As I started to research a solution, it was clear that debugging the Pico would be the most difficult challenge. It was. Given that the sensors would be on the vent tray behind me making the proximity tones difficult to hear, I decided to have the distance readings transmitted over Bluetooth to my cell phone and from there to my bluetooth hearing aids. This made the application development and debugging much easier since I could use Android Studio with a robust IDE and debugger.

It turned out that writing code in C for the Pico was pretty simple. To debug, I could use nested *printf* statements to "step through" the code - and it worked extremely well. All in all, it was straightforward to develop the Pico code and I found the SDK to be accurate, and well written.

Requirements

Pico

As stated earlier, due to debugging challenges on the Pico, I decided to limit how much functionality the Pico would handle. Here is a list of rough requirements:

- Use Bluetooth LE ("Low Energy") GATT and do my best to limit the power requirements since my power supply on the power chair is only 1 amp sourced by a single USB phone-charger-adapter provided by the chair manufacturer.
- Use C to write the code since it's not slowed by the interpreter; and because I have avoided learning Python and have non-trivial C code experience.
- During most of my commercial development experience, I would dig deep into the underlying APIs to see how things worked "under the hood". I really don't care now so for this project - just - "get 'er done" using the well documented high level APIs. If it works - don't play with it.
- Ultimately, the app would read the distance every second or less (currently at 750 ms) and return the three directional reading over bluetooth to my Android phone as a single 3 byte hexadecimal.

Android

I've written other Android Bluetooth programs and so this was an easy choice; ie, to put as much functionality on the shoulders of Android where I have a solid IDE and robust debugger. So here's the flow of the program:

- Find the device that advertises the GATT service and characteristic(s). See the discussion below on why I made the choices I did; but suffice here to say: I implemented a rather obscure Service, the Environmental Service with a single characteristic: Location_and_Speed. NOTE: since this is a super simple application, I really only used the service name and characteristic and did not implement the other features of this protocol. In fact, this was rather arbitrary and convenient since the PICO SDK supported it. In an ideal world, I would have just created a bogus service and characteristics and generated an appropriate and unique GUID for each.
- The Android phone will "Subscribe" to the notifications from the PICO, which is currently configured to send updates every 750ms (three-quarter second).
- Once the app finds the device, it connects and immediately subscribes to updates. Since we only have one service and one characteristic, this does not need to be driven by the UI - just make it go.

- The PICO will send the three distance readings, in a three byte hex to the Android phone. Android will then separate LEFT, RIGHT, and CENTER readings. This is where there are a bunch of choices as to how to handle the data readings. For now, this is where I landed:
    - Set a Min and Max threshold to alert. If the distance across all sensors is over the Max, just ignore. If the distance in any of the sensors is less than the MIN, play a solid tone for that sensor. If more than one sensor is below the MIN, play the solid tone for the closest one.

      NOTE: I found that it takes a few seconds for Android to establish the remote connection to the Bluetooth speaker (hearing aids). Therefore, the first tone would take too long to play. The solution was to increase the max distance to alert.
    - Each sensor has a unique tone. This isn't absolutely required since I assume I'll know which axis is getting closer; but, it was easy to implement and is kinda nice. After using it for a few months, it's subconscious to hear the tone and know which sensor is sending.
    - The closest distance of any of the sensors between the Min and Max will play a tone ranging from a solid tone (less than or equal to Min); and a slow tone as the distance gets closer to Max. So, as I start hearing tones, I know I'm getting close. As they get more rapid in frequency, I know I'm getting closer. Once the tone goes solid, I know I'm too close and so STOP!!!
- The tones will play as any other audio media on the appropriate audio speaker connected to my Android phone: the internal speaker; or, an attached bluetooth speaker; or, my bluetooth hearing aids. I use the bluetooth hearing aids almost exclusively since the tones will not bother others since I'm the only one who hears them

The Code

Pico Code Setup

The setup for the PICO was really rather simple since we did not try to debug the code other than through the printf statements. However, there were some things I found that made it easier. The following will be as much for me, in case I want to jump back down the road to remind me of what I did and why.

