Goals

1. Make a hand-held probe that can verify whether two signatures match
2. I envision being able to swipe the probe over two signatures to verify whether the signature is correct
3. This probe will grab signatures using the camera/motion sensor provided by the ADNS-9800 laser mouse chip
4. The probe will also be able to provide a "forge" mode that allows the user to forge a pre-recorded signature by following directions on a set of LEDs or similar

Introduction

I intend to build a portable, hand-held probe that allows you to scan over two signatures and verify whether or not they match. This can be used by, for example, lawyers and mailmen who wish to quickly and systematically verify that the same person signed two documents. The probe should be portable (self-contained, small, and lightweight) and easy to use (simply swipe over the two signatures to verify authenticity).

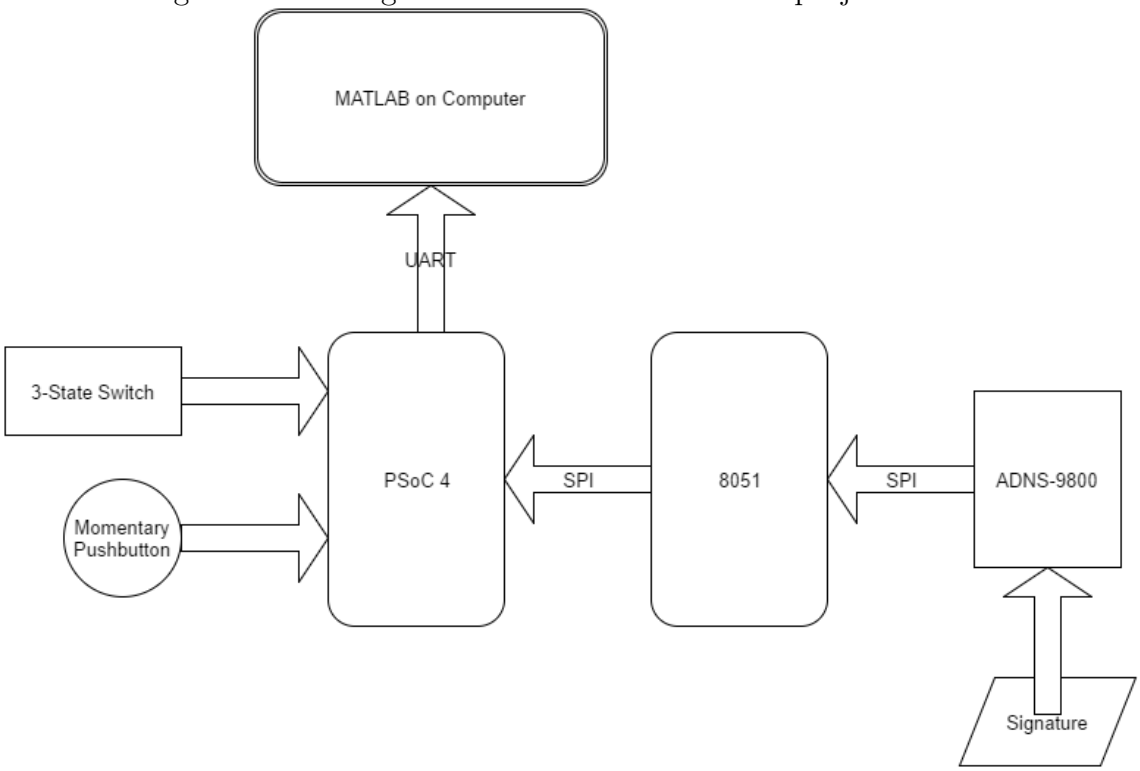
I am interested in this idea primarily because it is unique and because of the technology it integrates together. As a course 6-2, I not only like figuring out how hardware components fit together, but also how I can solve problems using innovative algorithmic techniques. Here, I get to learn about signature matching using artificial neural networks and different image vision techniques that can identify particular distinctive features in a signature. Ive also recently been interested in optical sensors (which provide both images and motion data) because they seem so widely applicable to anything that moves, so I wanted to learn how to use one with a microcontroller.

Finally, I believe this is a creative solution to the problem of not being able to verify the authenticity of signatures because it is easy-to-use and unintrusive in the daily work of people who might want to verify signatures. Signatures are used universally and on some very important documents. However, just based off of anecdotal evidence, the appropriate time and care is not being taken to verify the authenticity of signatures – often, in restaurants for example, even widely different signatures are taken to be from the same person without much thought. Given the popularity of optical character recognition (OCR) in solving other similar problems, such as automatically processing

mailing addresses, I believe that such a combined hardware and software approach to this signature verification problem will be very effective.

Hardware Overview

The following is a block diagram of the hardware for this project:



The ADNS-9800 chip captures the signature (using either motion detection or image capture, as described in the next section). This is communicated over SPI to the 8051, which will filter the signals from the 8051 and do some basic signal processing on them. It will then report only the relevant data to the PSoC. The PSoC also gets signals from a 3-state switch and a momentary pushbutton. The momentary pushbutton is to be held down when capturing a signature, and the position of the 3-state switch will determine what to do with the captured signature. The 3 modes available are “Record”, “Verify”, and “Forge”. The PSoC takes all this information and communicates it to a MATLAB program running on the computer. This MATLAB program is responsible for assimilating information from multiple signatures to either verify them or help you forge them.

