Ethan Bonpin

Professor Fund

EE 4323

February 28, 2022

New Part Assignment - Capacitive Touch Sensor Breakout

Section I - Connecting and Verifying

This new part assignment covers the Adafruit Standalone 5-Pad Capacitive Touch Sensor Breakout, featuring Atmel's *AT42QT1070* Touch Sensor IC. When oriented normally (text facing right-side up on a breadboard, for example), this breakout has five pins on the left 'bank' (or, in this case, vertical column) and five pins on the right. The left 'bank' pins consist of *Gnd* (Ground), as well as *0*, *1, 2, 3* and *4*. To interface with this breakout, we will use a Raspberry Pi Zero W with the power/GPIO pin headers soldered on. Appropriate jumper wires are used to connect the Pi to the breadboard (which will be used to test for functionality). The ground pin on the breakout can be connected to the ground rail (labeled '-') on the breadboard. We will also need to use a wire to bridge any open spot on the ground rail to any unused ground pin on the Pi (referring to *www.pinout.xyz* as a diagram showing the purpose of each pin). Left bank pins 0, 1, 2, 3, and 4 are used for the five capacitive touch inputs. For testing purposes, we can use five male-to-male jumper wires with one side of each wire left disconnected and the other side connected to the left bank pins labeled 0-4. After we finish wiring the circuit, when we touch the disconnected jumper wire for the pin that the wire is connected to, a green LED on the breakout located at the same pin number on the other side will illuminate. It is crucial to note that this breakout can only sense one capacitive touch input at a time. This is to protect against multiple touch inputs being used at the same time (which can cause undesired behavior).

The right bank pins consist of *VDD, 0, 1, 2, 3,* and *4*. Right bank pins *0, 1, 2, 3,* and *4* are used for the five capacitive touch outputs. These can be connected to almost any five GPIO pins on the Pi using jumper wires (except for GPIO2 and GPIO3, using BCM pin numbering). We cannot use these two pins as they have permanent pull-up resistors. This sensor breakout will need pull-down resistors so that the input read on the Pi is not left in a 'floating' state (could fluctuate between high and low). As such, the right bank pins have been connected to five vertically consecutive free GPIO pins on the Pi. These are GPIO5, GPIO6, GPIO13, GPIO29, and GPIO26. We will handle the software for the pull-down resistors on these pins in a later section of this report. At this point, the wiring should look similar to this:

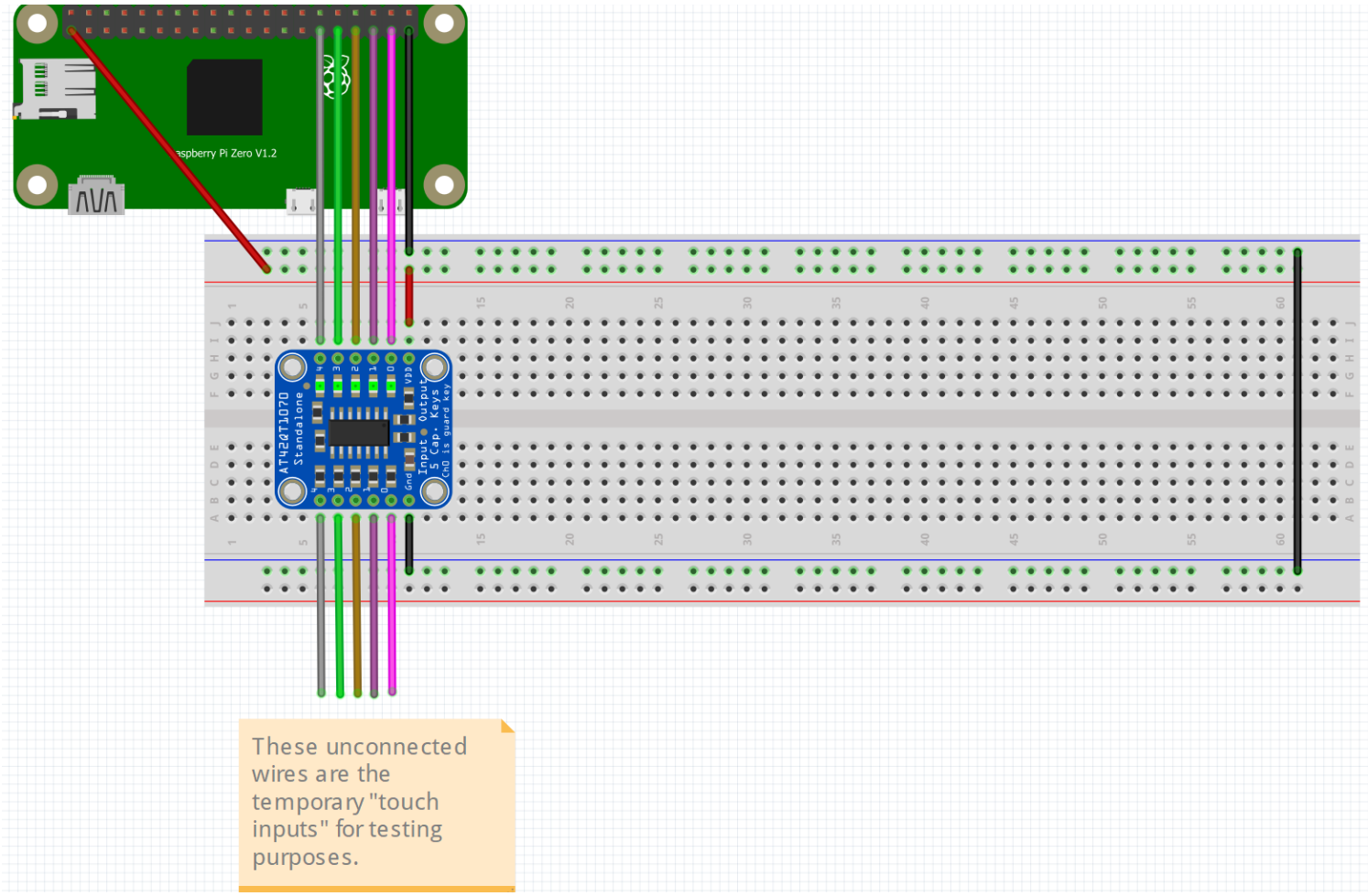

Fig. 1: Fritzing wiring diagram of Pi connected to breakout and breadboard

With the wiring for the breadboard, breakout, and Pi complete, the Micro USB power can now be connected to the Raspberry Pi to power it (and the circuit). Now, when we touch the end of one wire at a time for each of the wires disconnected on one side (input), the LED of the pin on the opposite side of the breakout will light up, as shown below:

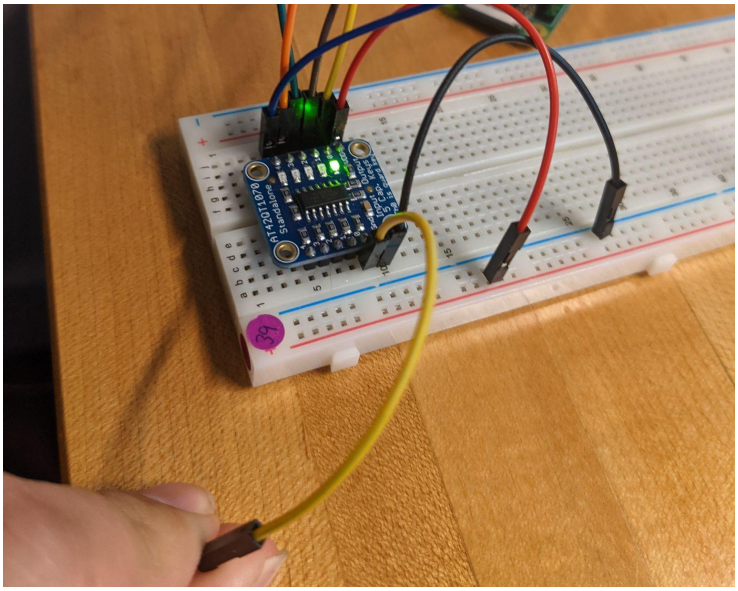

Fig. 2: Touching a jumper wire connected to breakout input (other wires removed for clarity)

Since we have verified functionality of each input pin and have observed how the LED on the opposite side lights up, we have verified that this part is properly working and not damaged or malfunctioning.