Start with One of the SDK Samples

1. Tom's Hardware and DigiKey have some great startup information and samples. To make things super simple, run the setup script (assumes running on Linux):

   wget https://raw.githubusercontent.com/raspberrypi/pico-setup/master/pico_setup.sh

2. TIP: not all the samples, including the blinking LED, work on the W. Use samples from the pico_w subfolder in the pico-examples tree. Pay particular attention to the CMakeList.txt in the samples and understand each area.

3. The SDK has a script that will take the xxx.gatt file and generate the required header file. From the CMakeList: *pico_btstack_make_gatt_header(simpleProx PRIVATE "${CMAKE_CURRENT_LIST_DIR}/simpleProx.gatt")*

4. Since the [BlueKitche/BtStack]() folks wrote the Pico W bluetooth SDK code, it's a great idea to put their sdk in your dev folder as well. I found looking at their samples and API were invaluable for doing the development here.

5. By far, the easiest and best sample for Bluetooth in the W, as of this writing, is the StandAlone sample. It's complete and compiles out of the Pico SDK. From your SDK, you would find it like this: *../pi/pico/pico-examples/pico-w/bt/standalone*

6. The simplest app to learn for the Pico W is the blink example. As noted before, the regular PICO's Blink example does not work on the W since the W's LED is on the wireless chip and not one of the GPIOs. So let's look at a simple project to blink the W's LED to discuss the parts.

    a. Locate the project here: *../pi/pico/pico-examples/pico-w/wifi/blink*

    b. As noted in the CmakeList.txt file you need:
    pico_cyw43_arch_none   # we need Wifi to access the GPIO

    c. The *picow_blink.c* file has two includes that you must have for the W to work with bluetooth, wifi, and or the wifi LED:
    #include "pico/stdlib.h"
    #include "pico/cyw43_arch.h"

    d. Once you have your source files created, you will have a sub-folder called "build"

    e. In that folder from the command line, you would execute the make process as follows: *cmake -DPICO_BOARD=pico_w ../* NOTE: you must include the board type for the maker to pull in the necessary build files; otherwise, the code for the W likely won't work!

    f. After cmake, you will make such as with an 8 core: *make -j8*

    g. After the make, you will find the xxx.uf2 file in the build folder. Copy the file to the pico: hold the bootsel button on the pico down while you connect the micro-usb to the box. Drag and drop the .uf2 file to the pico folder that pops up. Note, this will disappear and the pico will start running the code immediately on the automatic reboot.

h. As noted in “c” above, you only need the two header files to have the program work. However, to make life simpler and not require the debug probe and to simply use the printf, we want to output our debug statements over the USB port so our console can read the printf statements. Do do that, we need to include the standard IO header: *#include <stdio.h> // for usb out* And, our Init lines look like this:

   stdio_init_all();
   cyw43_arch_init();

i. Our entire blink,c file now has this:

```
#include <stdio.h> // for usb out
#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"  // for wifi and led

int main(){
    //Initialise I/O
    stdio_init_all();
    cyw43_arch_init();

    //Main Loop
    while(1){
        cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, 1);
        printf("LED ON!\n");
        sleep_ms(1000); // 0.5s delay

        cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, 0);
        printf("LED OFF!\n");
        sleep_ms(1000); // 0.5s delay
    }
}
```

j. So we now need to get our output through the command line and for that we’ll use Mincom. With the app running on the pico, and the pico connected to the box with minicom, run the following from the command: *sudo minicom -b 115200 -o -D /dev/ttyACM0*

Pico Code

As described in the Android code, this is a “one-off” program and so there are very few optimizations. If it works, use it. For example, if this were production, I would definitely spend more time making sure the data types are as small and efficient as possible; particularly since this is a microcontroller with limited resources.

Source Files - server.h

This header file has the function prototypes required by C; but the static GPIO pins defined here should be described.

In the Hardware section below, the layout of the GPIO and sensors I used is described. Suffice it to say here that this file will tell the functional code where to send the transmit tones, where to receive the return tones, and which sensor is operational.

