```
// Connor Noddin
// ECE 406
// Computer Engineering Capstone
// ADNS3050.cpp

#include <SPI.h>
#include "ADNS3050.h"

/*
 Description: Beings communications using
 SS/NCS pin
*/
void com_start() {
  digitalWrite(PIN_NCS, HIGH);
  delay(20);
  digitalWrite(PIN_NCS, LOW);
}

/*
 Description: Reads a register from the ADNS3050 sensor
*/
byte Read(byte reg_addr) {
  digitalWrite(PIN_NCS, LOW);  //begin communication
  // send address of the register, with MSBit = 0 to say it's reading
  SPI.transfer(reg_addr & 0x7f);
  delayMicroseconds(100);
  // read data
  byte data = SPI.transfer(0);
  delayMicroseconds(30);
  digitalWrite(PIN_NCS, HIGH);  //end communication
  delayMicroseconds(30);

  return data;
}

/*
 Description: Writes to a register on the ADNS3050
*/
void Write(byte reg_addr, byte data) {
  digitalWrite(PIN_NCS, LOW);
  //send address of the register, with MSBit = 1 to say it's writing
  SPI.transfer(reg_addr | 0x80);
  //send data
  SPI.transfer(data);
  delayMicroseconds(30);
  digitalWrite(PIN_NCS, HIGH);  //end communication
  delayMicroseconds(30);
}

/*
 Description: Starup protocol. Must be run to initialize sensor
*/
void startup() {
  //--------Setup SPI Communication---------
  /*
  byte out = 0;
  byte read = 0;
  byte bit = 0;
  */
  pinMode(PIN_MISO, INPUT);
  pinMode(PIN_NCS, OUTPUT);
  SPI.begin();
  // set the details of the communication
  SPI.setBitOrder(MSBFIRST);              // transimission order of bits
  SPI.setDataMode(SPI_MODE3);             // sampling on rising edge
  SPI.setClockDivider(SPI_CLOCK_DIV16);  // 16MHz/16 = 1MHz
  delay(10);
```

```
  //----------------- Power Up and config ---------------
  com_start();
  Write(RESET, 0x5a);        // force reset
  delay(100);                // wait for it to reboot
  Write(MOUSE_CTRL, 0x20);   //Setup Mouse Control
  Write(MOTION_CTRL, 0x00);  //Clear Motion Control register
  delay(100);
  Write(MISC_SETTINGS, B0000001); // Liftoff detection disable
  delay(100);
  Write(MOUSE_CTRL,B11000); // 2000 DPI
  delay(100);
}

/*
 Description: Convert register value from 2s complement to a regular
 32 bit integer
*/
int convTwosComp(int b) {  //Convert from 2's complement
  if (b & 0x80) {
    b = -1 * ((b ^ 0xff) + 1);
  }
  return b;
}

/*
 Description: Reads X movement from sensor register
*/
int getX() {  //returns the X acceleration value
  byte x = 0;
  x = Read(0x03);
  return (convTwosComp(x));
}

/*
 Description: Reads Y movement from sensor register
*/
int getY() {  //returns the Y acceleration value
  byte y = 0;
  y = Read(0x04);
  return (convTwosComp(y));
}
```

```
// Connor Noddin
// ECE 406
// Computer Engineering Capstone
// ADNS3050.h

#ifndef ADNS3050_H_  //Guard
#define ADNS3050_H_

// Pins used. These are default for the ATMEGA
#define PIN_SCLK 13
#define PIN_MISO 12
#define PIN_MOSI 11
#define PIN_NCS 10    //Digital Pin 10... SS
#define PIN_MOTION 9  //Digital Pin 9 is free

// All registers for ADNS3050
#define PROD_ID 0x00
#define REV_ID 0x01
#define MOTION_ST 0x02
#define DELTA_X 0x03
#define DELTA_Y 0x04
#define SQUAL 0x05
#define SHUT_HI 0x06
#define SHUT_LO 0x07
#define PIX_MAX 0x08
#define PIX_ACCUM 0x09
#define PIX_MIN 0x0a
#define PIX_GRAB 0x0b
#define MOUSE_CTRL 0x0d
#define RUN_DOWNSHIFT 0x0e
#define REST1_PERIOD 0x0f
#define REST1_DOWNSHIFT 0x10
#define REST2_PERIOD 0x11
#define REST2_DOWNSHIFT 0x12
#define REST3_PERIOD 0x13
#define PREFLASH_RUN 0x14
#define PREFLASH_RUN_DARK 0x18
#define MOTION_EXT 0x1b
#define SHUT_THR 0x1c
#define SQUAL_THRESHOLD 0x1d
#define NAV_CTRL2 0x22
#define MISC_SETTINGS 0x25
#define RESOLUTION 0x33
#define LED_PRECHARGE 0x34
#define FRAME_IDLE 0x35
#define RESET 0x3a
#define INV_REV_ID 0x3f
#define LED_CTRL 0x40
#define MOTION_CTRL 0x41
#define AUTO_LED_CONTROL 0x43
#define REST_MODE_CONFIG 0x45

void com_start();

byte Read(byte reg_addr);

void Write(byte reg_addr, byte data);

void startup();

int convTwosComp(int b);

int getX();

int getY();

#endif  // ADNS3050_H_
```

```
// Connor Noddin
// ECE 406
// Computer Engineering Capstone
// maine_3050.cpp

#include <SPI.h>
#include <avr/pgmspace.h>
#include "ps2mouse.h"
#include "ADNS3050.h"

// Allows device to send packets
int DEVICE_ENABLED = 0;  //Flag for if host sent device enabled signal

/*
Description: Initial setup, runs on boot only once.
