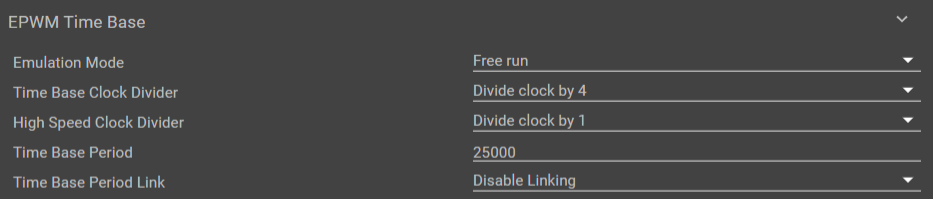
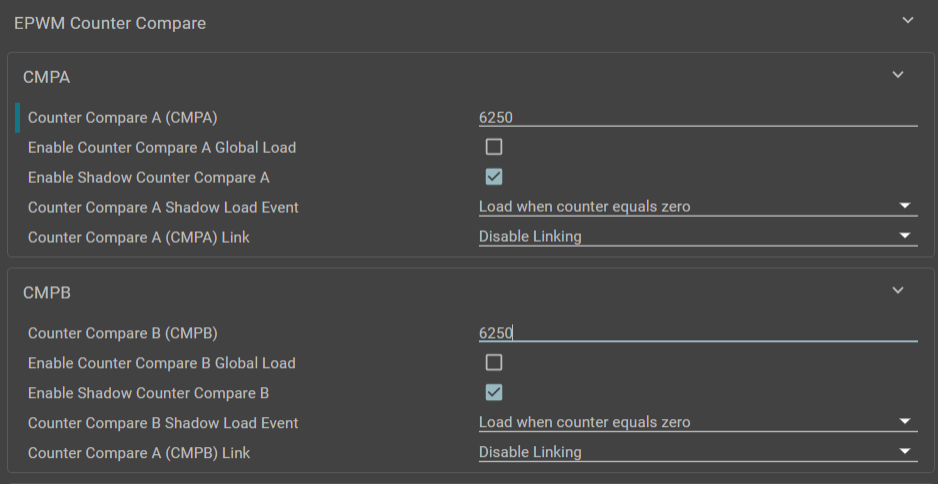
# PWM

**How to adjust PWM duty cycle**

****



Change Counter Compare A/B (CMPA/B) to % of the Time Base Period to get ***inverted*** duty cycle

**Ex:**

25% of 25000 = 6250 for a 75% duty cycle

**PWM Frequency**

Default clock rate

* 100MHz (100us period)

Time Base Clock Divider

* Divides 100MHz
* Ex. Divide clock by 4 => 25MHz clock (400us period)
* Ex. Divide clock by 128 => 781,250 Hz clock (1.28ms period)

Time Base Period

* Must be between [0, 65535] = [0, 216-1]
* Number of clock cycles for the period

Total equation to find frequency of PWM:

**To adjust PWM Duty Cycle at runtime:**

The sysconfig tool auto-generates certain .c and .h files. You can find these files in the Project Explorer under Generated Source -> SysConfig

In ti\_drivers\_open\_close.c, in the function “Drivers\_epwmOpen()”, functions are written to set each of the register values for Time Base Period, Base Counter, Clock Prescaler, etc.

Under the “/\* Counter Compare \*/” line, copying the EPWM\_setCounterCompareValue(...) function and its parameters into the epwm\_hr\_duty\_cycle.c file allows you to change this register value during code execution. The last parameter is the value of the CMPX register.

I inserted the following line in epwm\_hr\_duty\_cycle.c, on line 110 within the while(numIsrCnt > 0) loop:

EPWM\_setCounterCompareValue(CONFIG\_EPWM\_BASE\_ADDR, EPWM\_COUNTER\_COMPARE\_A, numIsrCnt % 25000U);

Where the Time Base Period is 25000 in sysconfig

The numIsrCnt % 25000U basically sweeps the duty cycle from 0% to 100% while the program executes. numIsrCnt refers to the number of times the short timer will expire, and it depends on the frequency and runtime variables set in the code.

Other useful functions for accessing these registers without using direct memory access/dealing with base addresses can be found within the ti\_drivers\_open\_close.c file, since the addresses for interfacing with each PWM module are set automatically by sysconfig. You can find (very very short) descriptions of these functions in the API reference, under

