Introduction

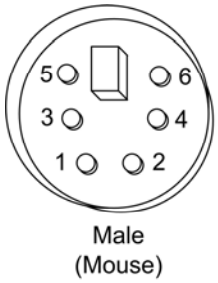
A computer mouse provides an excellent device to acquire 2D coordinate-based user input, since most users are already familiar with it. Most mice usually come with two or three buttons, though five is not uncommon. Additionally, some mice provide control of an extra axis of motion through the use of a scroll wheel. A computer mouse may be easily interfaced with a microcontroller for use with a multitude of projects, from robotics to graphics applications.

Computer mice use a few different types of protocols to interface with a host. PS/2 mice are the most common, with RS-232 mice losing popularity and USB mice gaining popularity. This documentation covers how to communicate with PS/2 mice using a PIC microcontroller. Most USB mice are bundled with a USB-to-PS/2 converter for use with older computers that don't support USB, so this documentation should work equally well with such mice.

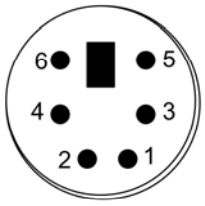
Interfacing with a PS/2 mouse involves two components: the PS/2 protocol and the PS/2 mouse interface. The PS/2 protocol is not specific to mice alone. Other devices, such as keyboards, stylus pads, and barcode scanners use the PS/2 protocol. The mouse demonstrated in this paper is a two-button mechanical PS/2 mouse, though optical mice and USB mice with USB to PS/2 adapters should work the same.

PS/2 Protocol

The physical connector for PS/2 is a 6-Pin Mini-DIN, as shown in Figure 1. Only four of the six pins are used: data, ground, VCC, and clock. It is recommended to strip the wires of the PS/2 extension cable listed in the supplier list of this guide to provide a female PS/2 plug that the mouse's male plug can connect to, so that you can connect any PS/2 mouse to your project and not have to resolder if a mouse failure occurs. The PS/2 extension cable uses the following colors for each pin: data is brown, ground is orange, VCC is yellow, and clock is green. PS/2 uses TTL voltages levels (+5VDC), though the mouse demonstrated in this documentation appears to work at +3.3VDC (the operating voltage of the 18F452 PIC on the E155 utility board) as well. PS/2 data is sent serially between the mouse and the host. Communication is bi-directional; the mouse can send movement data to the host, as well as receive commands by the host. Data is sent in an 11-bit sequence called a *data packet* on the data pin. The format of a packet will be described on the next page.



Male
(Mouse)



Female
(Host)

Pin	Description
1	Data
2	N/C
3	Ground
4	VCC
5	Clock
6	N/C

Figure 1: PS/2 6-Pin Mini-DIN Connector

The mouse and the host communicate using an open-collector bus which allows for bi-directional communication. An open collector is a line with a weak pull-up resistor and multiple devices attached, which are normally in a high-impedance state. Because of the weak pull-up resistor the bus is normally high, but allows any of the connected devices to drive the line low and send data.

Sending data with PS/2 works by: the host or mouse initiating communication, the mouse generating a clock signal by driving clock pin high and low, and the host or mouse sending data in sync with clock signal. Sample PS/2 communication (host to mouse and mouse to host) is shown in Figure 2. The mouse or the host initiates communication by driving the clock low for at least 100 µs (not shown in the figure). When the clock is released (no longer driven low), the first bit of the data packet (which is always zero/low) must be driven on the data line by whoever is sending. If the data line is not zero when clock is released, that implies that no communication is being initiated and the mouse will not begin generating a clock signal needed for data to be sent. Note that the mouse is not allowed to send data when the clock signal is being driven to ground by the host, or 50 µs thereafter, even if it is in the middle of sending a packet. Hence to inhibit communication, the host may drive clock low until it is ready to process data from the mouse. A data transmission from the mouse is aborted if communication is inhibited for 100 µs or more (i.e. if the clock is driven low by the host for more than 100 µs). The mouse is required to buffer data it wishes to transmit. Though keyboards buffer the last 16 keystrokes, mice only buffer the last movement.

