

# CCD Camera Embedded Control System

CSC214 Final Project – Ken Carr

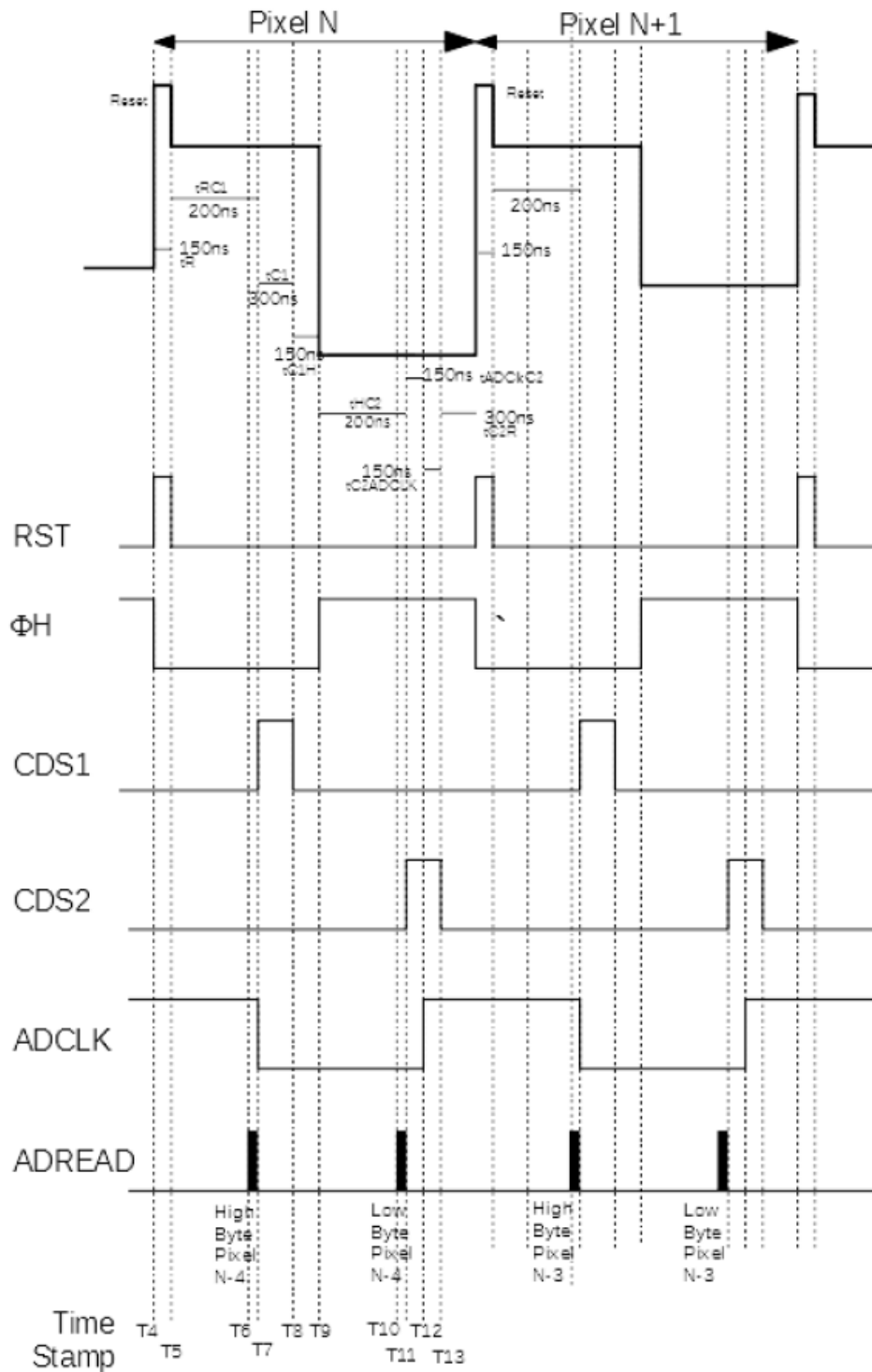
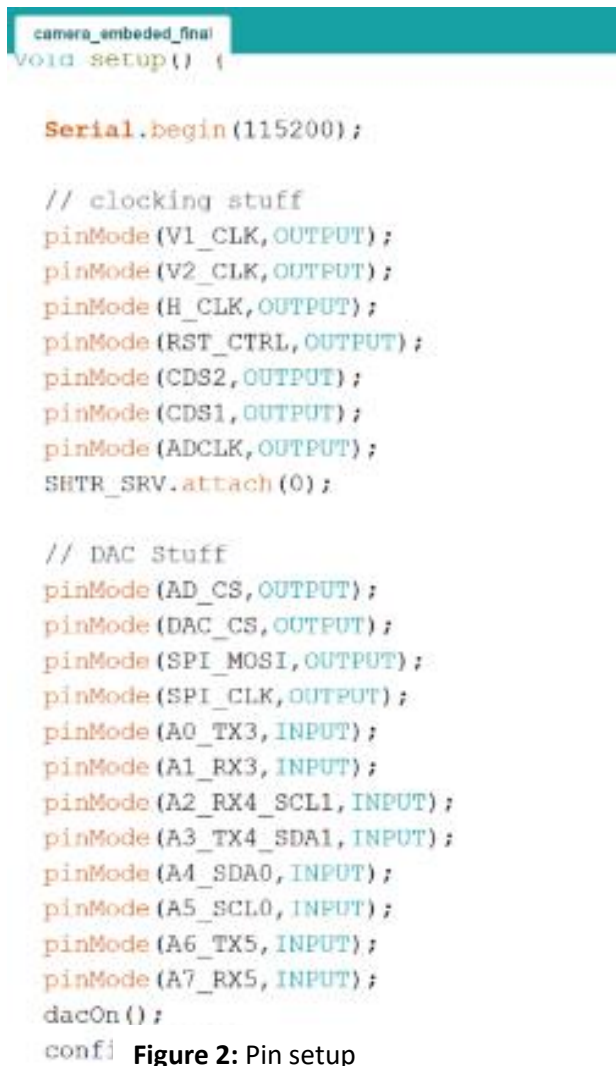