API reference -> APIs for SOC specific drivers -> APIs for EPWM

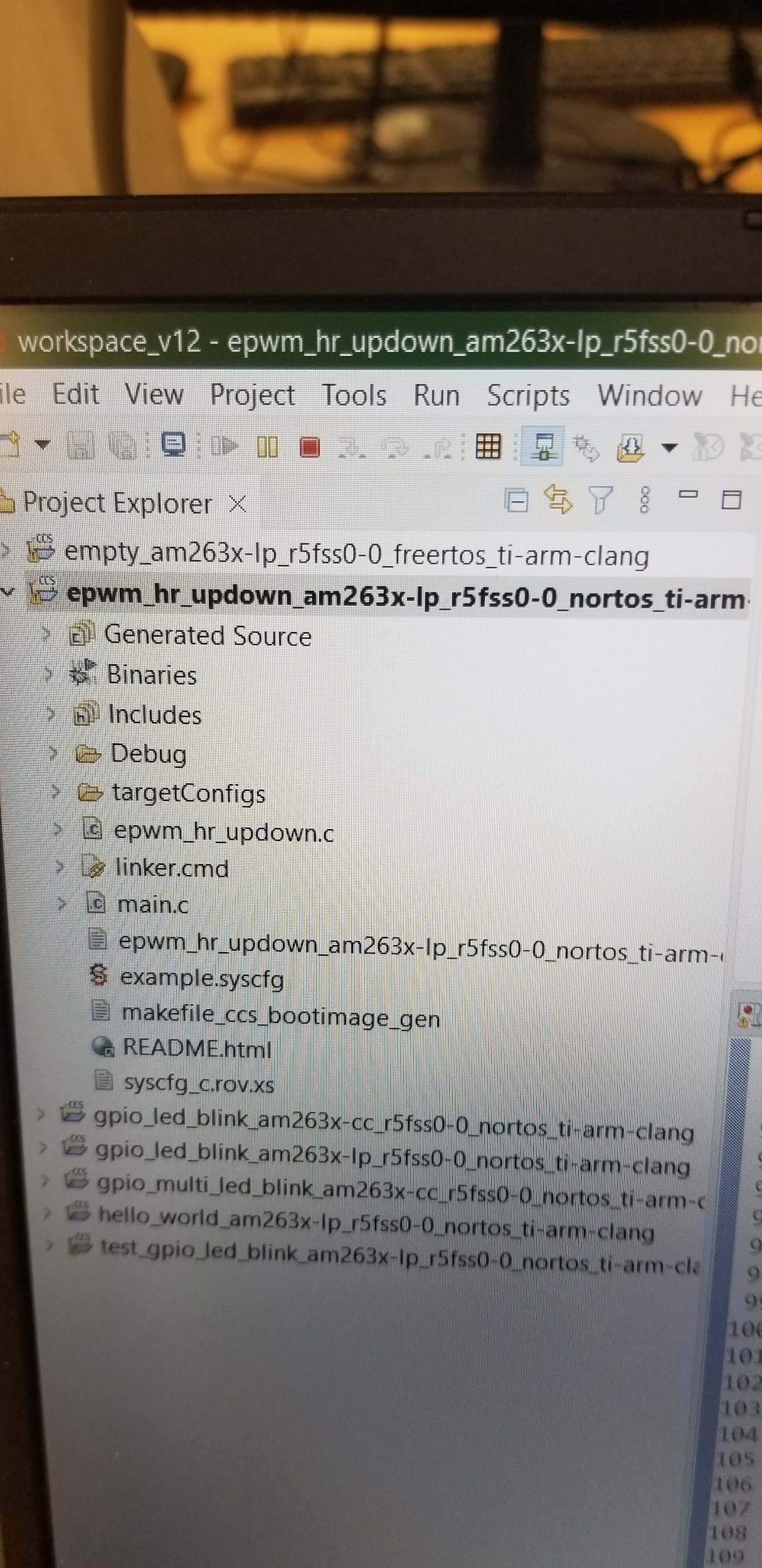
and scrolling down quite a ways.