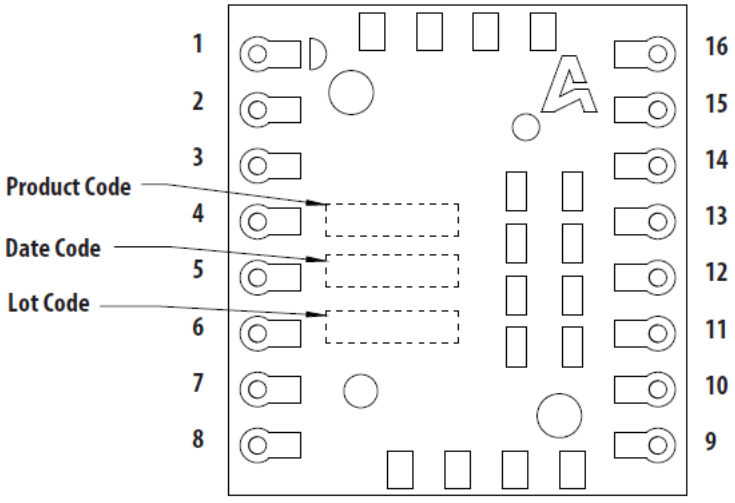
ADNS-9800 Sensor

The ADNS-9800 sensor is a laser mouse sensor that contains a very small camera to track motion over a surface. It works by taking pictures of the surface at a very high frame rate and calculating velocity by running optical flow. I picked this particular chip for its small size, good documentation, and ability to be used as a regular camera. It contains a number of registers, and the SPI protocol is used to read from and write to specific registers. There are three steps in getting the ADNS-9800 sensor working:

1. Wiring up the chip
2. Implementing the SPI protocol
3. Communicating with the chip's registers

Wiring up the chip

The following is a diagram and pin labeling of the ADNS-9800 chip:



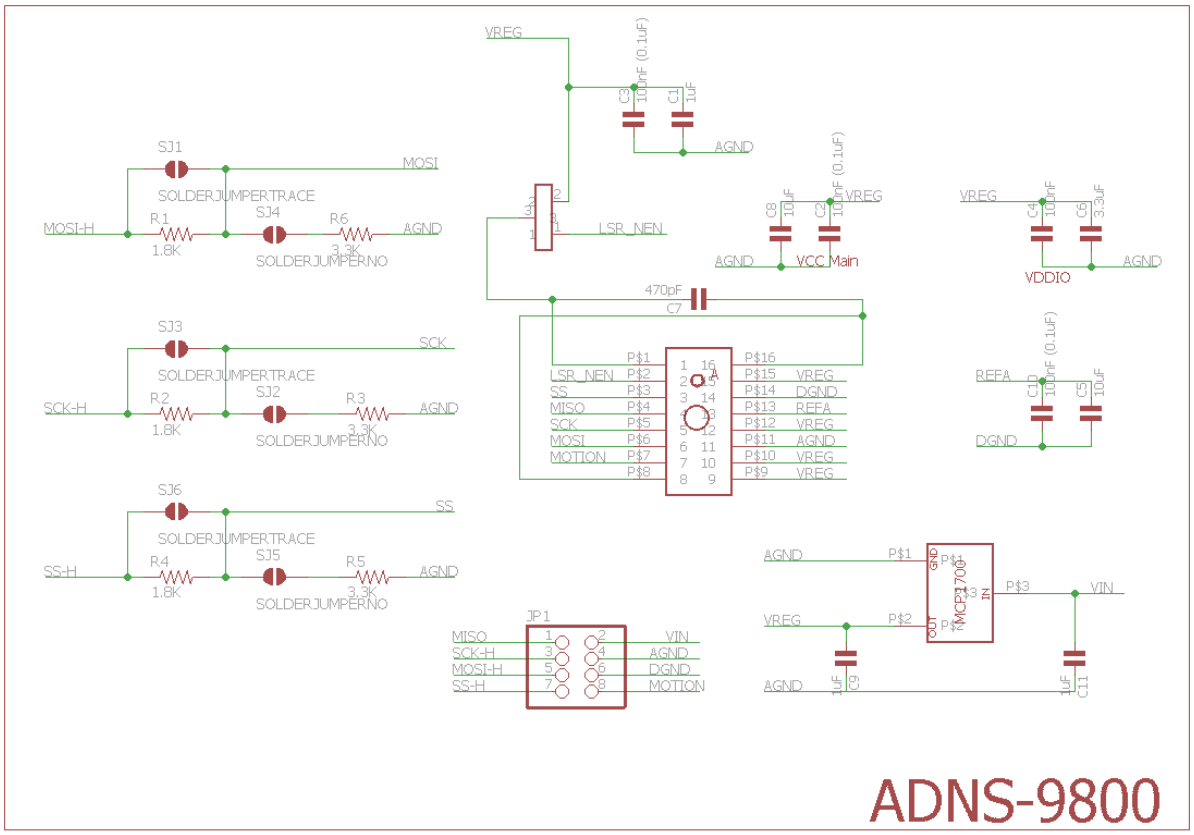
Pin No	Pin Name for 5 V mode	Pin Name for 3 V mode	Description
1	+VCSEL	+VCSEL	Positive Terminal Of VCSEL
2	LASER_NEN	LASER_NEN	LASER Enable (Active Low Output)
3	NCS	NCS	Chip Select (Active Low Input)
4	MISO	MISO	Serial Data Output (Master In/Slave Out)
5	SCLK	SCLK	Serial Clock Input
6	MOSI	MOSI	Serial Data Input (Master Out/Slave In)
7	MOTION	MOTION	Motion Detect (Active Low Output)
8	XYLASER	XYLASER	Laser Current Output Control
9	VDD5	VDD3	5 V input for 5 V mode 3 V Input for 3 V mode
10	PWR_OPT (GND)	PWR_OPT (VDD3)	Power Option: Connect to GND for 5 V Mode Connect to VDD3 for 3 V Mode
11	GND	GND	Analog Ground
12	REFB	VDD3	3 V Regulator Output for 5 V Mode 3 V Input for 3 V Mode
13	REFA	REFA	1.8 V Regulator Output
14	DGND	DGND	Digital Ground
15	VDDIO	VDDIO	IO Voltage Input (1.65 – 3.3 V)
16	-VCSEL	-VCSEL	Negative Terminal Of VCSEL

The ADNS-9800 can run in both 5V and 3V mode. In 5V mode, you provide 5V to VDD5 and GND to PWR_OPT, and an internal regulator in the ADNS-9800 chip will provide 3V to provide voltage signals to the other pins. In 3V mode, you provide 3V to VDD3 and 3V to PWR_OPT – you are responsible for providing the 3V yourself.