Figure 0: Main timing diagram.

## Key Software Highlights

The first part of the code above everything else are #defines shown in figure 1. It allows quicker clocking by accessing the GPIO directly. Digital Write is the easy Arduino way to do it but is slow and won't allow the response time needed for nanosecond timing. Each define sets up fast port bit manipulation for the specified GPIO pin. This happens before the setup and loop section of code in Arduino C. This method of GPIO control is used throughout the whole program, not just for the clocking signals.



```

camera_embedded_final
void setup() {

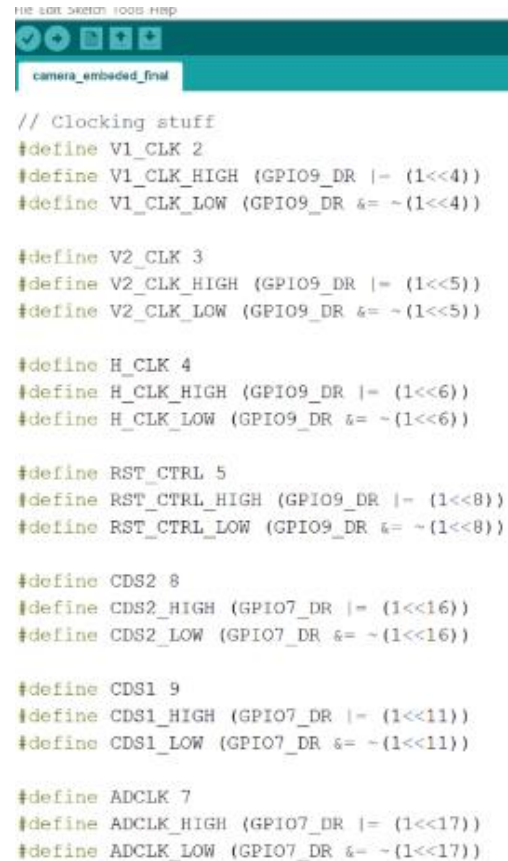
    Serial.begin(115200);

    // clocking stuff
    pinMode(V1_CLK, OUTPUT);
    pinMode(V2_CLK, OUTPUT);
    pinMode(H_CLK, OUTPUT);
    pinMode(RST_CTRL, OUTPUT);
    pinMode(CDS2, OUTPUT);
    pinMode(CDS1, OUTPUT);
    pinMode(ADCLK, OUTPUT);
    SHTR_SRV.attach(0);

    // DAC Stuff
    pinMode(AD_CS, OUTPUT);
    pinMode(DAC_CS, OUTPUT);
    pinMode(SPI_MOSI, OUTPUT);
    pinMode(SPI_CLK, OUTPUT);
    pinMode(A0_TX3, INPUT);
    pinMode(A1_RX3, INPUT);
    pinMode(A2_RX4_SCL1, INPUT);
    pinMode(A3_TX4_SDA1, INPUT);
    pinMode(A4_SDA0, INPUT);
    pinMode(A5_SCL0, INPUT);
    pinMode(A6_TX5, INPUT);
    pinMode(A7_RX5, INPUT);
    dacOn();
    confi