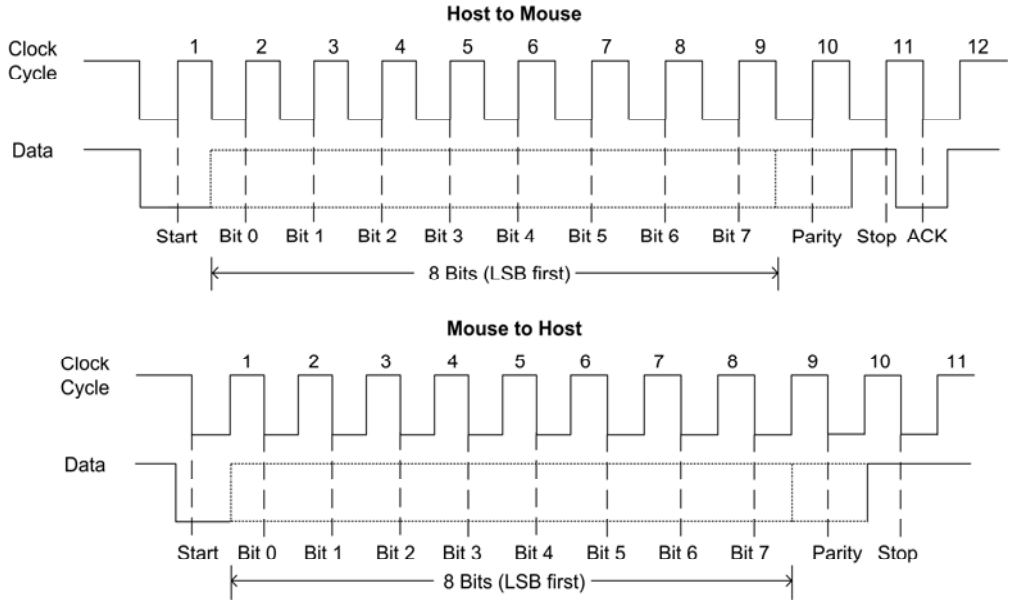


Figure 2: PS/2 Communication

After communication is initiated, the mouse begins generating a clock signal between 10 kHz and 16.7 kHz (a 60 to 100 µs clock period), which must begin within 15ms from when the host drove the clock low for host-to-mouse communication. Then the sender begins sending data on the data line. For host-to-mouse communication, data is read by the mouse on the rising edge of the clock, and for mouse-to-host communication data is read on the falling

edge. Therefore it is convenient for the host to drive the data pin to the next data bit on the falling edge when sending.

A data packet is the same for both host-to-mouse and mouse-to-host communication, with the exception of an extra acknowledgement bit sent by the mouse during host-to-mouse communication. A data packet consists of a start bit, which is always 0; followed by 8 bits of data (least significant bit first); then a parity bit, which is 0 if there are an odd number of 1's in the 8 bits of data, and 1 if there are an even number of 1's; and lastly a stop bit, which is always 1. For host-to-mouse communication, when the host stops driving the data pin at the end of the stop bit, the mouse drives the data pin low for one clock cycle to acknowledge it received the data.

The easiest way to connect a mouse to the PIC is by connecting two I/O ports capable of weak pull-up to the clock and data signals, as shown in Schematic 1 in the sample schematic section. In the idle state, the two I/O ports of the PIC should be configured as input with internal weak pull-ups. The PIC 18F452 supports internal weak pull-ups on the pins of PORTB. It can be configured by clearing bit 7 of the INTCON2 file register (to enable weak pull-ups on PORTB) and setting all bits of TRISB (to configure PORTB as input). Clock and data may then be polled by the PIC to determine if the mouse is sending data. When the PIC wishes to transmit data, it sets the two I/O ports to be outputs (by clearing bits of TRISB), drives both low, and then stops driving (releases) the clock pin by setting it back to being an input pin. The mouse will then begin generating clock signals while the PIC sends the appropriate data bits on the data line.

PS/2 Mouse Interface

PS/2 mice can receive and respond to commands as well as send movement data to the host, over the PS/2 protocol described above. Internally, mechanical mice have two 9-bit 2's-complement counters, one for the x axis and one for the y axis, which keep a count of how much a rotating disc has moved, corresponding to the movement of a rubber ball along a surface. Additionally, each counter has an associated overflow flag. The movement counters are reset after movement data is read by the host.

Commands may be sent to the mouse to configure settings and change the mode of operation. A mouse should respond to a command within 2 ms from when it was sent. A list of commands available for a host to send to a standard PS/2 mouse is given in Table 1. To initialize the mouse, first send Reset (0xFF). The mouse will go through a self-test and either return 0xAA for successful self-test, or 0xFC if there was an error initializing. If the self-test was successful it will then send a device id of 0x00. This indicates to the host that it is a standard PS/2 mouse, not another device such as a keyboard. After sending the device id, the mouse will then enter stream mode. At this point, the host may send any of the commands listed in Table 1 to the mouse. PS/2 mice can send Resend (0xFE), which is analogous to the Resend (0xFE) for host-to-mouse communication, as described below, for when the mouse incorrectly receives data (such as parity check failing). When the host receives a Resend command, it must resend the last data packet it sent to the mouse.