Section II - Describing Library Functions for Basic Functionality

Now that we have verified the basic operational functionality of this part, we now have to create library functions so that the output pins that are wired to the GPIO pins on the Pi can actually be used to detect when a specific input wire detects touch. For basic functionality purposes, we will need a function that will set up the GPIO pins accordingly, a function to detect input, and a function to notify the user of this input (three basic functions in total).

```
def setup_captouch():
 GPIO.setmode(GPIO.BCM)

 global out0 #declare global so we can call it from another function
 global out1
 global out2
 global out3
 global out4

 out0 = 26 # output 0 is BCM26
 out1 = 19 # output 1 is BCM19
 out2 = 13 # output 2 is BCM13
 out3 = 6 # output 3 is BCM6
 out4 = 5 # output 4 is BCM5

 GPIO.setup(out0, GPIO.IN, pull_up_down = GPIO.PUD_DOWN) # configure resistor on pin as pulldown
 GPIO.setup(out1, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
 GPIO.setup(out2, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
 GPIO.setup(out3, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
 GPIO.setup(out4, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
```

Fig. 3: setup_captouch library function

Before the setup function, we import *RPi.GPIO as GPIO* and *time*, as well as *sys*. Inside the setup function pictured above, we first set the pin numbering scheme to BCM, as that is the convention that will be used. Next, we declare global variables for the five pins so they can be called from the detection function. After that, a variable is set for the five output pins (*out0* through *out4*) with their respective BCM pin number in each. As mentioned earlier, the outputs on the breakout are connected to BCM26, 19, 13, 6, and 5 so they are set accordingly here. Next (and last), we use *GPIO.setup()* to set up each GPIO pin as an input for the Pi and set *pull_up_down* to *GPIO.PUD_DOWN* (this will set the configurable pullup/pulldown resistors to pull down on the inputs on the Pi to avoid a floating state). This is all that is needed to set up the breakout for basic functionality.

After the *setup_captouch()* function, we now need to set up a function to detect when one of the jumper wire inputs to the breakout is touched. This function is called *detect_captouch* and will use *RPi.GPIO* to detect when one of the inputs is triggered. The code for this function is as follows:

```
def detect_captouch():
 GPIO.add_event_detect(out0, GPIO.FALLING, callback = notify_user, bouncetime = 200) # add bouncetime to prevent false alarm
 GPIO.add_event_detect(out1, GPIO.FALLING, callback = notify_user, bouncetime = 200)
 GPIO.add_event_detect(out2, GPIO.FALLING, callback = notify_user, bouncetime = 200)
 GPIO.add_event_detect(out3, GPIO.FALLING, callback = notify_user, bouncetime = 200)
 GPIO.add_event_detect(out4, GPIO.FALLING, callback = notify_user, bouncetime = 200)
```

Fig. 4: detect_captouch function

This function is also rather simple. It consists of five *GPIO.add_event_detect* function calls (one for each touch sensing pin). Inside each call is the pin number for the respective input, *GPIO.FALLING* (which means detect on falling edge), a callback to *notify_user* (a function that outputs what input pin was triggered) as well as *bouncetime = 200* (which adds a 200ms delay between when a falling edge is detected and when it is detected again to prevent false alarms).

The last library function (as mentioned previously) notifies the user which pin was 'touched'. This function is called *notify_user* and takes *pin* as the input. This input is passed over from *GPIO.add_event_detect* in the previous function. This function simply prints the pin number that was pressed as follows:

```
def notify_user(pin):
 print("Pin", pin, "has been pressed.")
```

Fig. 5: notify_user function

Now that we have all library functions for the basic functionality of this capacitive touch sensor written, we need to complete the program to be able to run and test it. This means that we will need to add the following:

```
setup_captouch()
detect_captouch()

try:
 while True:
  time.sleep(0.01)
except KeyboardInterrupt:
 GPIO.cleanup()
 sys.exit()
```

Fig. 6: Additional Code for Testing

This code consists of a call to run the setup and detection functions, then endlessly sleep for 0.01 seconds until interrupted by keyboard (done by pressing Ctrl + C on the keyboard). If interrupted by keyboard, we run *GPIO.cleanup()* and *sys.exit()* to clean up the pins we configured then exit the program.