```

**Figure 2: Pin setup**



```

// Clocking stuff
#define V1_CLK 2
#define V1_CLK_HIGH (GPIO9_DR |= (1<<4))
#define V1_CLK_LOW (GPIO9_DR &= ~(1<<4))

#define V2_CLK 3
#define V2_CLK_HIGH (GPIO9_DR |= (1<<5))
#define V2_CLK_LOW (GPIO9_DR &= ~(1<<5))

#define H_CLK 4
#define H_CLK_HIGH (GPIO9_DR |= (1<<6))
#define H_CLK_LOW (GPIO9_DR &= ~(1<<6))

#define RST_CTRL 5
#define RST_CTRL_HIGH (GPIO9_DR |= (1<<8))
#define RST_CTRL_LOW (GPIO9_DR &= ~(1<<8))

#define CDS2 8
#define CDS2_HIGH (GPIO7_DR |= (1<<16))
#define CDS2_LOW (GPIO7_DR &= ~(1<<16))

#define CDS1 9
#define CDS1_HIGH (GPIO7_DR |= (1<<11))
#define CDS1_LOW (GPIO7_DR &= ~(1<<11))

#define ADCLK 7
#define ADCLK_HIGH (GPIO7_DR |= (1<<17))
#define ADCLK_LOW (GPIO7_DR &= ~(1<<17))

```

**Figure 1: Clocking Code defines**

The next part of code shown in figure 2 is the setup of the defined GPIO pins. They can be used as an input or output. For all the clocking function they will be outputs to generate square wave clock signals.

The DAC has 4 lines as output and 8 lines as input. The 2 output lines are programming channels called SPI (Serial Peripheral Interface). The operation of the pin is set here but the actually programming of the DAC happens in a function block that is called below.

Once all the pin variables and statuses are defined and the mode of the pin is set the actual code that will control the clocking program can be written. The first step of the embedded program is to take in what function needs to be executed. This is done by reading in a string that is a command. Each command is a line that contains the command as the first input, followed by values that are used to process the command. This is all part of the main loop.

Figure 3 shows the method that was used for tokenizing the string commands coming in from the host software. The software waits till a serial read is available and then takes in the string. The string is received as a character array and then the command is parsed out of the read line array by pulling characters till a space occurs. Values after that are parsed out as values for the math of the command function that is being performed. A string compare was done of the command to determine which one was used with a long list of if else statements. This function was also put in an infinite while loop within the Arduino run loop to ensure it can't terminate while it is running.



```

camera_embedded_final

void loop() {

  char c;
  char *tok;
  char ntok = 0;
  char *cmd;
  char *prm;

  while(1) {
    int ind=0;
    while(Serial.available() == 0); // waits for incoming command
    while ((c = Serial.read()) != '\n') {
      msg[ind++] = tolower(c);
      delay(1);
    }
    msg[ind] = '\0';
    ntok = 0;
    tok = strtok(msg, " ");
    while (tok != NULL) {
      tokens[ntok++] = tok;
      tok = strtok(NULL, " ");
    }
    cmd = tokens[0];

    if(strcmp(cmd, "xframe?") == 0) {
      Serial.print(xframe);
      Serial.print(" OK\n");
    }
  }
}

