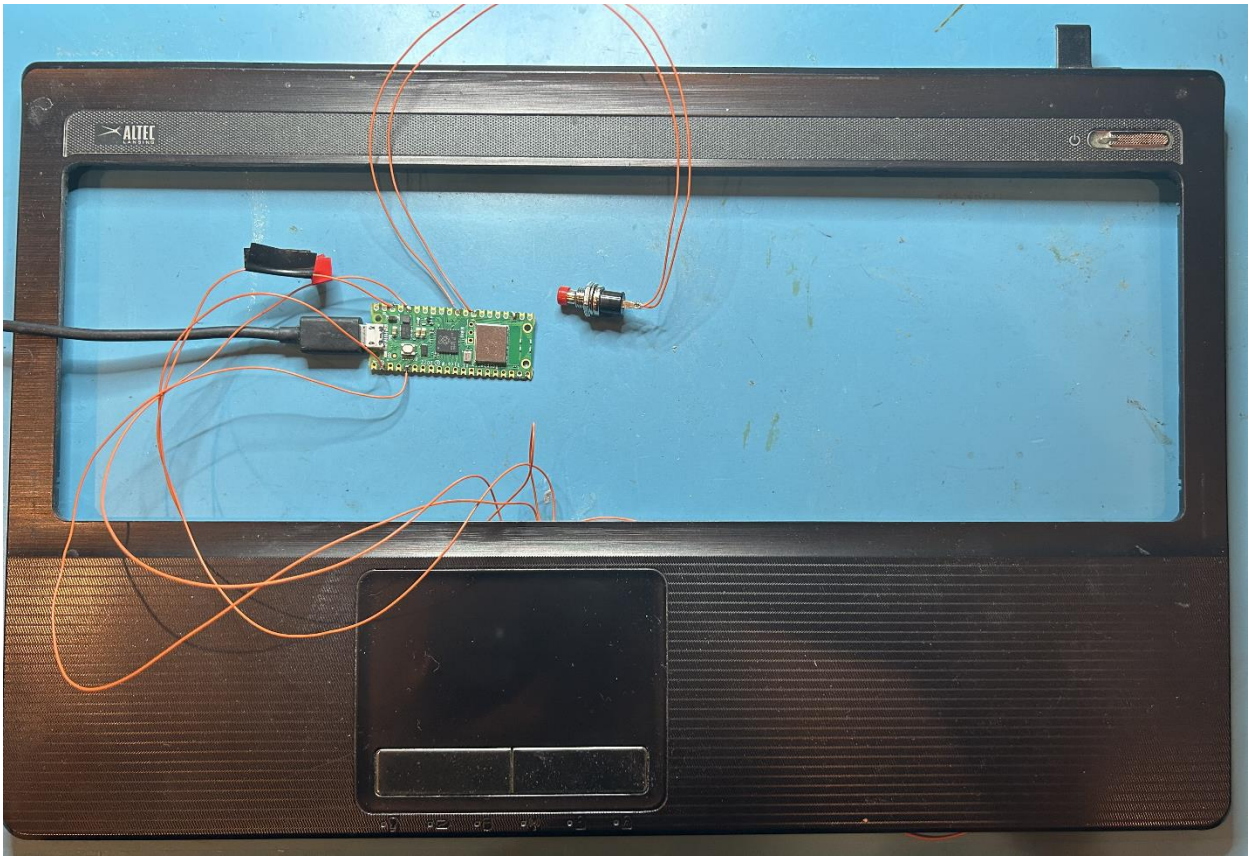
Raspberry Pi Pico Converts Laptop Touchpad PS/2 bus to USB



Use the information given at my Hackster tutorial to determine the power, ground, clock, and data pins on your touchpad (TP). This Synaptics TP has numbered test points that were connected to the Pico pins as shown.

Required Connections
T12 Clock to Pico Pin 5 GP3
T13 Data to Pico Pin 2 GP1
T22 3.3 Volts to Pico Pin 36 3V3
T23 Ground to Pico Pin 38 GND

This TP is powered with 3.3 volts but if a 5 volt TP is used, a level translator is needed to avoid damaging the Pico. This touchpad (like most) has active pullup resistors on clock and data so no additional resistors are needed although I did initially add 3K pull ups during code debug. With 3K resistors the rise time was 200ns and without the resistors, it was 600ns.

The code “Raspberry_Pi_Pico_TP.ino” at my repo should be programmed into the Pico using the Arduino IDE. After programming, you must cycle the Pico power so it starts from a power-up state. This is due to the use of the watchdog timer explained below and in this link.