To run this simple test program, we navigate to the directory in a command line and type *python3 captouch.py* (assuming the file is named *captouch.py*). If we touch a pin, we see that it appears in the command window. Here's an example of this behavior after touching some pins (being sure to confirm that the pin pressed matches the pin outputted):

```
^Cpi@raspberrypi:~/Desktop $ python3 captouch.py
Pin 5 has been pressed.
Pin 5 has been pressed.
Pin 5 has been pressed.
Pin 5 has been pressed.
Pin 5 has been pressed.
Pin 13 has been pressed.
Pin 26 has been pressed.
Pin 19 has been pressed.
Pin 6 has been pressed.
Pin 5 has been pressed.
Pin 5 has been pressed.
Pin 13 has been pressed.
Pin 13 has been pressed.
Pin 13 has been pressed.
Pin 13 has been pressed.
Pin 13 has been pressed.
Pin 13 has been pressed.
Pin 13 has been pressed.
Pin 13 has been pressed.
Pin 26 has been pressed.
Pin 26 has been pressed.
Pin 13 has been pressed.
Pin 6 has been pressed.
Pin 6 has been pressed.
Pin 5 has been pressed.
```

Fig. 7: Command line output from touching several pins randomly

At this point, we have wired the breakout to the Pi as well as written a program to test it. This means that this new part has been fully tested and is ready for project implementation.

Section III - Understand and explain in depth

Now that we have confirmed that the part is fully working and that the outputs work as expected, it is time to gain an in-depth understanding of how this touch sensor works on a lower-level scale.

This breakout board uses the *AT42QT1070* touch sensor IC in standalone mode, meaning it will communicate normally without the use of I2C. The other mode, *comms mode*, allows for the use of I2C communication as well as the use of more advanced features which are found in the datasheet:

2.2 Modes

2.2.1 Comms Mode

The QT1070 can operate in comms mode where a host can communicate with the device via an I^2C-compatible bus. This allows the user to configure settings for Threshold, Adjacent Key Suppression® (AKS®), Detect Integrator, Low Power (LP) Mode, Guard Channel and Max Time On for keys.

2.2.2 Standalone Mode

The QT1070 can operate in a standalone mode where an I^2C-compatible interface is not required. To enter standalone mode, connect the Mode pin to Vdd before powering up the QT1070.

In standalone mode, the start-up values are hard coded in firmware and cannot be changed. The default start-up values are used. This means that key detection is reported via their respective IOs. The Guard channel feature is automatically implemented on key 0 in standalone mode. This means that this channel gets priority over all other keys going into touch.

Fig. 8: Datasheet capture of 'modes' section explaining Comms vs. Standalone modes

Behind each touch input on the breakout is a 10 kOhm resistor wired in series before the *AT42QT1070* IC. The use of these resistors helps reduce noise and prevent electrostatic discharge, as well as other forms of digital interference.

In between the output pins of the IC and the green status LEDs on the breakout board are five 470 ohm resistors wired in series. These limit the current going to the LEDs to prevent damaging them.

The input power, labeled as *VDD*, has a capacitor between the power input and the IC's power input. This is done to prevent anomalies described by the datasheet such as "device oscillation, high current consumption and erratic operation". The datasheet does state that any input between 1.8VDC and 5.5VDC can be used, so even though we used the 3.3V power output on the Pi, we can use the 5V power output if desired.

As mentioned previously, the output pins will go low (falling edge) when the capacitive touch input is detected on the input. Now, we can observe this behavior by connecting the *Analog Discovery 2* to our circuit. To do this, we connect the *1+* pin to an open pin on the breadboard on the output side (wired in series) and the *1-* and *Ground* pins of the *Analog Discovery 2* go to the same row where the ground from the Pi comes in on the breadboard (so we share a common ground between the circuit and the AD2). Below is an example of this behavior on the scope of the *Analog Discovery 2*:

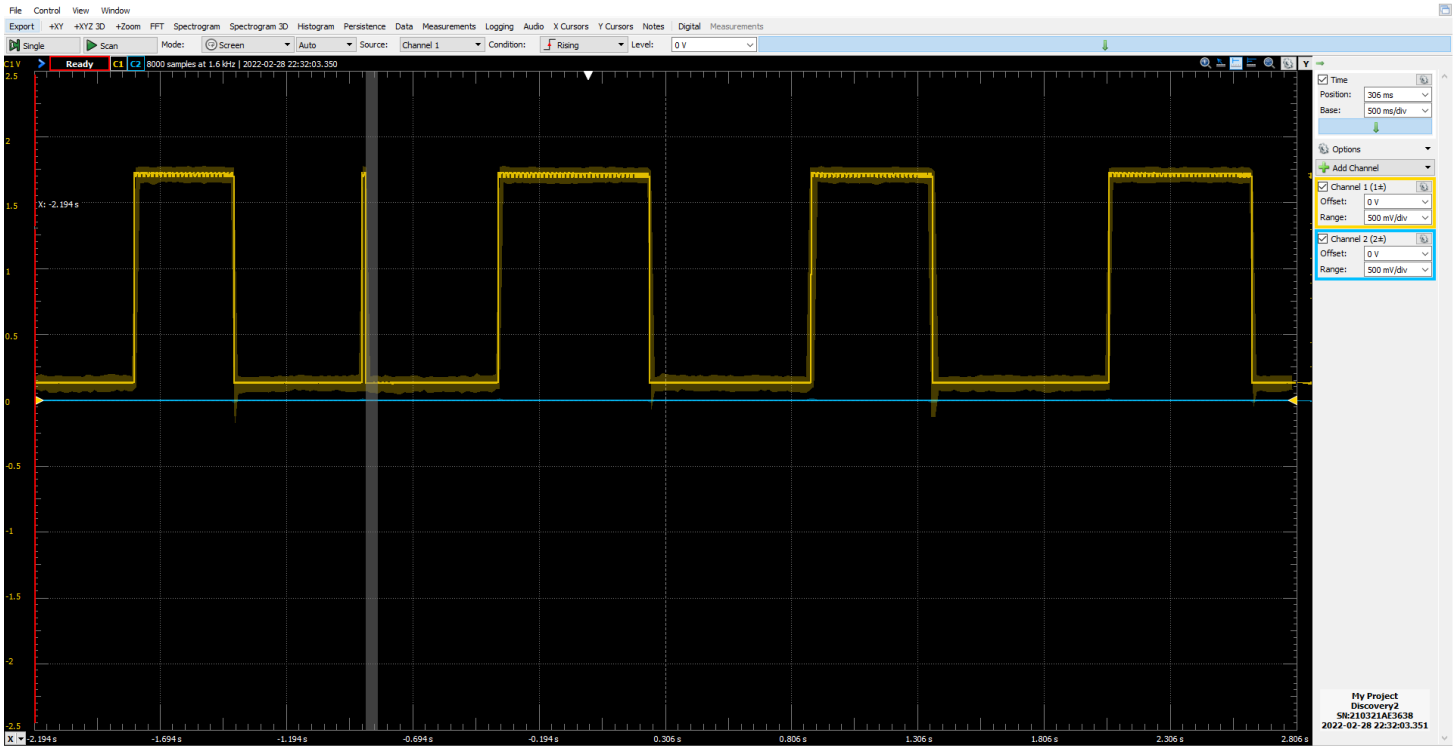

Fig. 9: Analog Discovery 2 Scope Output

Seen in Figure 9, the behavior is observed when the capacitive touch input is ‘pressed’. At idle (no touch), the output of the touch IC hovers at around 1.6V to 1.7V while when pressed, it drops down to around 150mV. At each point in Figure 9 where this drop from 1.6-1.7VDC to ~150mV occurs (known as the falling edge), the program we wrote earlier detects this fall to know that we have touched an input and output which pin was touched to the command line. This behavior repeats itself for all of the input/output pins.

The touch sensor works by detecting a *capacitive load*. Touching a pin on the input with a finger is an example of this. Essentially, when we touch the pin, a tiny amount of charge serves as a capacitor. This change in capacitance is picked up by the sensor which completes the circuit and in turn lights the LED on the breakout board and causes the output pin to go low. It is like pressing a button without any sort of physical feedback. Most mobile phones today use capacitive touch screens which is why we are able to use our fingers to swipe and tap to navigate around.