```

**Figure 3: Host Command Parsing**

In order to test the clocking functions, test code was created that would loop the function call so it would keep repeating. This is what allows an oscilloscope to display what the function is outputting. If this was not done nothing would be visible for observation as each function call is ran once quicker then we can see when called on by the cost software. For this code specifically I put a question mark next to test code, it would run until a 'q' was entered as a command to quit the test loop. Figure 4 shows an example of this testing method.

```

else if(strcmp(cmd, "vertical?") == 0) {
  bool test=true;
  Serial.print("vertical_shift() TEST\n");
  while(test) {
    if (Serial.available() != 0 && Serial.read() == 'q')
      test=false;
    vertical_shift();
  }
  Serial.print("vertical_shift() test over\n");
}

```

**Figure 4: Test code for vertical\_shift()**

Last part of the code are the different function blocks which comprise of most of the “math” behind making everything work. Figure 5 shows a function block for horizontal\_flush(). The function is called above and then executed below. When function blocks are used at the bottom of the code, a prototype function must be inserted at the top of the code to prompt the compiler as to what type of data is going to be used later in the code. This is an initialization of the function similar to initializing a variable.

```
void horizontal_flush() {
    for (int k=0; k<xframe; ++k) {
        RST_CTRL_HIGH;
        H_CLK_LOW;
        delayNanoseconds(50);
        RST_CTRL_LOW;
        delayNanoseconds(150);
        H_CLK_HIGH;
        delayNanoseconds(200);
    }
}
```

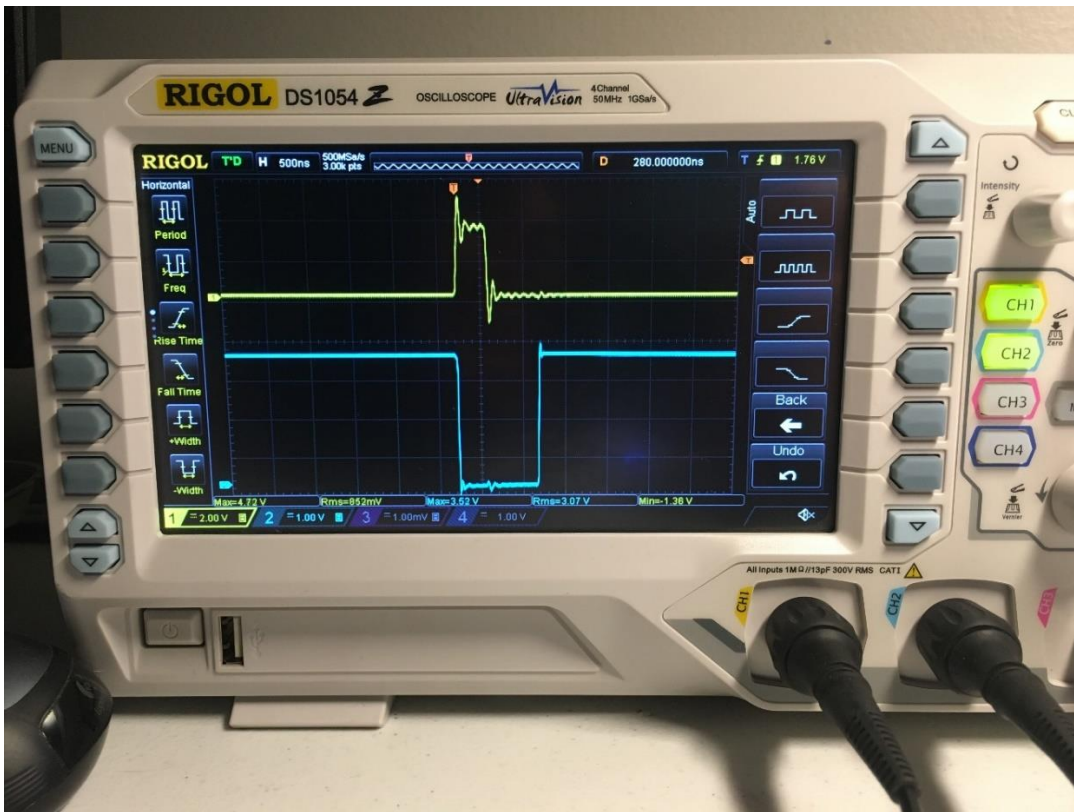
**Figure 5:** Test code for horizontal\_flush()

There are many functions that were needed to make the code work successfully. Figure 6 shows the function blocks that were needed. The prototype of each one is listed with a brief description of what its purpose is.