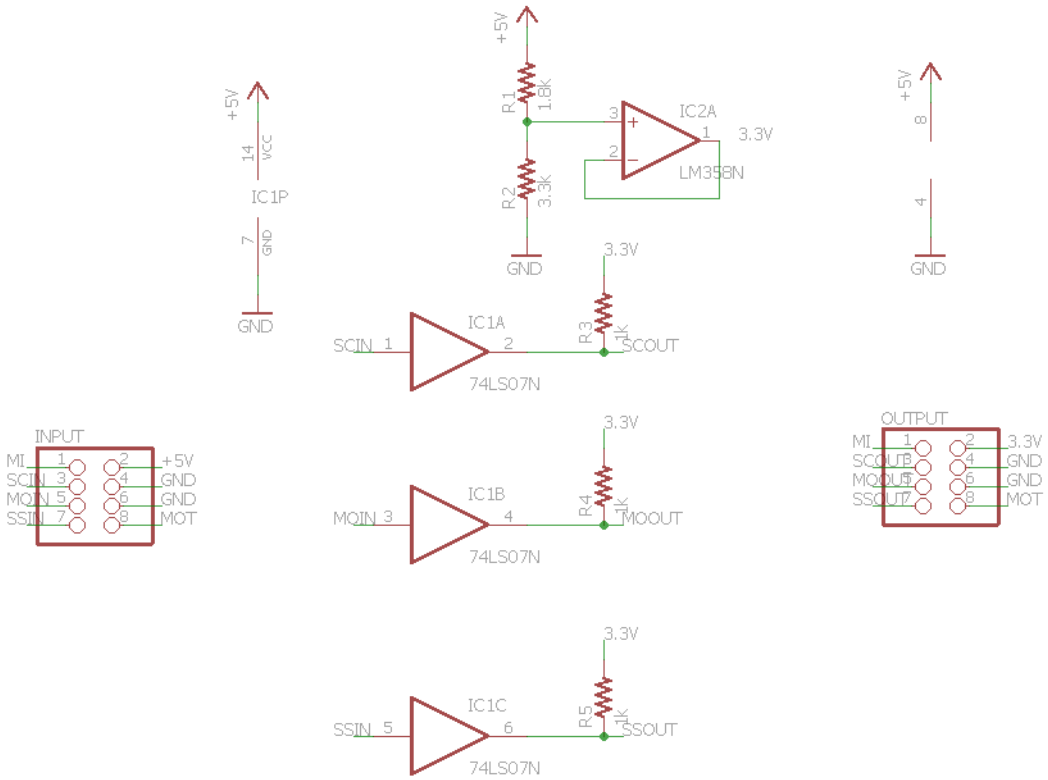
Pins 2-6 (NCS, MISO, SCLK, and MOSI) are used in SPI communication and can be left floating when just trying to get the chip wired up properly. Pin 7 (MOTION) is another pin used for communicating with the microcontroller – it goes high whenever the chip detects motion. In my informal testing when I got the chip wired up, MOTION would go high whenever I placed my finger on the lens. GND and DGND are both wired to ground, and in 3 V mode (which is what I would be using, and the reason will be explained later), VDD3 is wired to 3 V and PWR_OPT and REFB are wired to VDD3. REFA (pin 13) is a 1.8 V regulator output and can be ignored.

This leaves the following pins: +VCSEL (pin 1), LASER_NEN (pin 2), XYLASER (pin 8), VDDIO (pin 15), and -VCSEL (pin 16). LASER_NEN and XYLASER relate to the operation of the laser, which has to be on in order for the mouse to illuminate the surface and capture images. A microcontroller communicating via SPI sets the appropriate register value to enable or disable the laser. This will set LASER_NEN to low, which will activate a FET. This will allow XYLASER to draw power from the power supply and activate the laser. The other pins also serve different purposes and their wiring is described in more detail in the datasheet for the ADNS-9800.

The following is a schematic for the breakout board that I used for the ADNS-9800 chip:



However, the issue here is that it cannot be used in 5 V mode because the voltage divider at MOSI-H, SCK-H, and SS-H do not actually divide unbuffered voltage. The pin of the 8051 contains an internal pull-up resistor which makes this voltage divider not divide voltage properly. To get around this I ran this breakout board in its native 3V and made a 5-to-3V level converter myself. The following is a schematic for the 5-to-3 TTL converter:



This contains an op-amp for converting 5V to 3V, and 3 buffers from the 74LS07 open-collector-output chip to perform the level conversion. This setup got the chip working at the right TTL level.

Implementing the SPI protocol

The ADNS-9800 chip uses the SPI protocol with a max clock frequency of 2 MHz to communicate register values with a microcontroller. The data direction is MSB to LSB, an inactive SCLK signal is high, and data is read by the chip on rising clock edges. There are four types of commands:

1. Write operations: 16 bits are sent over MOSI. The first byte indicates the address of the register to write, and the second byte indicates the data to write to register. The first bit in the signal is 1 to indicate that it is a write command.