Establishes GPIO pins and sensor. Also sends initial packet sequence
*/
void setup() {

  //Initiate Serial communication for debugging
  Serial.begin(SERIAL_RATE);  //9600 bits/second (Baud rate)

  delay(INIT_DELAY);  // 500 ms delay for PS/2 standard

  // Initialize the mouse buttons as inputs:
  pinMode(LEFT, INPUT);
  pinMode(MIDDLE, INPUT);
  pinMode(RIGHT, INPUT);

  // In assignments for reading data and clock bus
  pinMode(DATA_IN, INPUT);
  pinMode(CLK_IN, INPUT);

  // Out assignments for controlling data and clock bus
  pinMode(DATA_OUT, OUTPUT);
  pinMode(CLK_OUT, OUTPUT);

  startup();  // Begin ADNS 3050 sensor

  // Write self test passed
  while (ps2_dwrite(BAT) != 0);
  // Write mouse ID
  while (ps2_dwrite(ID) != 0);
}

/*
Description: Runs indefinitely. Establishes handshake with PS2_host.
Then, reads sensor and buttons. Finally sends 3 data packets to PS2_host.
*/
void loop() {
  byte* data;  //3 data packets
  byte tmp;    //Temporary byte from functions

  // Check if host is trying to send commands
  if (((digitalRead(DATA_IN) == LOW) || (digitalRead(CLK_IN) == LOW)) && DEVICE_ENABLED ==
0) {
    while (ps2_dread(&tmp));  // If this fails it halts the program
    ps2_command(tmp);
    if (tmp == ENABLE) DEVICE_ENABLED = 1;
  }

  /*
  Serial.print("\n");
  Serial.print("Sensor X: ");
  Serial.print(sensor_x, DEC);
  Serial.print("\t Sensor Y: ");
  Serial.print(sensor_y, DEC);
  Serial.print("\n");
```

```
  delay(5);
  */

  // Writes data to host
  if (DEVICE_ENABLED == 1 || FORCE_ENABLE == 1) {

    data = get_bytes();  // Gets all data from sensors

    //Writes to DATA and CLK lines
    ps2_dwrite(data[0]);
    ps2_dwrite(data[1]);
    ps2_dwrite(data[2]);

    delay(DATA_DELAY);  // Delay between bytes. Increases stability

    /*
    Serial.print("Byte 1: 0x");
    Serial.print(byte_1, HEX);
    Serial.print("\t Sensor X: ");
    Serial.print(sensor_x, DEC);
    Serial.print("\t Sensor Y: ");
    Serial.print(sensor_y, DEC);
    Serial.print("\n");
    */
  }
}
```

```
// Connor Noddin
// ECE 406
// Computer Engineering Capstone
// ps2mouse.cpp

#include "ps2mouse.h"
#include <avr/pgmspace.h>
#include <Arduino.h>
#include "ADNS3050.h"


/*
 Description: Sets the clock to 1, then back to 0 after a delay
 The clock frequency is 10-16.7 kHz.