For example, the location of the center trigger and echo pins for this app are set here to: center trigger pin is GPIO pin 13 and the center echo pin is GPIO pin 11. There is more detail in the Hardware section below, but suffice it to say here that the Proximity Sensors have four wire connections: power, ground, trigger, and echo. The Trigger is the pin that sends the outbound sound burst (10 milliseconds) and the Echo pin is the pin that listens for the reflected sound. Distance is measured by determining how long it takes for the sound to reach and return from the target; using the speed of sound

The important point is that if the number of sensors changes, or the location of where the Sensor Pins are attached to the microcontroller; that is, which GPIO location, then the code would need to be modified and recompiled.

Source Files - server.c

This module has pretty simple functionality.

To start with, the code will loop through the three sensors and using the Echo and Trigger Arrays of the GPIO locations, will send a short, 10 millisecond, Trigger tone to the outbound Trigger port on the sensor. It next records the current time that is effectively the send time.

Next, it enters a “hard loop” that listens for the return or Echo tone. From this, we know how far the object is to the sensor. Specifically, we use the speed of sound to determine the distance, ignoring things such as humidity, etc which are relatively minor since we are only looking for relative and approximate distances. NOTE: this app is really focused on relative distances; that is, “am I getting closer to an object”. If exact distances are required, one would want to attach a humidity sensor; or better yet, buy a sensor with an onboard humidity sensor that allows for distance adjustments using the relative humidity.

We use the formula: *float distance_cm = (echo_duration / 2.0) * 0.0343;* This converts the time the sound takes to travel to and from the object, divides by 2 since it was a round-trip and we only care about one leg which represents the distance TO the object; and multiples by the conversion factor of .0343

The remainder of the code handles the outbound Bluetooth messages. See the SDK if you don’t understand it.

Source Files - startup.c

The variable HEARTBEAT_PERIOD_MS is used to determine how often the microcontroller updates the distances for the three sensors. This is currently set for three-quarter of a second or 750 milliseconds.

The function heartbeat_handler is not necessarily required. However, the blinking onboard LED on the microcontroller is useful to see if the code is running or has “hung”. Once the code was tested, it did not have the hang problem; but, the function is still there just in case.

Source Files - server.c

Android

The code is well documented but pay attention to the following tips:

1. The AndroidManifest.xml file has key sections that are needed for the broadcast receiver to work properly. Don’t make changes to this unless you research and know what they are doing!
2. The Manifest file has a number of required permissions. The app defaults to simply checking to see if the permissions have been granted. If not, warn and close. Thus, the responsibility is on the user to set these manually. If this were a production app, we could do that programmatically.

Source Files - ProximityDataClass.java

The following two lines will set the Min & Max distances for notifications. As described earlier, the tones will not transmit if the distance is above the Max distance.

```
private static final int MAX_DISTANCE_CM = 60;
private static final int MIN_DISTANCE_CM = 10;
```

If the distance is between the Max and Min distances, repeated tones will sound in increasing frequency as the distance decreases.

When the distance is MIN or less, a solid tone will sound.

The remainder of the code in this module is self describing. But in a nutshell:

1. Find the closest distance transmitted in the current 3-sensor readings and from which sensor it is sourced from.
2. Assign the appropriate tone (a different tone for each sensor) to that reading.
3. Get the distance and if greater than or equal to the Max, then bail.
4. Calculate how long the tone plays based on how close the distance is between the Min and Max. If close to the Max, only play for a very brief period. If closer to the Min, play for a longer period.

The tones will play from the Android phone to whatever output device it normally plays from. In my case, that's almost always my bluetooth hearing aids.

Source Files - ProximityDataClass.java

This module implements the Android service stack. There were a number of design choices and the code module describes most of them along with the reasons the choices were made. But in essence, this was designed as a "one-off" application and not a production app. If it were a production app, this module would need quite a bit more work to make it ship-ready.

NOTE: this app should run a service since it will stay up and continue to broadcast even if the phone is dark and in standby.