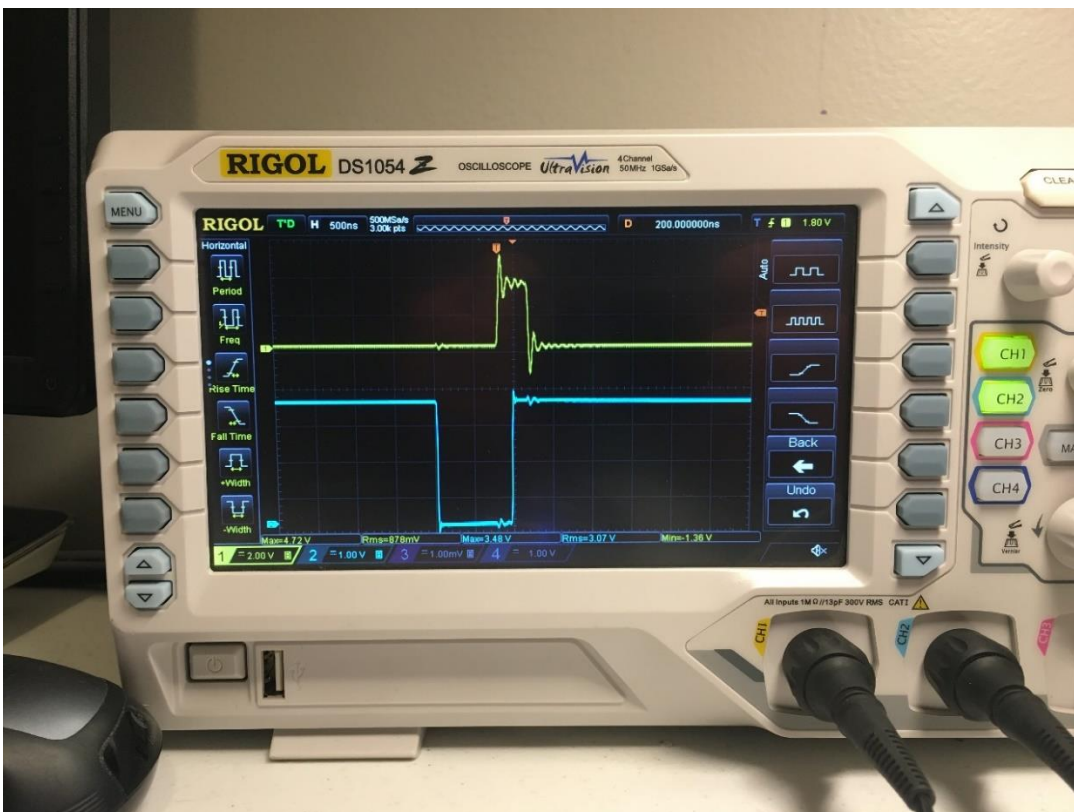
|  |  |
|--|--|
| void vertical_shift();                     | Move scan line of image to bottom into horizontal shift register |
| void horizontal_flush();                   | Clears all accrued charges from the CCR                          |
| void horizontal_shift(unsigned char* buf); | Read image data  |
| void config9826(int cdsMode);              | Configure 9826 chip  |
| void set9826(ushort addr, ushort val);     | Set values for 9826  |
| void dacOn();                              | Initialize DAC   |
| void dacOutput(int chn, double volt);      | Set output voltages  |
| void dacsClear();                          | Clear DAC values   |
| inline unsigned char read9826();           | Read bytes from 9826   |
| double readTemp();                         | Read temperature of CCD  |

After all functions are defined, chips are programed, and host can communicate with the embedded software, the testing will occur and the results can be found in the testing section below. The testing that could be done remotely without the board was the clocking and checking enables of the chip. The DAC output voltages couldn't be confirmed as the chip was not present to do so.