 Data is read from device to host on FALLING edge
 Data is read from host to device on RISING edge
*/
int ps2_clock(void) {
  delayMicroseconds(CLOCK_HALF);
  digitalWrite(CLK_OUT, HIGH);  //This is inverted
  delayMicroseconds(CLOCK_FULL);
  digitalWrite(CLK_OUT, LOW);  //This is also inverted
  delayMicroseconds(CLOCK_HALF);
  return 0;
}

/*
Description: Sends 11 bit packet to PS2 data line if communication
is not being inhibited
Summary: Bus States
Data = high, Clock = high:  Idle state.
Data = high, Clock = low:  Communication Inhibited.
Data = low, Clock = high:  Host Request-to-Send
*/
int ps2_dwrite(byte ps2_Data) {

  int p = parity(ps2_Data);  //Gets parity before bit manipulation

  delayMicroseconds(BYTE_DELAY);  //Delay between bytes

  // Never transmit if the host is inhibiting communication
  if (digitalRead(CLK_IN) == LOW) {
    return -1;
  } else if (digitalRead(DATA_IN) == LOW) {
    return -2;
  }

  // First bit is always 0
  //Logic is inverted because open collector
  digitalWrite(DATA_OUT, HIGH);
  ps2_clock();

  //Send entire byte, LSB first
  //Logic is inverted because open collector
  for (int i = 0; i < 8; i++) {
    if ((ps2_Data & 0x01) == 0x01) digitalWrite(DATA_OUT, LOW);  //Writes high if least sig
nificant bit is 0
    else digitalWrite(DATA_OUT, HIGH);                           //Writes low if least sign
ificant bit is 0
    ps2_clock();                                                 //Clocks current data
    ps2_Data = ps2_Data >> 1;                                    //Get next bit
  }

  // The parity bit is set if there is an even number of 1's
  // in the data bits and reset (0) if there is an odd number of 1's in the data bits
  // The parity bit forces odd parity in the packet
  if (p == 1) {
    digitalWrite(DATA_OUT, HIGH);  //Low if odd number of ones
```

```
    ps2_clock();
  } else {
    digitalWrite(DATA_OUT, LOW);  // High if even number of ones
    ps2_clock();
  }

  // Stop bit is always 1
  digitalWrite(DATA_OUT, LOW);  // Always high
  ps2_clock();

  delayMicroseconds(BYTE_DELAY);  //Delay between bytes

  return 0;
}

/*
Description: Reads 11 bit data packet from PS2 data line.
Also sends ACK bit, for a total 12 bit packet. This also
detects if the host timeouts
*/
int ps2_dread(byte* read_in) {
  unsigned int data = 0x00;
  unsigned int bit = 0x01;

  unsigned char calculated_parity = 1;
  unsigned char received_parity = 0;

  // Only reads when CLK is pulled low
  // Timesouts if host has not sent for 30 ms
  // This is effectively the start bit
  unsigned long init = millis();
  while (((digitalRead(DATA_IN) != LOW) || (digitalRead(CLK_IN) != HIGH))) {
    if ((millis() - init) > HOST_TIMEOUT) {
      Serial.println("Read failed, host timeout!");
      return -1;
    }
  }

  //First packet bit is here which is always 0!
  ps2_clock();

  // Reads 8 data bits with LSB first
  for (int i = 0; i < 8; i++) {

    if (digitalRead(DATA_IN) == HIGH) {
      data = data | bit;
      calculated_parity = calculated_parity ^ 1;
    }

    bit = bit << 1;
    ps2_clock();
  }

  // parity bit ... clock is from last iteration of loop
  if (digitalRead(DATA_IN) == HIGH) {
    received_parity = 1;
  }

  //Clock for stop bit
  ps2_clock();

  // Acks the packet that was just read
  digitalWrite(DATA_OUT, HIGH);
  delayMicroseconds(CLOCK_HALF);
  digitalWrite(CLK_OUT, HIGH);
  delayMicroseconds(CLOCK_FULL);
  digitalWrite(CLK_OUT, LOW);
  delayMicroseconds(CLOCK_HALF);
```

```
  digitalWrite(DATA_OUT, LOW);

  // Returns data to main loop
  *read_in = data & 0x00FF;

  //Parity check
  if (received_parity == calculated_parity) {
    return 0;
  } else {
    //Parity is wrong and an error occured
    Serial.print("Calculated Parity: ");
    Serial.print(calculated_parity, DEC);
    Serial.print("\t Received Parity: ");
    Serial.println(received_parity, DEC);
    Serial.println("Parity error in read function!");
    return -2;
  }

  return 0;
}

/*
Description: Checks parity of byte. This is odd parity. Consequently, if the parity
of the byte is odd, this returns a one
*/
int parity(byte p_check) {
  byte ones = 0;  //Total number of ones

  for (int i = 0; i < 8; i++) {
    if ((p_check & 0x01) == 0x01) {
      ones++;  //Adds to parity if lowest bit is 1
    }
    p_check = p_check >> 1;  //Gets next bit
  }

  return (ones & 0x01);  //Checks if parity is odd
}

/*
Description: Takes a byte that was read from the host as an input
Then, based off the byte, sends the exact appropriate response.