Source Files - MainActivity.java

As mentioned previously, this app is written as a "one-off" to solve a specific need and IS NOT production ready. Therefore, a number of the design choices would need to be revisited for a production app. Here are some thoughts on the code in this module.

Remember that the Bluetooth Intent Filter must also be declared in the AppManifest.xml file or the Broadcast Receiver callbacks will not work.

The bluetooth permissions required by Android are NOT set programmatically. Since these are only set once when the app is installed, we opted to keep the code tight and to notify if not set, and to require the user to set them manually. Basically, just two permissions: *Location* (fine) and *Nearby Devices*. If this were a commercial app, we'd do this programmatically and there is ample sample in the Android SDK.

This module also has the user interface elements. These are super simple and inelegant. There is little interaction with the UI since we just want to connect to the microcontroller and display the distances. In use, the phone is on standby in my pocket and I focus on driving the chair so the UI is not used except at startup. However, the distances are handy for troubleshooting. For example, I also have a backup camera and one of the techs moved the camera sensor in front of the prox sensor and I could see the distance was around 2cm. So, I knew which sensor was the problem and was able to reposition the camera sensor so the proximity sensor

Hardware

I opted to use the Raspberry Pi Pico W because

- it's small: about 2 inches long by 1 inch wide
- It has Bluetooth LE built in
- Has plenty of documentation and sample code to get started

I have an infrared camera on my wheelchair and so I could not use the various infrared proximity sensors; so, I went with the ultrasonic sensors; specifically, the HC-SR04 which claims accuracy of .3cm (about a tenth of an inch) and will measure distances up to 14 feet. Plus, it has a small power consumption of less than 2ma.

The only problem with the sensors is that they are 5 volt and the Pico W is 3.3. I could use the power input pin on the PICO W which is 5 volts as the power input to the sensor; but, the return ECHO pin sends the response as 5 volts and there's a risk of damaging the board which runs at 3.3v. Therefore, I had to step down the voltage using resistors so that the voltage was reduced from 5v to ~3.3v.

Given the loss of feeling in my finger tips, this was an unnecessary step since I could have spent a few more dollars and bought a similar device that supports both 5v and 3.3v such as the HC-SR04+

The power chair has a 5v/1amp usb charger port that is more than adequate to power the Pico. It is surprisingly snappy and accurate.