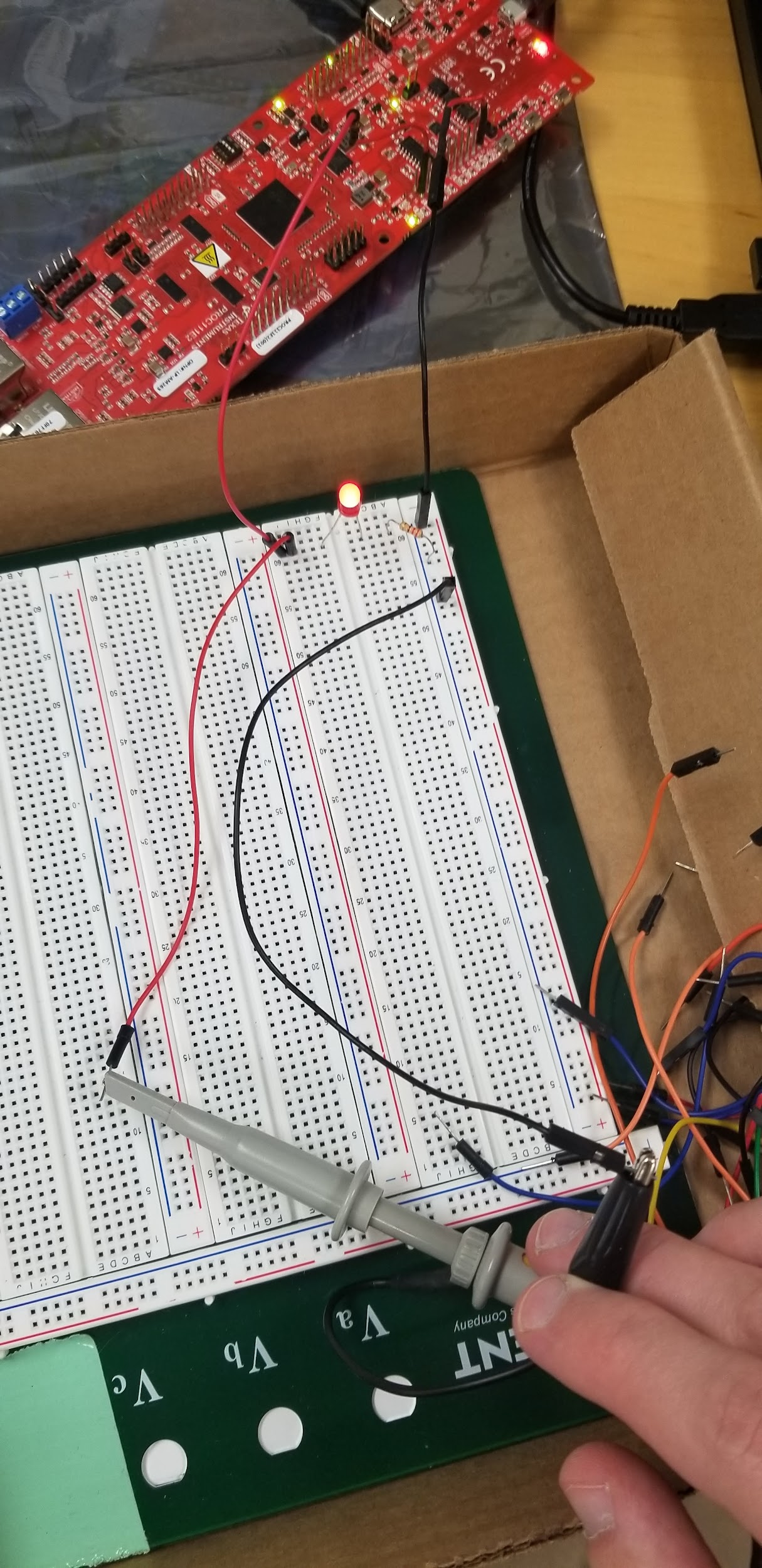
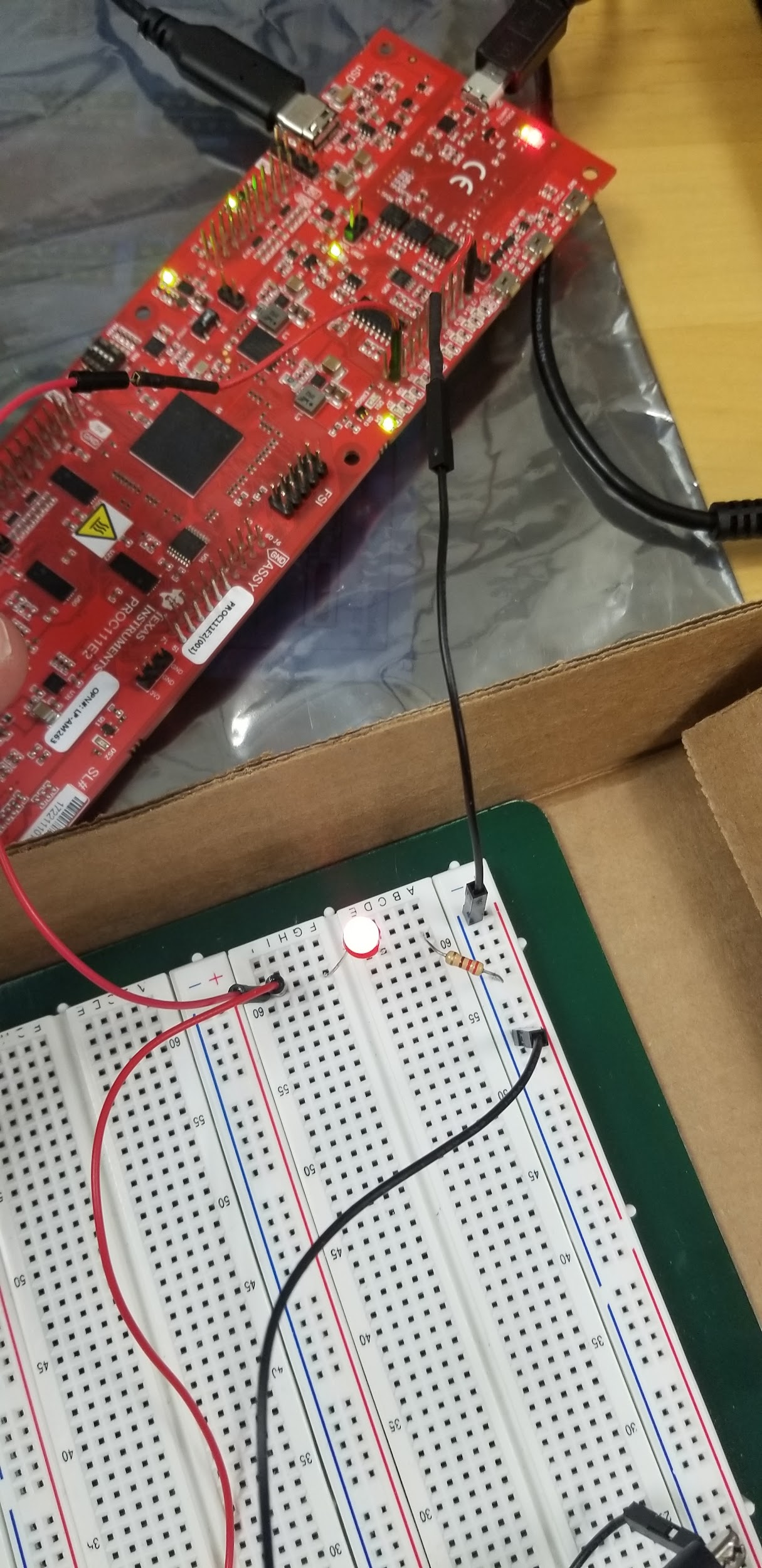
In short, if we know the base period (which will correspond to the frequency of PWM from the equations above), which should not change throughout motor execution, we can hardcode that value somewhere in our code. Then, the duty cycle can be set on-the-fly as some fraction of this number, which we should know ahead of time