## Testing Results

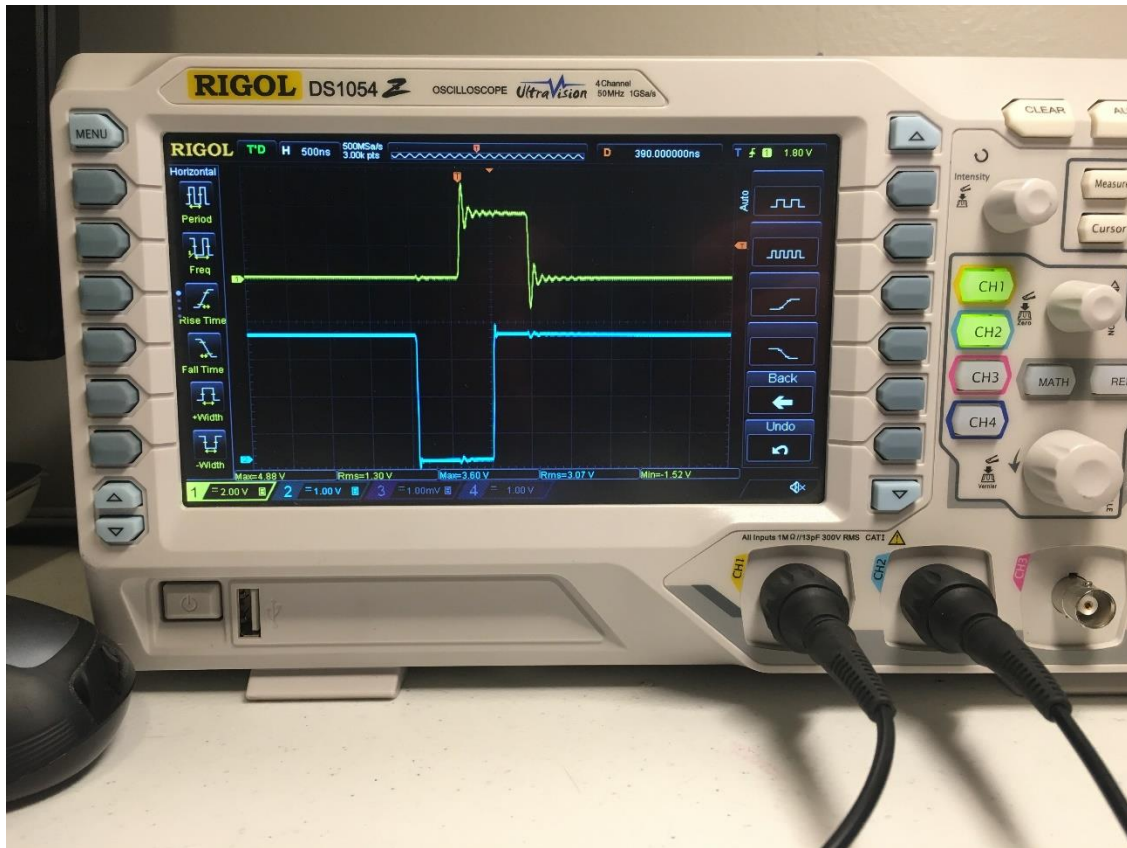


**Figure 1:** CDS1 (GPIO\_9) goes high for 200ns while ADCLK (GPIO\_7) is low for 800ns.



**Figure 2:** ADCLK (GPIO\_7) goes high 150ns after CDS2 (GPIO\_8) goes high for 300ns





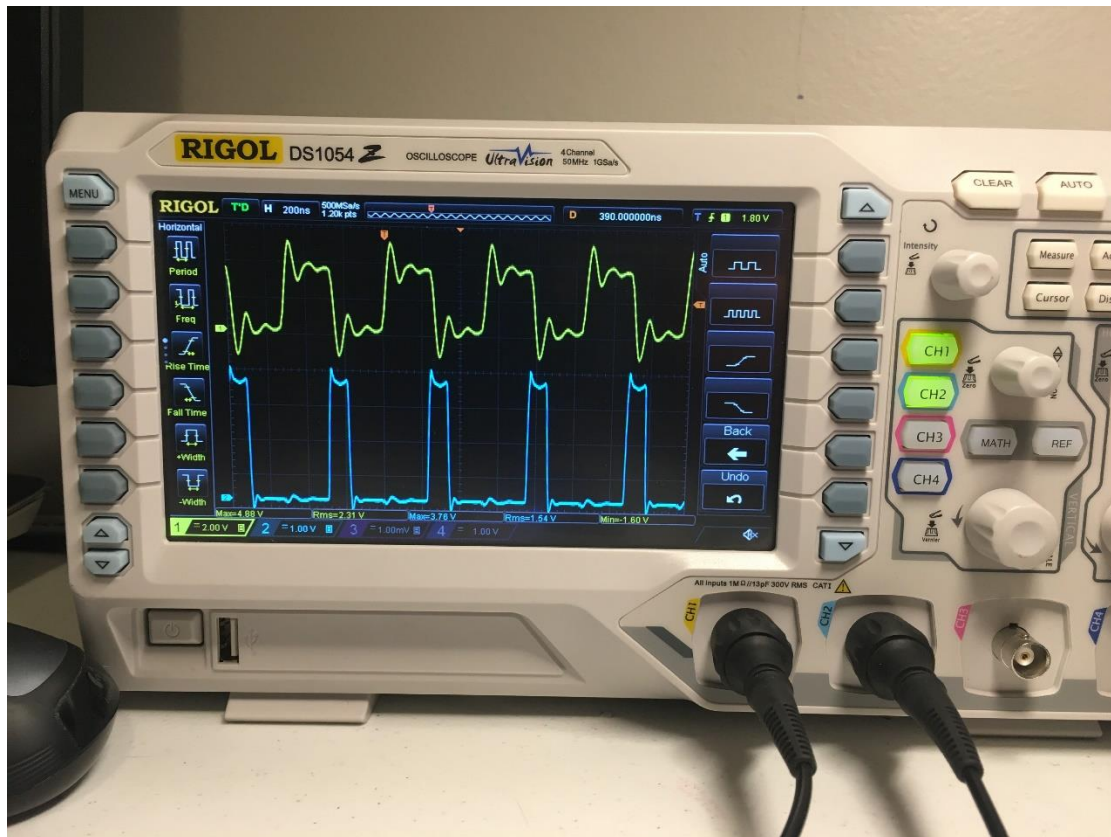
**Figure 3:** ADCLK (GPIO\_7) goes low 450ns before H\_CLK (GPIO\_4) goes high for 700ns

#### **Testing Procedure:** vertical\_shift()

To test the 4 clock signals I used the ADCLK at GPIO\_7 as a reference. It is represented in figures 1-3 by the bottom blue signal on the oscilloscope. Observing that the other 3 signals appear to be appropriately aligned with the ADCLK signal as the reference it is reasonable to believe that the clocking is correct. Since the pulses are only called once in the program, a testing mode was set up in the code to call each clocking function in a loop to allow measurements to be taken.

#### **Lessons Learned:** vertical\_shift()

A future improvement of this test would be to have 4 channels running at the same time and a higher frequency oscilloscope so all 4 signals could be compared at once for greater confidence that they all clock correctly. Setting up the clocking took multiple iterations and rereading the clocking diagram to get it correct. Found it was easier to sort through problems by checking each high and low against  $t=0$  to see if the response time of the pulse was correct through the whole graph instead of against another pulse that may or may not be correct.



