What we will learn in this lab

- Some challenges and techniques for implementing and debugging real-time signal-processing algorithms such as PTC.
- An appreciation for how the wide diversity of human physiology makes biomedical algorithms challenging.

Background: big picture lab overview

The big picture of this lab is straightforward. You feed the Nucleo board with an ECG waveform. It then digitizes the waveform and analyzes it to find each QRS complex. It signals each QRS complex by beeping a speaker. It also calculates the time between each new QRS complex and the previous one, and thus displays your instantaneous heart rate on a four-digit seven-segment display.

Everyone's ECG is different, and so locating QRS complexes robustly across many different people is easier said than done. We'll get there in three steps:

- The lab is set up out of the box to replay Joel's canned ECG over and over, and use that as its input. The PTC algorithm as described in class should work fine on this ECG (though your implementation may have bugs).
- Switch to using your own canned ECG. Does it still w
  ork? If so, try some other canned ECGs until you get it to fail. Debug and fix until it works. This will probably involve finding and fixing an algorithmic problem.
- Finally, you can try putting on the ECG pads and going live.

Background: debugging aids

Writing code to implement PTC is relatively easy. You have the metacode, and hopefully changing that to code will be an easy application of the library we've developed. What's more challenging, though, is the debug.

If all the code and algorithms work fine, then you'll hear a beep with every heartbeat – no problem. But what if the output is not beeping as desired? How do you figure out what went wrong?

There are some elephant-in-the-room debug issues here:

- As you've likely learned debugging software in the past few years, non-repeatable software can be excruciatingly difficult to debug. Unfortunately, like most human beings, every one of your heartbeats is slightly different from every other one.
- An ADC is sampling your ECG 500 times per second, and performing filtering and algorithmic calculations at this same speedy rate. Even if you hook up a UART and print out data, your human eyes cannot even remotely examine 500 numbers and

calculations per second to figure out what is going wrong. You need a better way to see the calculations than watching near-meaningless numbers stream by.
- Most of the interesting variables that you would want to look at in order to debug the algorithm are stuck inside the microcontroller, and it's obviously impractical to stick a scope probe on them – or to try and single-step with data flying by so fast.

To help you with the first issue, we've added a task in *lab7_main.c* to replay a canned ECG, such as the ones that you recorded last week. It will replay the recorded ECG over and over, using DAC #1 of the STM32L432's two DACs. As noted above, *lab7_main.c* is initially set up to replay Joel's canned ECG.

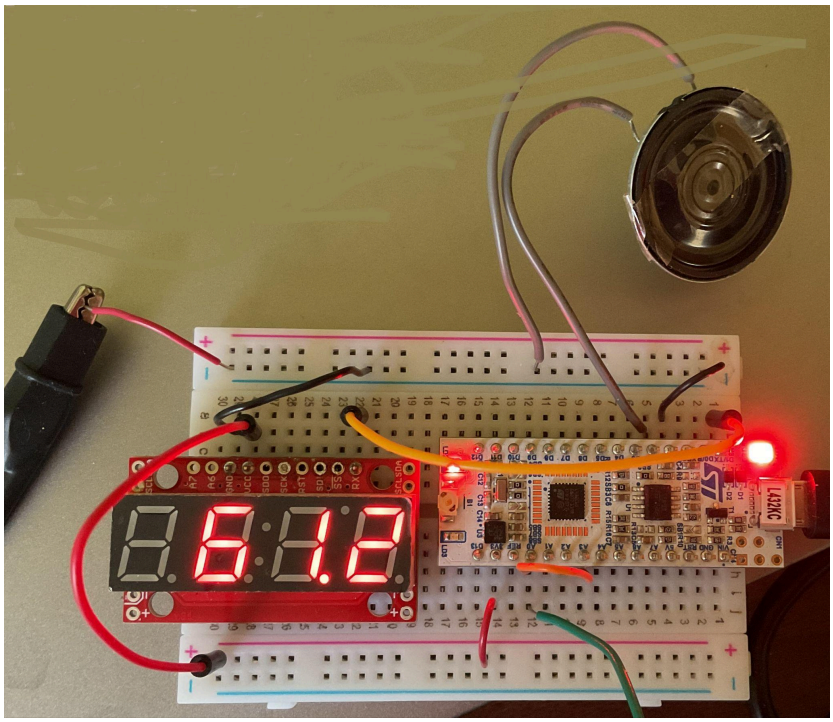
To help you with the second and third issues, you'll learn about multiple methods for dealing with streaming-data debug. More on that later…