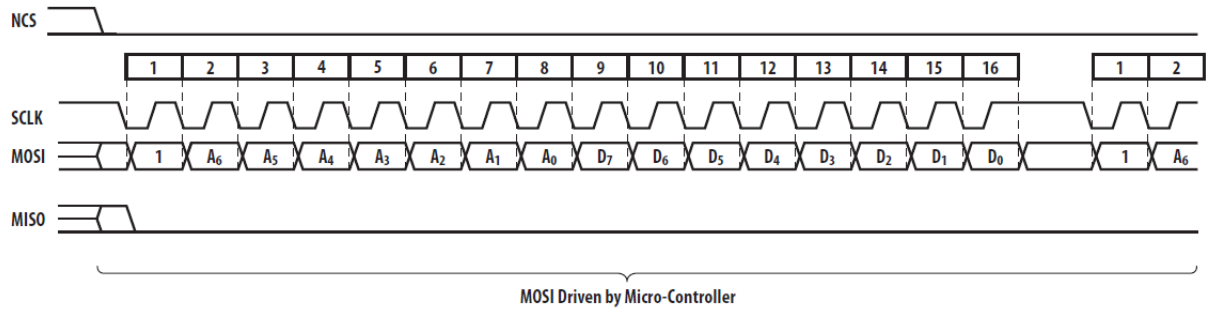
2. Read operations: 8 bits are sent over MOSI and 8 bits are received over MISO. The byte sent over MOSI indicates the address of the register to read (with the first bit a 0 to indicate reading), and the byte received over MISO is the data contained in the register. There is a also a required timing between sending the first byte and receiving the second.

3. Motion burst operations: In some cases reading each register corresponding to different kinds of motion individually using the read and write operations described above can be too slow. In motion burst operation, the MOTION_BURST register is written to once. Then, when it is read using a normal read operation, 14 bytes are returned instead of 1, corresponding to the values of 14 different motion-related registers instead of just 1 at a time.

4. Image burst operations: Image burst mode allows you to download an entire frame that the chip has most recently captured. Since each frame is 30-by-30, this means transferring 900 8-bit pixel values over SPI. In this operation, you must first write to the FRAME_CAPTURE register twice, read the MOTION bit register until the LSB is set, and then read 900 bytes from the PIXEL_BURST register. This mode, however, disables motion tracking and requires a hardware reset. Therefore, switching back and forth between motion burst mode and image burst mode is not entirely practical.

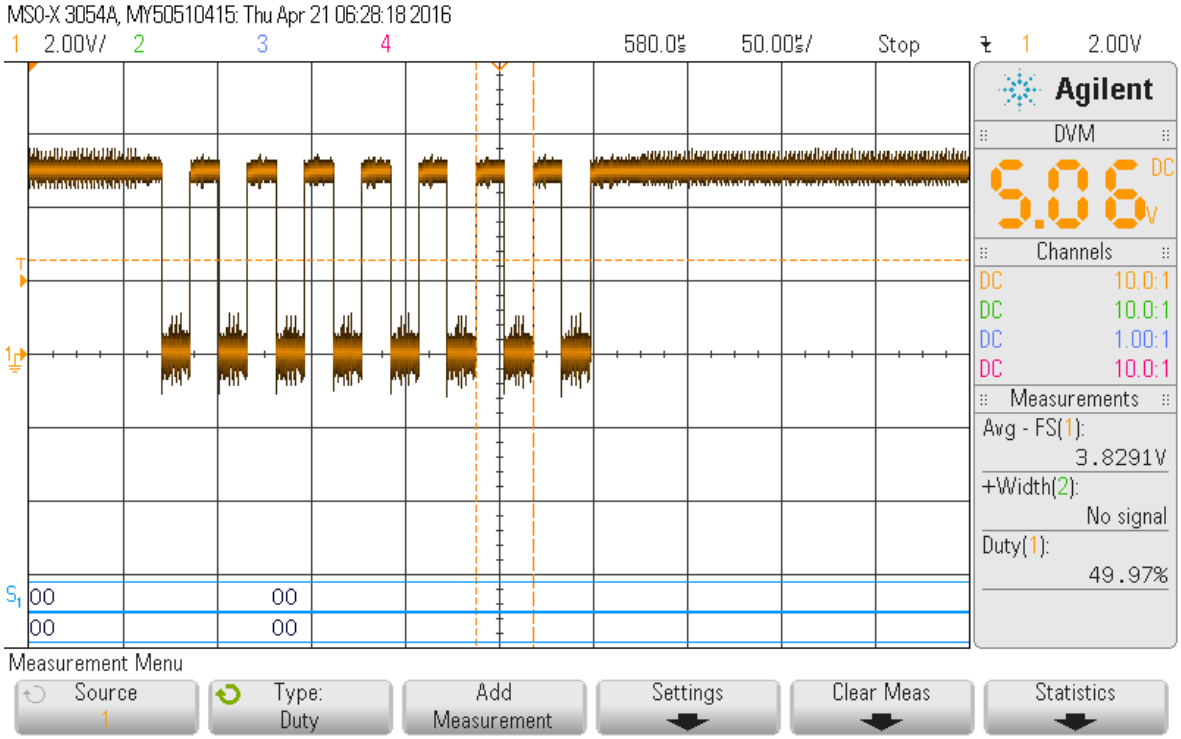
Write operations

The following is a timing diagram for this operation:

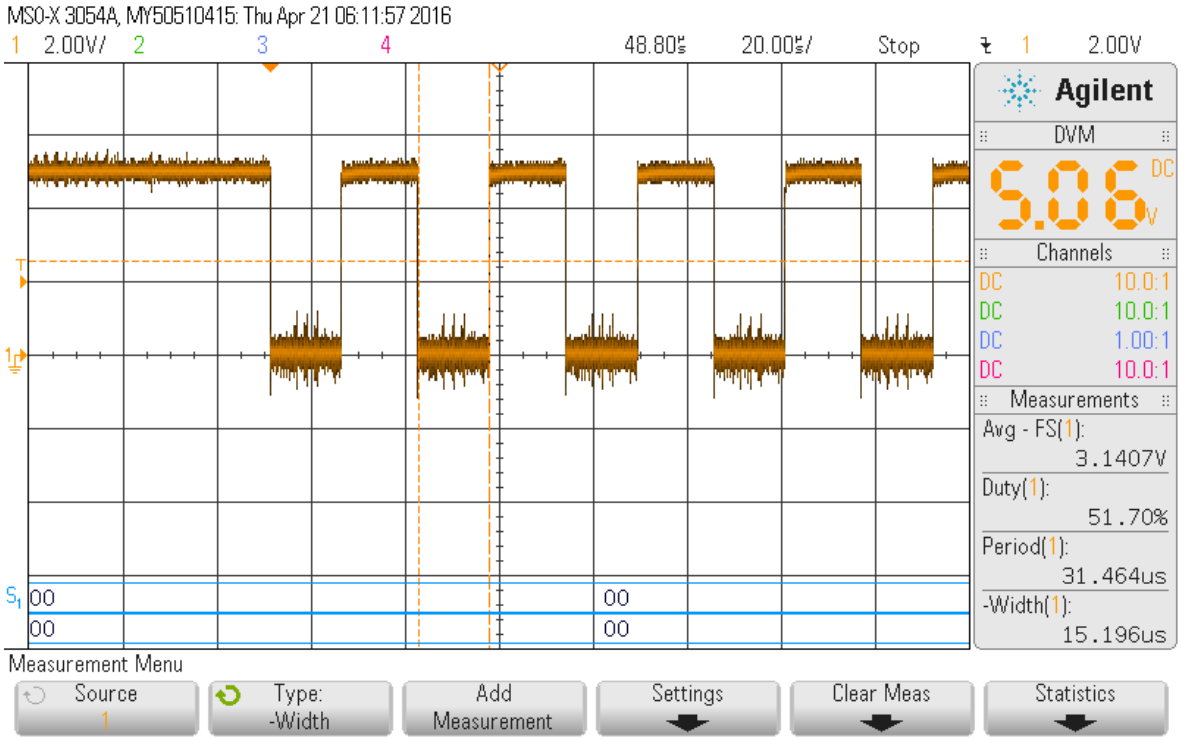


I implemented this by first writing the `read_spi` and `write_spi` subroutines on my microcontroller. These two subroutines drive an SCLK signal and either write a byte over MOSI or read a byte over MISO. They do not drive the SS pin low; that must be done in a higher level subroutine that takes into account which slave to select.

The following is an SCLK waveform I got on `read_spi`:



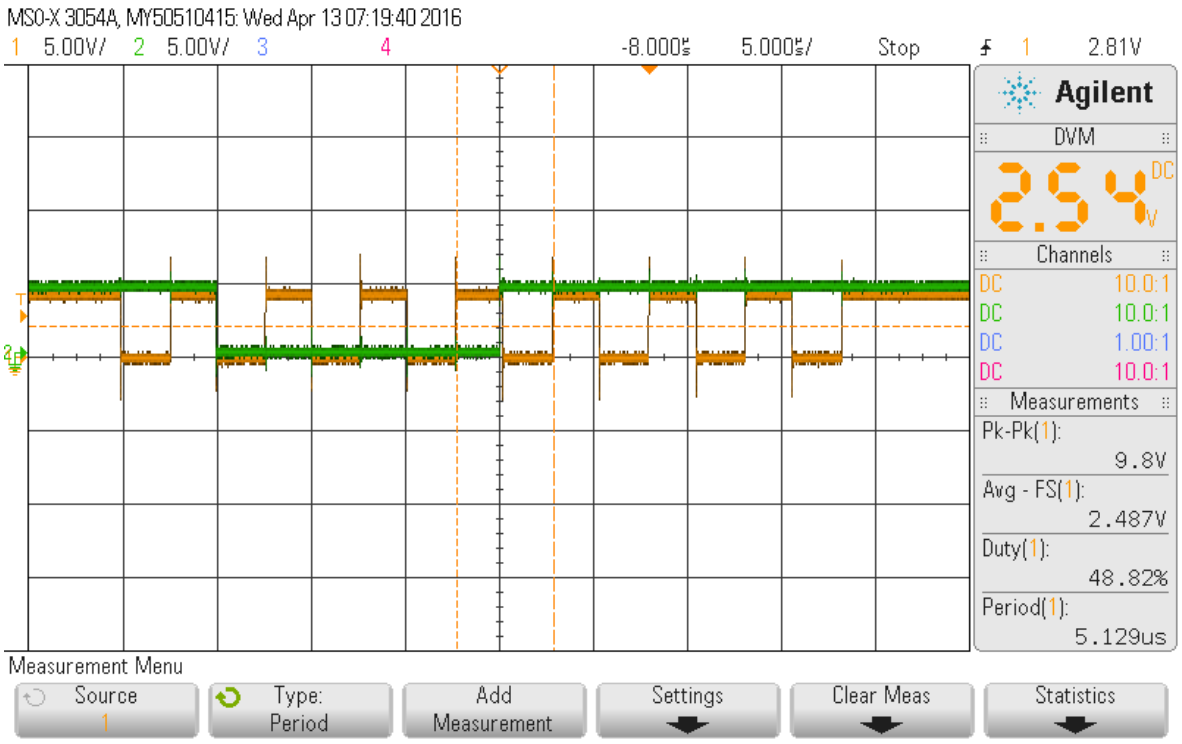
And the following is an SCLK waveform I got on `write_spi`:



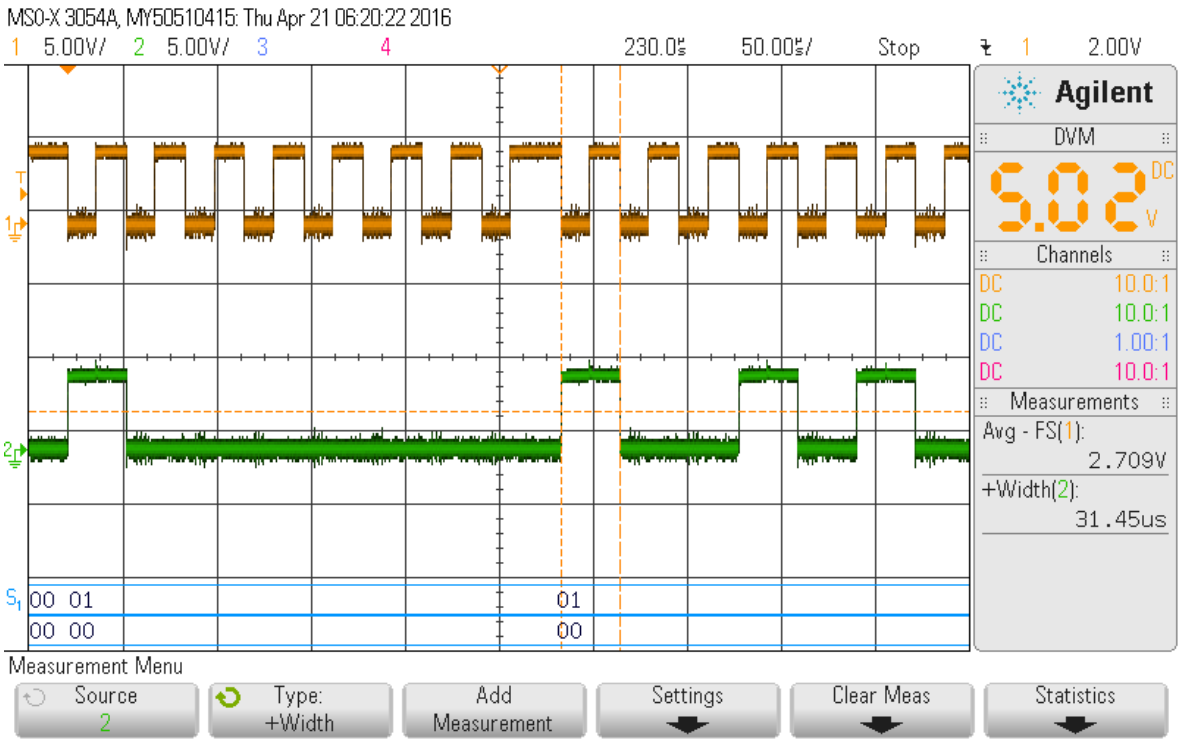
As you can see, the duty cycle is about 50% and the period is on the order of 30 us. These waveforms were not expected to be exactly the same because different opcodes and therefore different timings are used in `write_spi` and `read_spi`. However, they

do fit the requirements of the ADNS-9800 chip. I later modified these to have a smaller period by giving up on the duty cycle having to be 50% with the same result in terms of SPI operations but a slightly faster transfer rate.

Next, I implemented the `write_adns` subroutine, which performs a write operation on the ADNS-9800 chip. It lowers NCS, does a `write_spi` to write the register address over MOSI (with the MSB set to 1), then another `write_spi` to write the register value over MOSI, and then finally raises NCS. I got the following waveforms for the SCLK and MOSI lines:



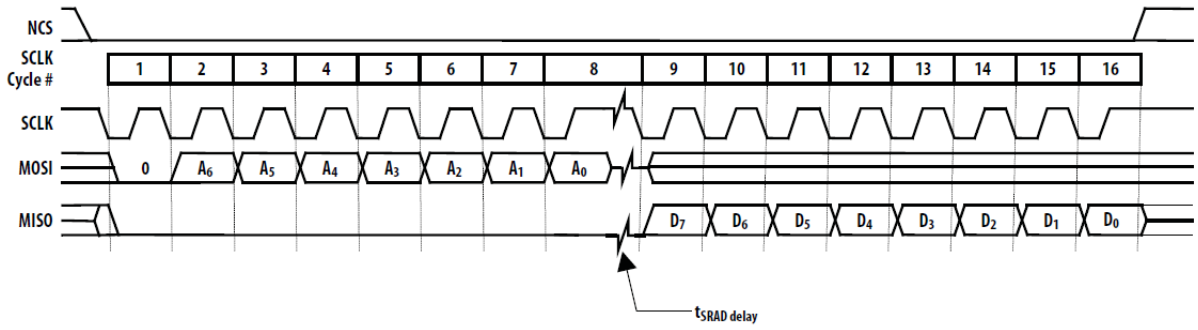
And I got the following writing the value `#0F5h` to register `#00h`:



I did not have a way to test this, however, without implementing a read operation as well.

Read operations

The following is a timing diagram for this operation:



To implement a read operation, I wrote a subroutine called `read_adns`, which lowers NCS, writes the register address to MOSI with the MSB set to 0 using `write_spi`, waits for a bit, and then reads the register value over MISO using `read_spi`. The following are the waveforms for SCLK and MISO during a read operation:

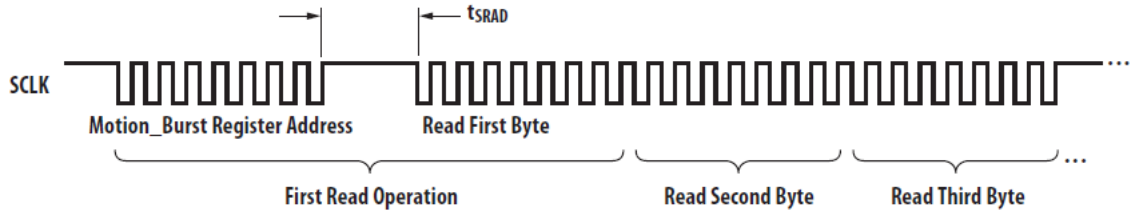
These two diagrams are reads of registers `#01h` and `#00h`, respectively, which the datasheet says should always have the values `#03h` and `#33h`, respectively. As you can see above, this is indeed confirmed by the scope.

To test the write operation, I wrote to a register and tested whether read returned that

same value. It did, so I was confident that the write operation worked.