**Example 1: HR updown**

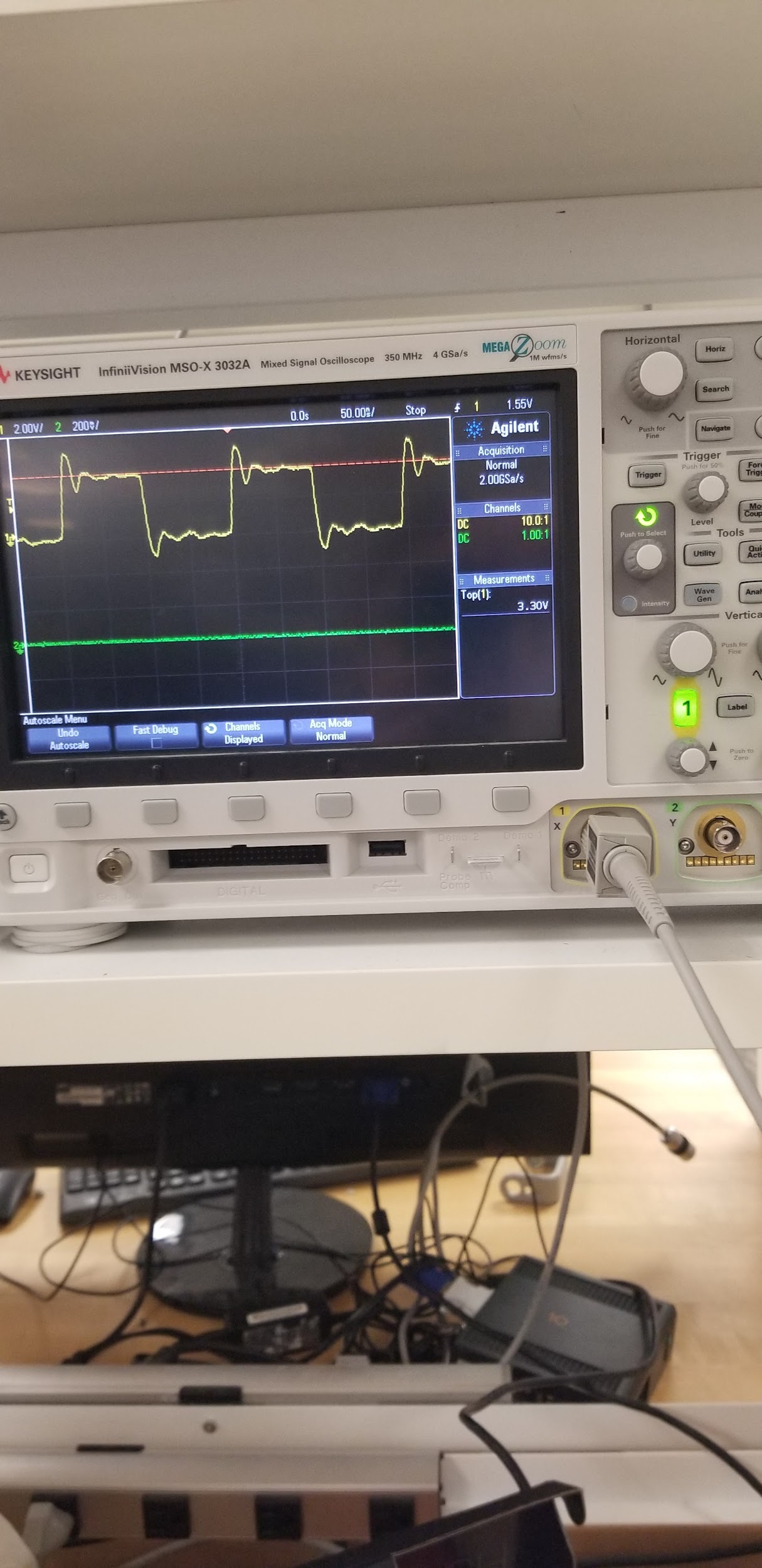
So far, I have been able to view the PWM signal on the scope, but I don’t understand in the code how the frequency/duty cycle is set. This is just one of the PWM signals on pin J2.11 (PWM0\_A). I probed this signal and also connected it to an LED to see if it was on. I used the oscilloscope to view the signal, you can press “auto scale” (right below the “Single” button on the top right) to adjust to the right size once the example is running. Then, you can press run/stop or Single to trigger the next edge (I think).