Arduino C code running on the Pico is a lot slower that similar code running on a Teensy. The bit-bang PS/2 code is barely fast enough to keep up with the TP. Small microsecond delays in the code are not possible and larger microsecond delays are not accurate. I used an oscilloscope to adjust the code to get the desired timing. Future compilers will probably get faster and the delays may need to be adjusted accordingly. There seems to be a big delay when changing a Pico GPIO pin from output to input. This is done to act like an open drain driver which for most Arduino/Teensy microcontrollers, is not available in hardware. I’ve read the Pico has the hardware capabilities to disable an output driver which might be faster than switching it to be an input. Once I learn how to control the output enable in the Arduino code, there may be speed improvements.

While loops are used to watch for the PS/2 clock edges driven by the TP. I tried adding additional code statements in the while loop in order to break out if hung, but this makes the code fail because it’s too slow. My solution is to use the hardware watchdog timer in the Pico to issue a reset if the code hangs. Once the wiring is correct, the Pico doesn’t hang so this is only a problem when figuring out the clock and data connections. As a further aid in debugging the connections, an error signal is driven high if the clock signal is incorrect. This error signal is on Pico pin 19 GP14. Monitor this pin with a meter while trying to get the TP wired correctly.

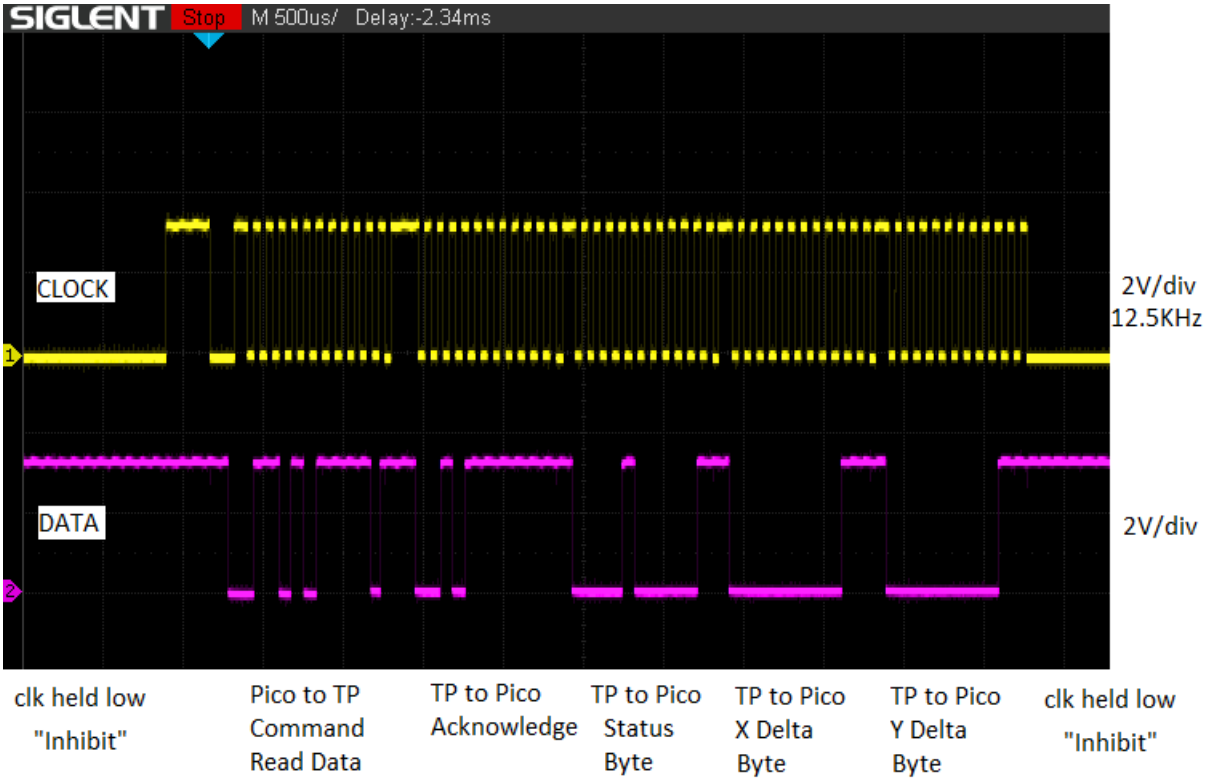
There are lots of places where error handlers could be added to the code. These include bad Ack bit, bad Ack byte, bad TP self-test, bad parity, bad stop bit, etc. Other than resetting the TP, I’m not sure what else could be done.