Motion burst operations

The following is a timing diagram for this operation:



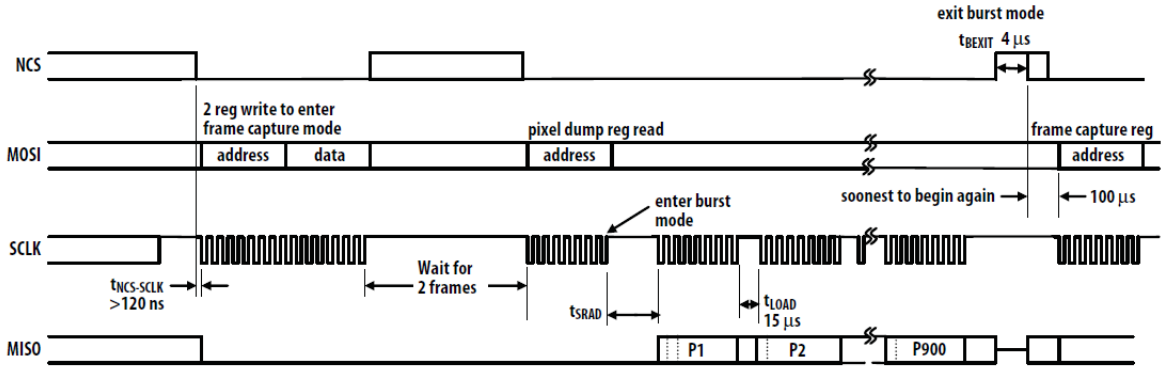
The way I implemented motion burst mode was to:

1. Designate a 14-byte portion of the RAM where the data from the 14 registers read will be stored consecutively
2. Write a subroutine called burst that continuously calls `read_spi` and writes bytes consecutively to a portion of RAM until a specified position in memory is reached
3. Write a subroutine called `motion_burst` that takes care of writing to the MOTION_BURST register, lowering NCS, etc.

Running motion burst mode requires the laser to be turned on, which will be described in a later section.

Image burst operations

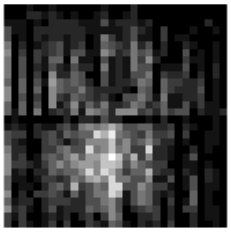
The following is a timing diagram for this operation:

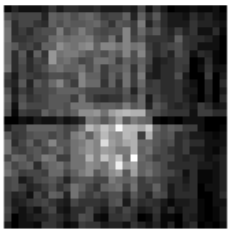


In this mode, 900 pixels are to be transferred. I again reused the `burst` subroutine I wrote for motion burst mode. I designated a 900-byte part of RAM to store a frame and wrote a subroutine called `image_burst` to read and write to the appropriate registers (according to the timing diagram above) and call the `burst` subroutine to store all the values read over SPI to RAM.

Running image burst mode also requires the laser to be turned on as well as a hardware reset, which will also be described in a later section.

To test that this code works, I printed out the 900 pixel values over the serial port and wrote a MATLAB script called `viewImage.m` that takes in 900 pixel values and shows you an image reconstructing it. At first, I saw only very dark frames because the spacing between the lens and surface was not correct. The following is an example of such an image after constructing my mechanicals to have the right lens height and after modifying the visualization algorithm to normalize pixel brightness, taken over a 5 dollar bill and over my driver's license, respectively:





As you can see, what is in the image is not very clear, the resolution is low, the result is noisy, and the physical window size is very small (only about a square millimeter). For these reasons, I decided to stick with recording pen motion instead of capturing the signature using a camera.

Testing

To test whether my communication with the chip worked, I made a repl that supported the following operations:

1. `R`: Read a register and print out its value
2. `W`: Write to a register
3. `M`: Perform a motion burst capture and print out the values of the 14 registers that we get
4. `I`: Perform an image capture and print out the values of all 900 pixels that we get

5. `P`: Perform a power-up sequence (reset the hardware and enable the laser)

This helped me with my next task, which was to figure out what different operations did and test them out before I coded them.

Communicating with the chip's registers

The next step in getting the ADNS-9800 working was to figure out what the different registers mean and how to perform operations such as power-up and laser enable. The following is a table of the registers relevant to my project and their description:

Register Name	Address	Description	Remarks
Motion	02h	Contains information about whether motion or image capture data is available	Motion data is available if MSB is set to 1, an image frame is available if LSB is set to 1. Reading this register freezes Delta_* values.
Delta_X_L	03h	Lower byte of accumulated motion in X direction since the last read	Reading it clears the MSB of MOTION and resets the value of this register
Delta_X_H	04h	Upper byte of accumulated motion in X direction since the last read	Reading it clears the MSB of MOTION and resets the value of this register
Delta_Y_L	05h	Lower byte of accumulated motion in Y direction since the last read	Reading it clears the MSB of MOTION and resets the value of this register
Delta_Y_H	06h	Upper byte of accumulated motion in Y direction since the last read	Reading it clears the MSB of MOTION and resets the value of this register
Configuration_I	0fh	Resolution of the ADNS sensor in terms of how many counts are reported per inch travelled	Resolution of the sensor in CPI is the value of this register times 200
Frame_Capture	12h	Used to capture the next complete frame of 900 pixels during image burst mode	Writing to this register starts image burst
Laser_CTRL0	20h	Used to turn the laser on and off	Setting the LSB to 1 disables the laser and setting it to 0 enables it
Power_Up_Reset	3ah	Used to reset the chip (set all registers to their default values)	Write 5ah to this register to start up or to exit image burst mode
Motion_Burst	50h	Used to capture the values of 14 motion-related registers at once	Reading this register grabs all 14 values at once
Pixel_Burst	64h	Used to capture the next complete frame of 900 pixels during image burst mode	Reading this register grabs all 900 pixel values at once

Mechanicals

The next step was to construct the mechanicals with the following goals in mind:

1. Optimize the distance between the sensor and the reading surface as to allow maximum motion pickup
2. Attach to a pen-holding device that allows you to write while tracking motion
3. Have an easily-holdable button on the side that you can press and hold to record the signature
4. Have a 3-state switch that allows you to flip between RECORD, VERIFY, and FORGE modes

I did my designs in Solidworks and 3-D printed the parts.