This function includes every possible command the host can send to the device.
If the command is not in this list, it is not a legal command.
*/
int ps2_command(byte input) {

  unsigned char val;  //Value for when host is sending data for a command

  switch (input) {
    case RES:  //Resets mouse
      ack();
      while (ps2_dwrite(BAT) != 0);
      while (ps2_dwrite(ID) != 0);
      break;
    case RESEND:  //Error occured
      ack();
      break;
    case DEF:  //Sets defaults
      ack();
      break;
    case DISABLE:  //Disables data reporting
      ack();
      break;
    case ENABLE:  //Enables data reporting ...0xF4
      Serial.println("Enable signal received!");
      ack();
      break;
    case SET_RATE:  //Reads sample rate
```

```
      ack();
      ps2_dread(&val);
      ack();
      break;
    case GET_DEV_ID:  //Gets device id
      ack();
      ps2_dwrite(ID);  //0x03 also works
      break;
    case REMOTE:  //Sets remote mode
      ack();
      break;
    case WRAP:  //Sets wrap mode
      ack();
      break;
    case RES_WRAP:  //Resets wrap mode
      ack();
      break;
    case READ:  //Reads data
      ack();
      break;
    case STREAM:  //Enables stream mode
      ack();
      break;
    case STATUS_REQ:  //Requests status of mouse
      ack();
      while (ps2_dwrite(ID) != 0);
      while (ps2_dwrite(STAT_1) != 0);
      while (ps2_dwrite(STAT_2) != 0);
      break;
    case SET_RES:  //Sets resolution of sensor
      ack();
      ps2_dread(&val);
      ack();
      break;
    case SCALE_2:  //Sets scaling 2:1
      ack();
      break;
    case SCALE_1:  //Sets scaling 1:1
      ack();
      break;
    default:
      Serial.print("Unknown command.... likely an error in the read function");
      break;
  }
  Serial.print("Sent response to command: 0x");
  Serial.println(input, HEX);
  return 0;
}

/*
Description: Acks a host command. Per the PS2 standard,
this must complete during the handshake. Consequently, this keeps
attempts to do so until successfull.
*/
void ack() {
  while (ps2_dwrite(ACK) != 0);  //0xFA
}

/*
Description: Returns the states of the buttons as the least
significant bits of a single byte. Works for a button being
off as well as on
*/
byte get_button_states(void) {
  int leftState, middleState, rightState;  // If button is pressed or not

  byte buttons = 0x00;
```

```
  // read the state of the pushbutton value:
  leftState = digitalRead(LEFT);
  middleState = digitalRead(MIDDLE);
  rightState = digitalRead(RIGHT);

  //Default state of switches is high
  // Check the value of the left mouse button
  if (leftState == LOW) {
    buttons = buttons | L_BUTTON;
  } else {
    buttons = buttons & ~L_BUTTON;
  }
  // Check the value of the right mouse button
  if (rightState == LOW) {
    buttons = buttons | R_BUTTON;
  } else {
    buttons = buttons & ~R_BUTTON;
  }
  // Check the value of the middle mouse button
  if (middleState == LOW) {
    buttons = buttons | M_BUTTON;

  } else {
    buttons = buttons & ~M_BUTTON;
  }

  return buttons;
}

/*
Description: Returns 3 bytes. First is control, second is X movement, third
is Y movement
*/
byte* get_bytes(void) {

  byte tmp;                                    //Temporary byte from functions
  int sensor_x, sensor_y;                      //Sensor x and y movement
  static byte data[3];                         //3 data packets