The code initializes the TP for remote mode (instead of stream mode) so it can be polled at a regular rate of 25msec which fits in nicely with a keyboard scanning routine. I used USBMouseKeyboard.h instead of USBMouse.h for this reason.

Refer to the Synaptics TouchPad Interfacing Guide 510-000080 – A for information on the following setup parameters that are sent to the TP during initialization:

- Software Reset - 0xff. After 1 second, the TP self-test results are read back.
- TP Disable - 0xf5. The TP is disabled to make sure it receives the following commands.
- Load Mode Byte - 0x40. This requires a special sequence given in paragraph 3.5.2 of the spec.
- Set Resolution - (0xe8) and (0x03). 8 counts/mm resolution (default is 4 counts/mm).
- Set Sample Rate – (0xf3) and (0x28). 40 samples/sec. The default value used by Synaptics.
- Remote mode - (0xf0). The TP will only send data when polled.
- TP Enable - (0xf4). The TP is enabled.

The Pico holds the clock line low to inhibit the bus. Every 25msec, the Pico releases the clock and it's pulled high with a resistor. After 250usec, the Pico sends the clock low and waits at least 100us before driving the data line low to initiate a request to send. These are the two delay values that I fine-tuned in the code. The Pico releases the clock because it's now under control by the TP. Below is a scope capture showing the Pico to TP read data command (0xeb) followed by the TP Acknowledge byte (0xfa). The status byte, x-delta byte, and y-delta byte follow as described below. Note the TP is running the clock at 12.5KHz.



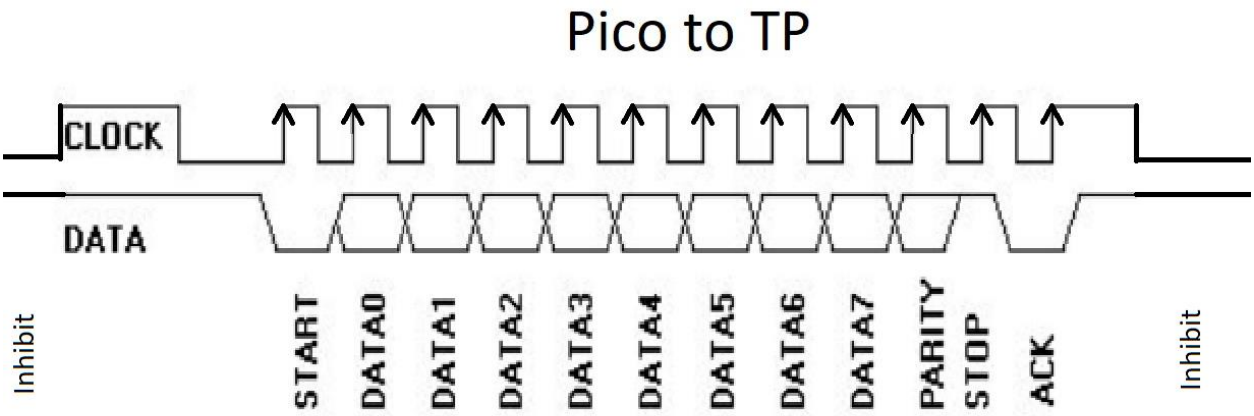
The bit definitions of the status byte are shown below.

Bit 7	*Bit 6*	*Bit 5*	*Bit 4*	*Bit 3*	*Bit 2*	*Bit 1*	*Bit 0*
Y Overflow	X Overflow	Y Sign	X Sign	Allways 1	Middle Button	Right Button	Left Button

- The overflow bits are set if the X or Y delta values exceed their 9 bit 2's complement size (-256 to +255). Even the fastest/largest movement on the touchpad doesn't cause an overflow if the touchpad data is read at least every 50msec.
- The Sign bits are the 9th bit that go with the X and Y delta bytes that are received after the status byte.
- The button bits are set when the corresponding TP button is pushed. Code could be added if your TP has a middle button.

The Arduino Pico function, MouseKeyboard.move needs 16 bit 2's complement values for the X and Y delta's. The code converts 9 bit 2's complement to 16 bit by copying the sign bit into the upper byte.

This shows the Clock and Data bits when the Pico sends a command to the TP. Note the data is clocked into the TP on the rising edge.



This shows the Clock and Data bits when the TP sends data to the Pico. Note the data is clocked into the Pico on the falling edge.