The example never ended for me, even when I hit the red square button to stop execution. I unplugged the power from the board and plugged it back in to end. At that point the oscilloscope stopped showing the square-ish waveform.





These are the leads to the oscilloscope. There’s one BNC connector that I connected to channel 1 (Yellow “X” channel) on the oscilloscope, and I used the cable that has these two probes on the end. One is for probing the PWM signal coming from J2.11, and the other is for probing ground.



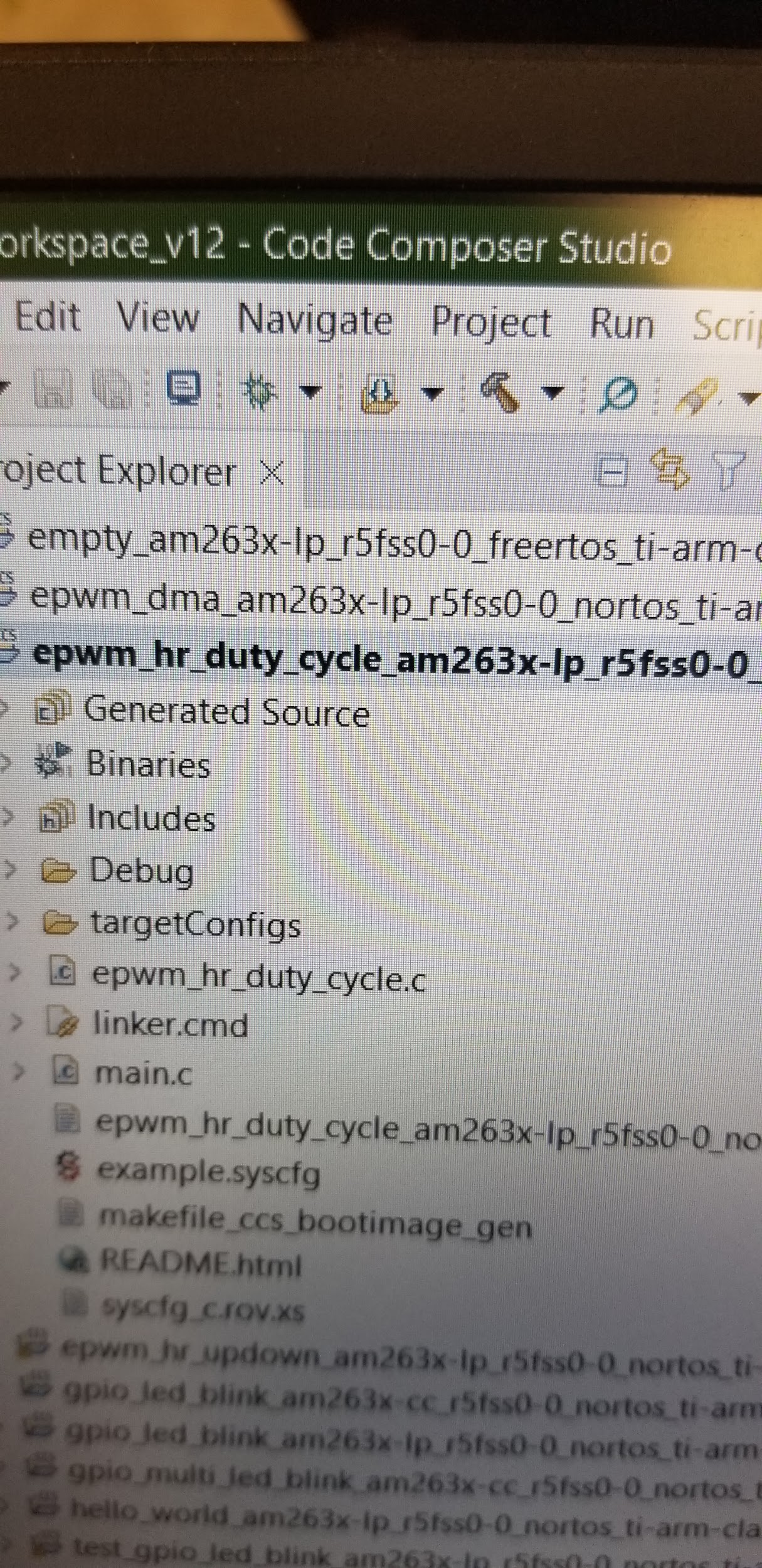
This is the output that I see. It looks like the top is about 3.3 V, so that’s the high level for the PWM signal.