8051

On the 8051 I wrote code that would communicate via SPI with both the ADNS-9800 and the PSoC. In this system, it is the master to the ADNS-9800 and the master to the PSoC. It communicates with both by flipping the right slave-select line.
Furthermore, I had some basic signal processing on my 8051. In its main loop, the 8051:

1. Polls the ADNS-9800 in motion burst mode
2. Updates a displacement counter
3. If the manhattan distance for the displacement exceeds a certain quantity, it clears the displacement counter and reports a hit to the PSoC

In this way, the 8051 acts as a software-based 2-dimensional Schmitt trigger. We can imagine that the reading surface is broken down into a grid of diamonds (diamonds are the "circles" in Manhattan space). By using the above algorithm, we can clearly see that the threshold for traveling to the next grid cell is different from the threshold for moving back. This acts as to "debounce" the optical sensor motion.

PSoC

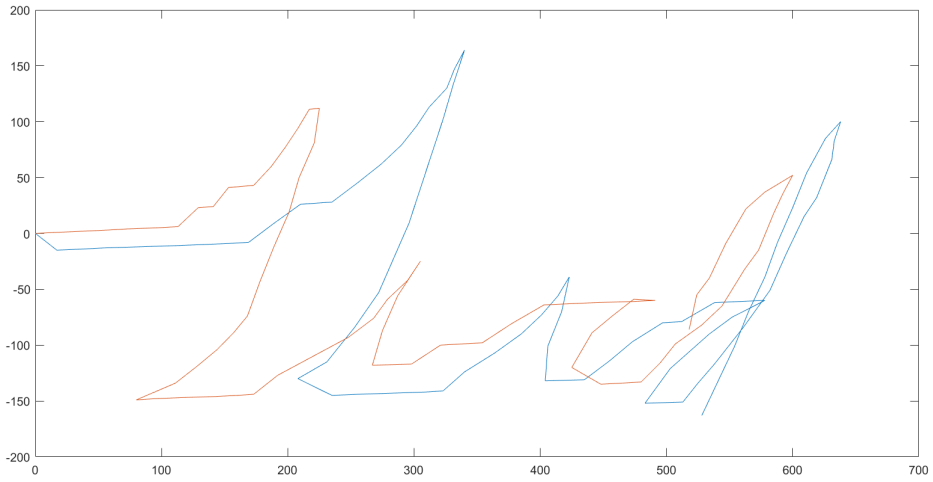
On the PSoC, I have an SPI slave unit, a UART unit, and multiple pins to receive signals from the button and switch. This reports data to MATLAB over UART.

MATLAB

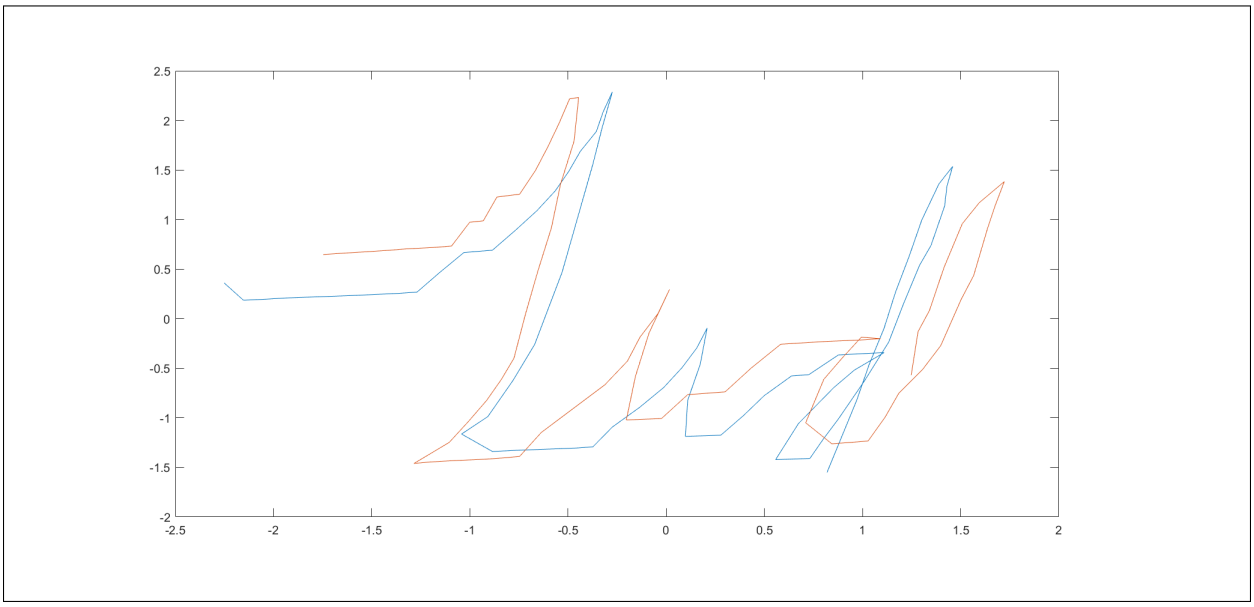
I tried two different approaches to verifying the signature:

1. Center and normalize the signature by comparing mean and standard deviation. Compute average sum-of-square displacement of points.
2. Center and normalize signature by comparing mean and standard deviation. Compute angle displacements at each joint required to match the signatures together.

The first approach worked better. The following are signatures of my own name that I captured:



And the following is the signatures normalized:


2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2