**Figure 4:** RST\_CTRL (GPIO\_5) goes high for 50nS while H\_CLK (GPIO\_4) goes low for 800nS

#### **Testing Procedure:** horizontal\_flush()

To test the flushing function a test function was created that would loop the horizontal\_flush() function in a continuous loop so a reading on the oscilloscope could be read to determine if the function was working correctly.

#### **Lessons Learned:** vertical\_shift()

This function seemed to be a little nosier. I did get better results with a different board when I was doing testing, first bread board I used happen to be a cheaper one that I just had at hands reach. Also ended up switching a pulse train to another output to try and trouble shoot if that pin was the issue. Issue ended up being with my code, I put an added delay to try and put a gap between loop iterations, but instead of a space I got an incorrect output without a space. The timing was thrown way out of sync.

#### **Individual Project Reflection:**

Despite the unprecedented circumstances I greatly enjoyed your class. I definitely learned a lot and got to advance my C coding capabilities and really get to put a full system together from an embedded software stand point. It is unfortunate that we didn't get to do more in the lab as I was looking forward to that lab time. I greatly appreciate the fact you have extensive industry knowledge to share with us. Other then the whole COVID situation I would say the most "frustrating" part was trying to do the embedded as a "team" because the cross collaboration just wasn't there when we would meet up. I enjoy leaning others programing styles compared to mine, it helps with my growth in software. The positive was it forced me to work on almost all of it, increasing my understanding of the code as a whole.