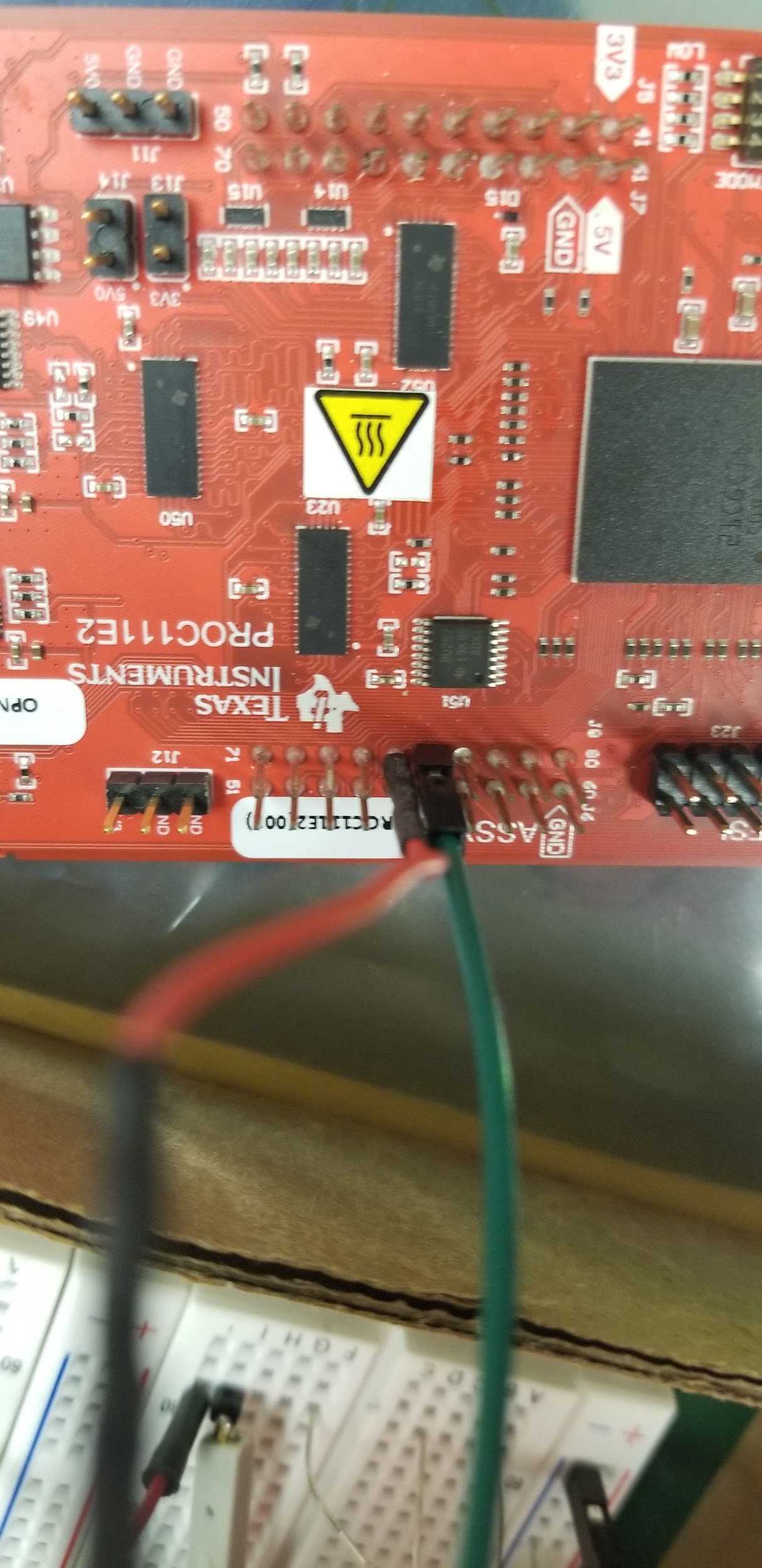
**Example 2: HR Duty Cycle**

I got this example to work, but when I changed the parameter APP\_EPWM\_OUTPUT\_FREQ from 1\*1000 to 2\*1000 or 10\*1000, I did not see a change in the frequency of the signal measured on the oscilloscope. The only thing I observed is that the code ran for 2x or 10x longer when I made those changes. I can’t find anything in the code that affects the duty cycle. We did change the APP\_EPWM\_RUN\_TIME from 60 to 120 and it ran for 2 minutes instead of 1. That’s the only intentional measurable change we’ve been able to make so far.