  byte byte_1 = BYTE_1_BIT, byte_2, byte_3;  // 3 bytes for PS/2 packet

  tmp = get_button_states();  // Gets state of all three buttons

  byte_1 = byte_1 | tmp;  //Saves states to byte 1

  // Code for sensor
  sensor_x = getX();  //Extract change in x
  sensor_y = getY();  //Extract change in y

  //Sets sign bits from sensor
  if (sensor_x < 0)
    byte_1 = byte_1 | X_SIGN;
  else
    byte_1 = byte_1 & ~X_SIGN;

  if (sensor_y < 0)
    byte_1 = byte_1 | Y_SIGN;
  else
    byte_1 = byte_1 & ~Y_SIGN;

  //Sets overflow bits from sensor in 2s compliment
  if (sensor_x > 255) {
    byte_1 = byte_1 | X_OVERFLOW;
    sensor_x = 0xFF;
  } else {
    byte_1 = byte_1 & ~X_OVERFLOW;
  }
  if (sensor_x < -255) {
    byte_1 = byte_1 | X_OVERFLOW;
```

```
    sensor_x = 0x01;  //2s compliment, sign bit above
  } else {
    byte_1 = byte_1 & ~X_OVERFLOW;
  }

  if (sensor_y > 255) {
    byte_1 = byte_1 | Y_OVERFLOW;
    sensor_y = 0xFF;
  } else {
    byte_1 = byte_1 & ~Y_OVERFLOW;
  }
  if (sensor_y < -255) {
    byte_1 = byte_1 | Y_OVERFLOW;
    sensor_x = 0x01;  //2s compliment, sign bit above
  } else {
    byte_1 = byte_1 & ~Y_OVERFLOW;
  }

  //Gets lower 8 bits of both sensor data for movement
  byte_2 = sensor_x & 0x00FF;
  byte_3 = sensor_y & 0x00FF;

  data[0] = byte_1;
  data[1] = byte_2;
  data[2] = byte_3;

  return data;
}
```

```
// Connor Noddin
// ECE 406
// Computer Engineering Capstone
// ps2mouse.h

#ifndef ps2mouse_H_  //Guard
#define ps2mouse_H_

#include <Arduino.h>

// Packet Manipulation
#define R_BUTTON 0x02
#define L_BUTTON 0x01
#define M_BUTTON 0x04
#define X_SIGN 0x10
#define Y_SIGN 0x20
#define X_OVERFLOW 0x40
#define Y_OVERFLOW 0x80
#define BYTE_1_BIT 0x08

// PS/2 Commands
#define RES 0xFF  //Reset
#define RESEND 0xFE
#define DEF 0xF6  //Default
#define DISABLE 0xF5
#define ENABLE 0xF4
#define SET_RATE 0xF3
#define GET_DEV_ID 0xF2
#define REMOTE 0xF0
#define WRAP 0xEE
#define RES_WRAP 0xEC
#define READ 0xEB
#define STREAM 0xEA
#define STATUS_REQ 0xE9
#define SET_RES 0xE8
#define SCALE_2 0xE7
#define SCALE_1 0xE6

#define BAT 0xAA
#define ACK 0xFA
#define ID 0x00
#define STAT_1 0x02
#define STAT_2 0x64


// Misc Setup
#define FORCE_ENABLE 0    //Good for debugging data packets. Will make most hosts not work
#define CPI 1000          // Counts per inch of sensor
#define SERIAL_RATE 9600  // Bits/second

//Timings - This is what affects the performance and stability of the mouse
#define INIT_DELAY 2500  //Delay before sending first packets... 2500 for USB adapter. 500
for motherboard
#define CLOCK_HALF 20    //10 - 16.7khz clock is standard
#define CLOCK_FULL 40
#define BYTE_DELAY 3000    //1500 works good. Higher is more stable
#define HOST_TIMEOUT 1000  //30 is default
#define TIMEOUT 1000
#define DATA_DELAY 4  // Lower is smoother but less stable

#define SS 10  // SS pin on arduino. For nano 10 is default. Uno 3 is default

// GPIO pin assignments for mouse buttons
#define LEFT 2    // Left mouse button
#define MIDDLE 3  // Middle mouse button
#define RIGHT 4   // Right mouse button

// GPIO pin assignments for PS/2 connection
```

```
#define DATA_IN_A A2  //normally 8
#define DATA_IN 8
#define CLK_IN 7
#define DATA_OUT 6
#define CLK_OUT 5

int ps2_clock(void);

int ps2_dwrite(byte ps2_Data);

int ps2_dread(byte* read_in);

int ps2_command(byte input);

void ack();

int parity(byte p_check);

byte get_button_states(void);

byte* get_bytes(void);

#endif  // ps2mouse_H_
```