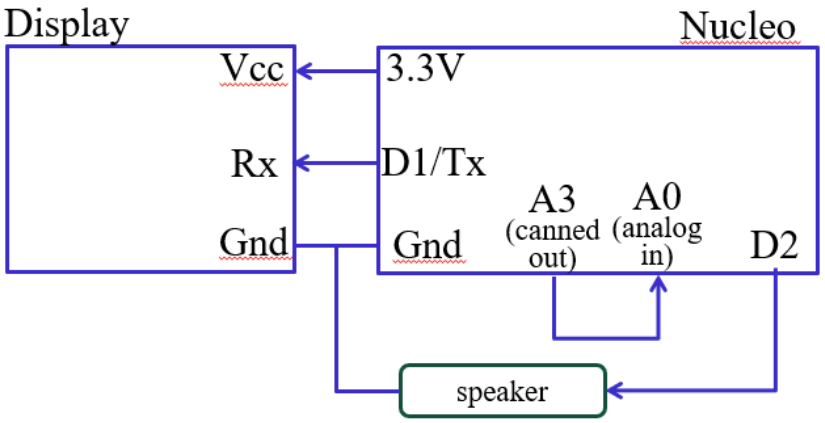
Initial software set up

- As usual, you can build this lab in either of two ways. First, start with the zipped, basic VScode/FreeRTOS environment and add a few new files to it from our website. Second, make a copy of any of our other FreeRTOS labs, delete the extras that you added, and use that as your clean base.
- In either case, once you have a clean VScode/FreeRTOS directory tree set up, rename it to *lab7_PTC* and add the extras for this week. The source files are *lab7_main.c, lib_ADC.c, lib_DAC.c*, *lib_GPIO.c, lib_UART.c* and *ecg_normal_board_calm1_redone.c_txt*, with the usual include file *lib_ee152.h*. You can then build the project as usual.
- Edit the file *lab7_PTC/include/FreeRTOSConfig.h* and change the parameter *configTOTAL_HEAP_SIZE* to 5000 (each FreeRTOS task that we use will create an object that requires memory from the heap, and this lab uses lots of tasks). If your code hangs, you may have forgotten this step :-).

Initial hardware set up

Next is the hardware setup. You'll need a few more parts than normal. First, as usual, the Nucleo board. Next, the LCD display. And finally, a speaker for the usual ECG "beep-beep" audio 🙂. The picture and schematic are below:





- DAC #1 drives the canned ECG out on pin A3. The ADC expects its input on pin A0, so you must connect those two with a wire.
- Attach a small speaker to get a beep at every QRS you detect. As you can see in the function *task_beep*(), audio output comes from pin D2. You would thus hook up a speaker between pin D2 and ground. It signals each QRS by driving a square wave to the speaker via GPIO.

- You will probably also want to observe the ECG signal on a scope. Perhaps the easiest way to do so is to attach the display and the Nucleo board to the same breadboard, as shown in the picture below. The schematic is below the picture.

Part 1*: *running with Joel's canned ECG

With your initial hardware and software set up, you should see the canned ECG on the scope, as well as the LED flashing.

We took a quick look at the code in class; take a bit deeper look now yourselves to see how it works.

Our debug aid uses DAC #1 to drive a canned ECG out on pin A3. After doing the usual build-and-flash routine, you should see the ECG on the scope (like last week, and set the scope in scan mode at roughly .2 seconds per division, rather than attempting to use a trigger). If you're lucky, you'll also hear a beep at each QRS, and the LCD display will show a heart rate of roughly 60 beats/min.

Checkout for part 1 is just to show us that it works.

Part 2: learning to debug with the second DAC and/or GPIO pins

In lab 3, you learned how to use FreeRTOS instrumentation to debug high-speed interactions between tasks. Now it's time to start learning how to debug the processing of high-speed data streams. As mentioned above in the Background section, this is our elephant-in-the-room problem for this lab.

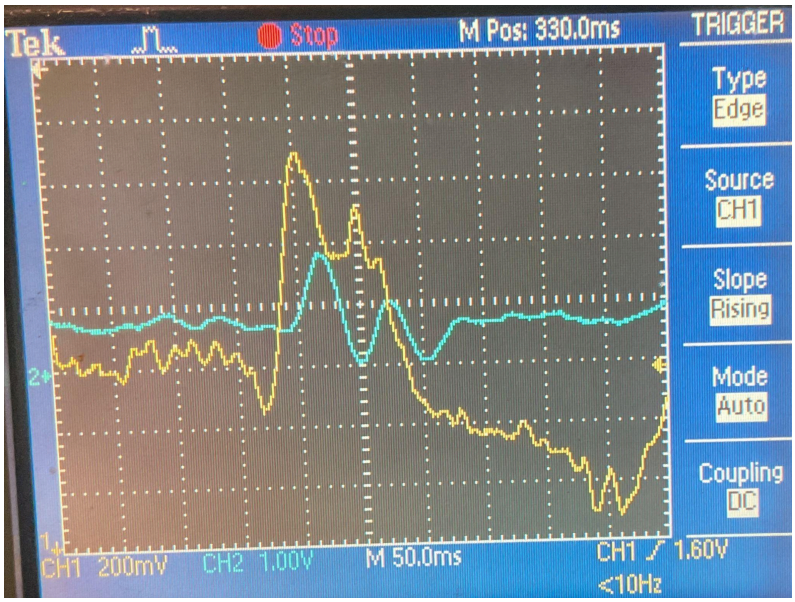
To debug algorithm problems, it's extremely helpful to see the signals involved – but they are computed on the fly inside of the CPU, and you cannot stick a scope probe inside of the CPU! Instead, our first useful technique will be to send the signals back out of the chip so that you can view them on a scope.