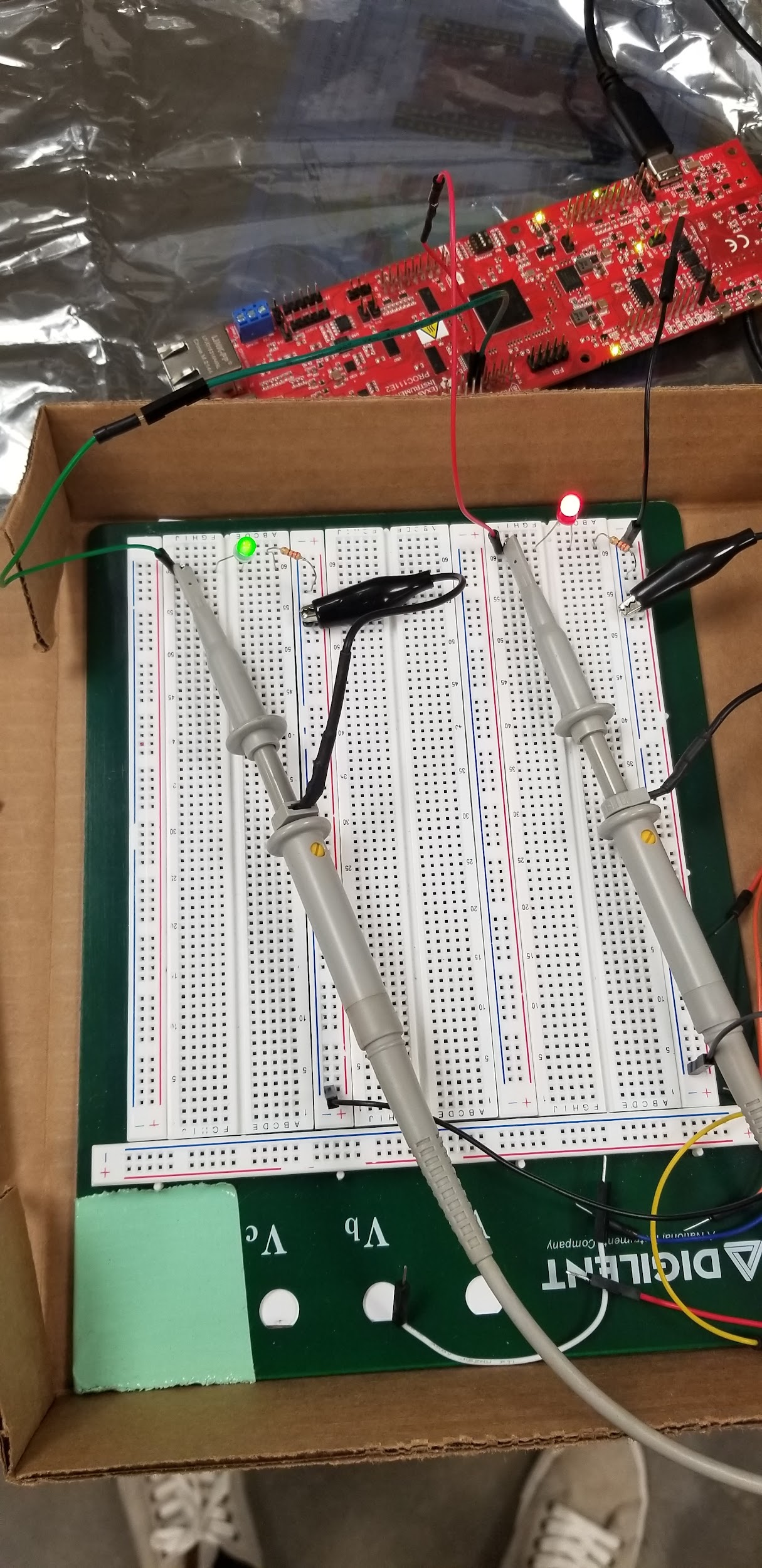
If you figure out how to change the duty cycle, that’s amazing. Otherwise we can look at it all together.