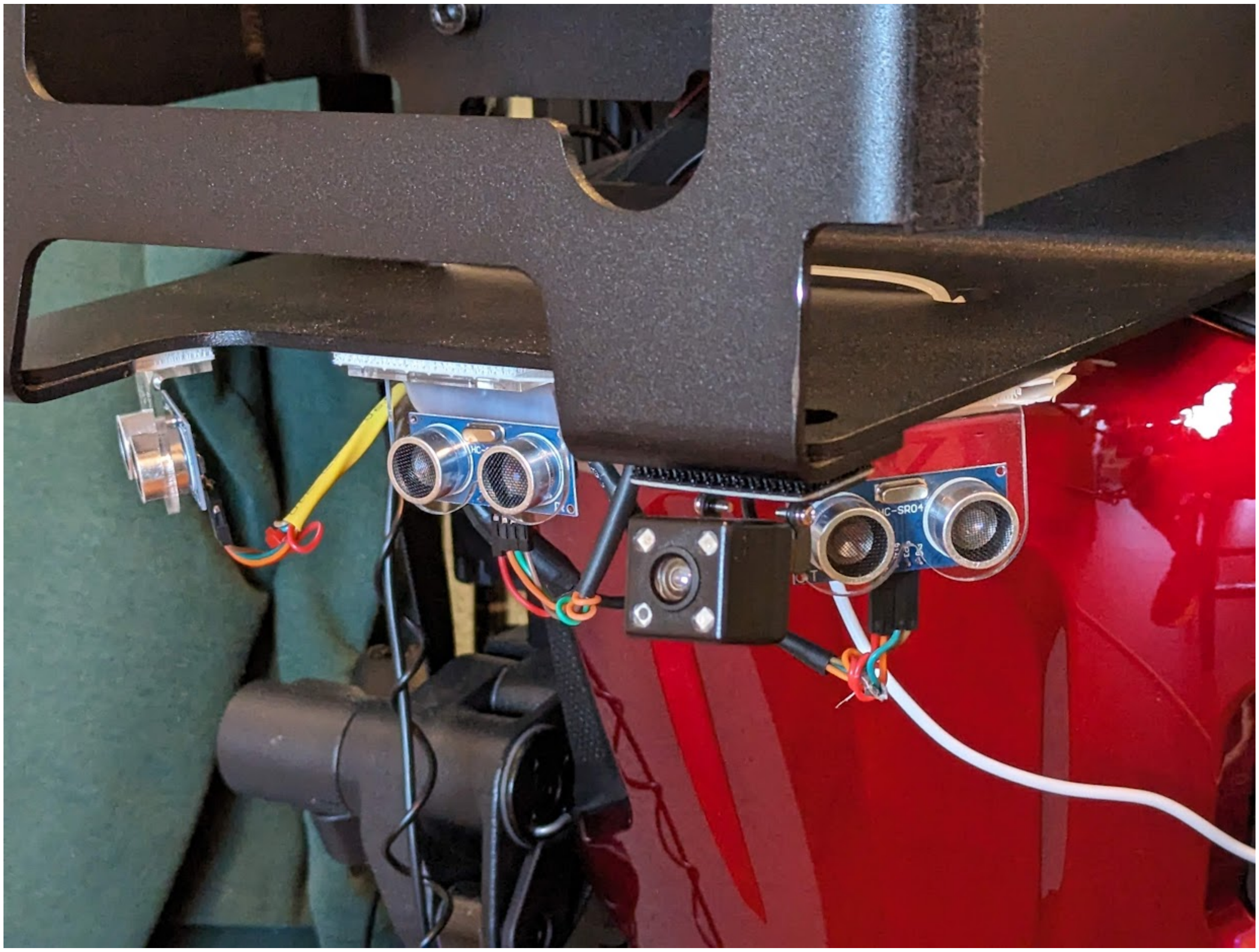
Every three-quarter seconds (750 milliseconds), the Pico will broadcast the three proximity readings over Bluetooth LE (Low Energy).

The Android app running on my phone will connect to the PICO and SUBSCRIBE to notifications from the PICO. The notifications are the current readings from the three proximity sensors. When those three readings are received, the Android app will break them apart and display them on the app; and transmit the appropriate tones as described above.

The only minor issue is in the way that the Android phone transmits the sounds. To save energy, the phone will disconnect from the remote bluetooth speaker - my bluetooth hearing aids in my implementation. Therefore, the first tone will take a couple of seconds to start to play - which I found to be problematic. The workaround I used was to set the max distance greater so that the tones will start to play sooner. Another solution would be to have the Android device send very brief tones continuously. However, I found that annoying. Another option would be to simply use the phone speaker instead of bluetooth. That works well too; but again can be annoying for folks around me.

I have a tray that holds my ventilator on my powerchair and under that I mounted the three sensors next to my backup camera. One is straight back and the other two are at about 45 degrees right and left.

Here is an image of the final installation:


HC-SR04

Conclusion

This was a fun and useful project.

I was asked if it would work for the blind and I considered working on another project that would help them as well. I decided that I'm retired and so I would not do this. However, I did some preliminary research and here's how I might approach it:

1. The PICO requires such a small amount of power, that a portable battery power pack would last for weeks. However, I think a small USB power bank would easily power a device like this, be easily rechargeable, and also serve as a backup power for the user's phone.
2. I'd probably design the device to be worn around the neck with the sensor's eyes going straight out. The sensor and the PICO would fit nicely in a small package that could easily be 3D printed and hung from the neck. The blind user could then move it right or left as needed; but would measure distance directly in front.
3. For the apps, we could leave them almost the same. However, the distances could be increased to go out several feet instead of just a few inches. Like this app, the sensor/Pico would play tones that would be transmitted to the blind user's cell phone, earbuds, or hearing aids.