Lab7_main.c has a few lines of code to use DAC #2 as a debug aid. Specifically, you can pick any signal you want to examine and drive it out to DAC #2 every 2 ms. To start with, find the line

```
analogWrite (A4, sample);
```

in *task_main_loop*(). If you use your scope's channel 2 to view Uno pin A4, you'll see a real-time plot of *sample* on channel 2, lined up with the plot of the ECG input on channel 1. Since *sample* is merely the ADC version of the canned ECG signal that's on channel 1, the two signals "should" be near identical. It most likely will, one the other hand, be mostly gobbledy-gook. Clearly, something has gone wrong – and your first job is to figure out what and fix it, and then show Joel or a TA.

Once you've figured that out, it's time to move on to a slightly harder problem. This time, use the same facility to view the signal *deriv_2*. Again, it may not make a lot of sense, and again your job is to fix it so that it does make sense (or at least makes as much sense at possible!). Here's roughly what the correct signal should look like:



While using a DAC to bring an internal variable out to a pin as a voltage is great, it may not be great enough. With only two DACs, you can only create two signals at once. Furthermore, one of the DACs is hardwired to create the repeatable ECG waveform. You are thus limited to displaying this waveform and one other signal. You may want more signals, or at least a different choice of signals.

Many of the signals you might want to view are digital (i.e., Boolean) rather than analog. In this case, you can drive them onto a GPIO pin to avoid the issue of having only two DACs. This can be especially helpful if you have one of the scopes that can display four channels rather than two. And so your third job is to display both *sample* and *dual_QRS* (using a DAC for *sample* and a GPIO pin for *dual_QRS*).

Demonstrate these three tasks (correctly displaying *sample* and *deriv_2*, and simultaneously displaying both *sample* and *dual_QRS*) to Joel or one of the TAs.

Part 3*: *using your own canned ECG

Next up is using your own canned ECG from last week. Copy your ECG data file into your *src* directory, and change the **#include** statement just above *task_canned_ECG*() to use your data file. With this done, rebuild and re-flash the code.

If you're lucky, everything will still work fine! If not, it's time to debug. In fact, the code we've given you has at least one major algorithmic shortcoming, so it's likely that the

algorithm will not quite work for your canned ECG. Thus the fun begins again – may the force be with you :-)

In order to make your debugging job easier (and to learn some nice new debug techniques), the ultimate in debug power comes from doing your debug fully on a host, with the ability to use a standard IDE as well as any graphical plotting tool you like. Obviously, a Linux host does not have ADCs, DACs or FreeRTOS, and so some amount of software McGyvering is needed. The most obvious strategy is to throw away all of *lab7_main.c* except for *task_main_loop* (which then becomes our new *main*) and the analysis functions it calls. Instead of reading from an ADC, we add a custom version of *analogRead*() that pulls numbers from a file (you can find this in the file *lab7_main_host.c*). With this done, you can use any IDE to debug your simple single-threaded code. Perhaps the easiest way to plot signals is to just print them to a file from C++, with one line (and as many numbers as you like) per timepoint. Then use the utility *lab7_plot.py* to plot it.

This section has two checkouts. First, demonstrate that your canned ECG runs through your bug-fixed PTC algorithm fine. Second, show us how you used host-based debugging to help.

***Part 4*: board-level hook up to use your on-the-fly ECG**

If you've gotten this far, then you figured out how to improve PTC so that it works with your canned ECG. Congratulations! Your next step is to move away from your canned ECG and go live.

In order to use your ECG coming straight from your heart without being pre-recorded, we'll need to use the AD8232 board as a preamp, just as we did in lab #6. You should wire up and power the AD8232 as we did for lab #6.

Since you'll be using the AD8232 to drive an ECG into the Nucleo board, you will no longer need to use an internal DAC to do so. Thus, before driving pin PA1 from the AD8232 board, you must first disconnect the wire that drives pin PA3 into pin PA1. And that's pretty much it, other than rebuilding, flashing and seeing what happens.

Check-out summary

1. Part 1: show that you've correctly assembled the board and it works fine for Joel's canned ECG.

2. Part 2
    a. Correctly displaying *sample* and *deriv_2*. For the first two, explain what the problem was originally, and how you fixed it. How do you find the correct min and mix range on data that's streaming by so fast?
    b. Simultaneously displaying both *sample* and *dual_QRS* on a scope.

c. It may seem odd that PTC works fine on Joel's abnormal ECG and fails on your presumably normal ECG. Can you argue that this is less surprising than it may seem? Hint: remember what we said in class about how the algorithm was tested.

3. Part 3: a correctly-functioning PTC algorithm on your own canned ECG, and demonstrating your use of host-based debug.

4. Part 4: correct functioning with your on-the-fly (not prerecorded) ECG.

What to hand in

Nothing :-)