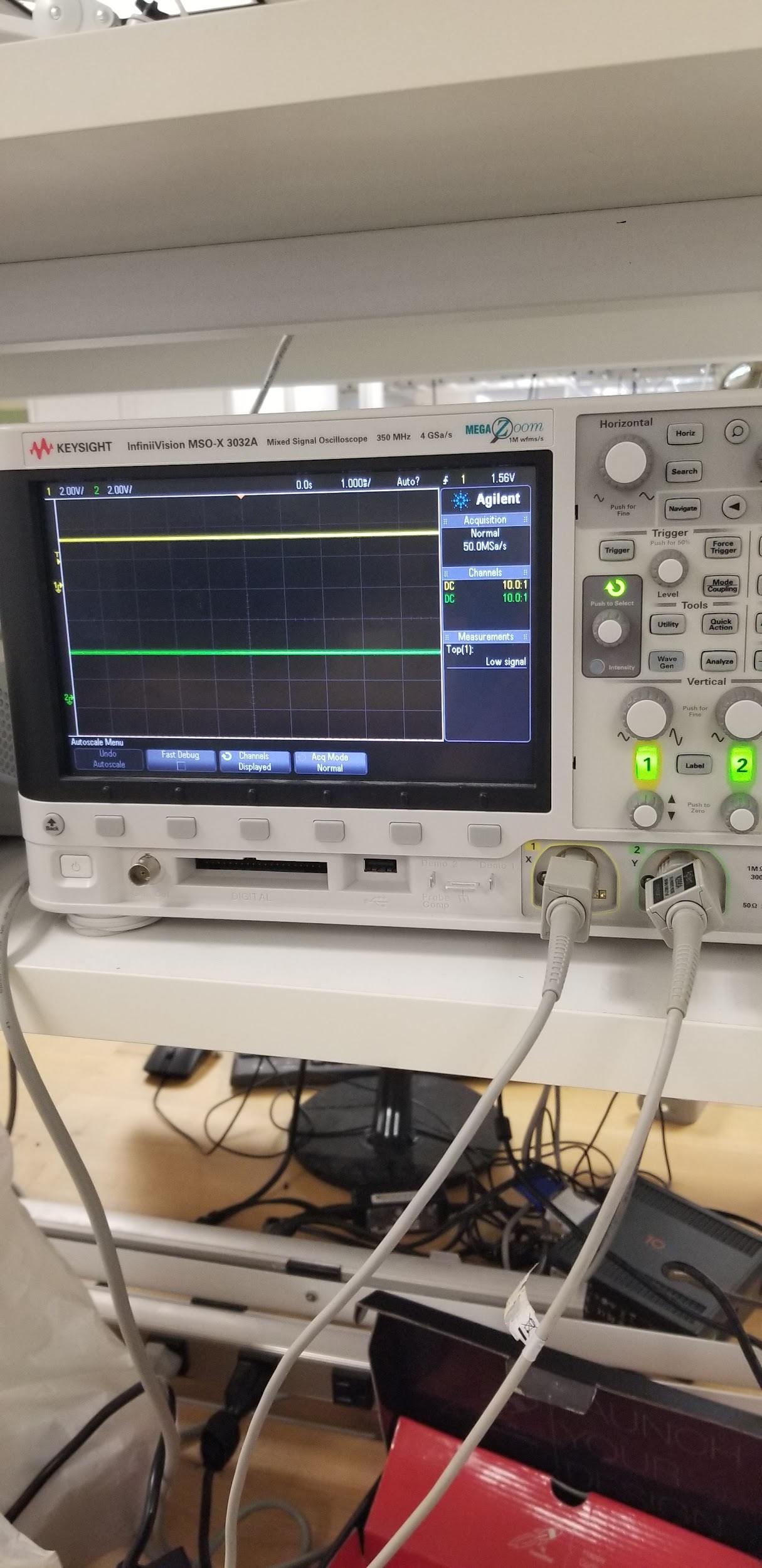
For the board configuration, I just doubled the previous connections. 2 PWM signals, 2 LEDs, etc. I’ll leave it all connected just how it was for this example. I have little stick-out pins connected to probe from the oscilloscope so I don’t need to stick a wire out of the board (so it looks cleaner).





These are the pin connections for the PWM signals, PWM9\_A and PWM9\_B. These are in the readme in the project, or you can look at sysconfig, see that it's PWM9, and look at the LP user guide to check the physical pins. They should be 75 and 76.





**PWM DMA Example**

This example uses the DMA module (direct memory access) to change the register values for the duty cycle and frequency registers.

In “ti\_drivers\_open\_close.c”, a file generated by the sysconfig

EPWM\_setTimeBasePeriod(CONFIG\_EPWM0\_BASE\_ADDR, 20000);

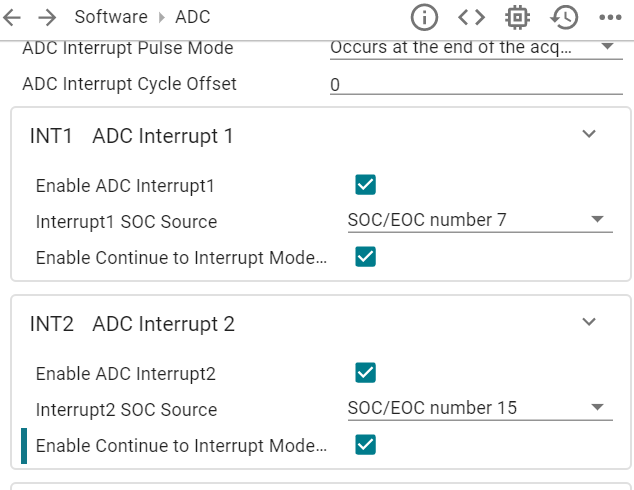
# ADC

Used the “adc\_soc\_continuous” example

We set up a circuit with a potentiometer to allow for varying voltage to be read.

For the AM263x, the only external connection needed is just a wire connected to pin 24 (off of J3). The code as given will not fully work as there is a portion in the code that will infinitely loop

If we set the sysconfig to be the following, the code should run.



This program reads all the input voltage values and stores them in an array before printing out sample voltage values, so make sure that the voltage is being adjusted at the beginning of the program (I’ll see if I can modify the code so that the adjustment can be better observed)

# PWM/ADC Potentiometer Control

In the SIMBA google drive folder, download the folder “pot\_pwm” and extract it. Then, in code composer, File->Open Projects from File System. Click “Directory” and select the extracted folder, then import it.

For LP connections:

J75: PWM output, connect to an oscilloscope

J23: ADC input, connect to a potentiometer output that sweeps between 0 and 3.3V

For the potentiometer, connect J23 to the middle leg (for the 3006P potentiometers). Connect a 3.3V power supply across the other two legs of the potentiometer. Make sure the power supply, launchpad, and oscilloscope ground are all connected, and one leg of the potentiometer is connected to ground.

Then, run the example and turn the potentiometer CLOCKWISE to increase duty cycle, COUNTER CLOCKWISE to decrease duty cycle.