PS/2 mice support three modes of operation besides reset: stream mode, remote mode, and wrap mode. Stream mode is the most common mode for PS/2 mouse operation. There is an

option for stream mode called data reporting. If data reporting is enabled (0xF4), the mouse will automatically send movement data to the host, if any movement occurred, limited by the maximum sample rate set by 0xF3 (covered on next page). When data reporting is disabled, the host must request a packet of data with the Read Data (0xEB) command. Data reporting should be disabled (0xF5) before any other commands are sent to the mouse. Remote mode is identical to stream mode with data reporting disabled, that is, movement packets must be requested explicitly by the host using Read Data (0xEB). Lastly, in wrap mode the mouse will echo data to the host that was sent to the mouse by the host, except for Reset (0xFF) and Reset Wrap Mode (0xEC), which are the only two commands available to exit wrap mode.

Two settings affect the usage of the movement counters. The *resolution* affects how much a movement counter is updated relative to the amount the mouse physically moved. Resolution is set by sending Set Resolution (0xE8) to the mouse, waiting for an acknowledgement (0xFA), and then sending the new resolution byte value. Four values are available for resolution: 0x00 is for 1 count/mm, 0x01 for 2 counts/mm, 0x02 for 4 counts/mm, and 0x03 for 8 counts/mm. Another setting called *scaling* affects the movement reported, but not the movement counters themselves. When scaling is 2:1 the reported movement is twice that of 1:1 scaling. For example, if the x-axis movement counter is 10, in 1:1 scaling the mouse will send 10 as the x-axis movement, whereas if the scaling is 2:1 the mouse will send 20 as the x-axis movement. Lastly, the sample rate controls the interval at which movement data is automatically sent from the mouse while data reporting is enabled in stream mode. The sample rate is an integer value between 10 to 200 samples per second.

Command	Name	Mouse Action
0xFF	Reset	Sends 0xFA (acknowledgement) and enters reset mode. In reset mode mouse will then send 0xAA for successful self-test along with 0x00 for the device id, or it will send 0xFC for failure.
0xFE	Resend	Sends last packet sent.
0xF6	Set Defaults	Sends 0xFA and resets settings to defaults. Does not change mode.
0xF5	Disable Data Reporting	Sends 0xFA and disables data reporting packets. Only used in stream mode. Also resets movement counters.
0xF4	Enable Data Reporting	Sends 0xFA and disables data reporting packets. Only used in stream mode. Also resets movement counters.
0xF3	Set Sample Rate	Sends 0xFA and reads another byte from the host, representing the new sample rate, and responds with another 0xFA.
0xF2	Get Device ID	Sends 0xFA and then sends its device id.
0xF0	Set Remote Mode	Sends 0xFA, resets movement counters, and enters remote mode.
0xEE	Set Wrap Mode	Sends 0xFA, resets movement counters, and enters wrap mode.
0xEC	Reset Wrap Mode	Sends 0FA, resets movement counters, and enters the mode mouse was in previous to entering wrap.
0xEB	Read Data	Sends 0xFA, sends a 3-byte movement packet, and resets movement counters.
0xEA	Set Stream Mode	Sends 0xFA, resets movement counters, and enters stream mode.
0xE9	Status Request	Sends 0xFA and then sends a 3-byte status packet.
0xE8	Set Resolution	Sends 0xFA, reads resolution byte from host, another 0xFA, and resets movement counters.
0xE7	Set Scaling 2:1	Sends 0xFA and sets scaling to 2:1.
0xE6	Set Scaling 1:1	Sends 0xFA and sets scaling to 1:1.

Table 1: PS/2 Mouse Commands

Default settings are 100 samples per second sample rate, 4 counts/mm resolution, 1:1 scaling, and data reporting disabled. The defaults are set whenever the mouse is Reset (0xFF) or when Set Defaults (0xF6) is sent to the mouse. The current status of the settings, along with button status, may be requested from the mouse with Status Request (0xE9). The packets returned by Status Request are shown in Figure 3. Stream mode is 1 if the mouse is currently in stream mode. If 0 it indicates the mouse is in remote mode. It is not possible to request the mouse status in wrap mode, because Status Request is not a valid command in this mode. Data reporting is 0 if data reporting is disabled and 1 if enabled. 2:1 scaling is 1 if 2:1 scaling is enabled; else the mouse is in 1:1 scaling.

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Byte 1	0	Stream Mode	Data Reporting	2:1 Scaling	0	Middle Button	Right Button	Left Button
Byte 2	Resolution (0x00, 0x01, 0x02, or 0x03)							
Byte 3	Sample Rate (0x0A to 0xC8)							

Figure 3: PS/2 Mouse Status Request Packets

Movement packets are sent from the mouse when either the host requests a packet with Read Data (0xEB), or when the mouse automatically sends out a movement packet while in stream mode with data reporting enabled. Movement packets are three bytes of data in a specific order, as shown in Figure 4. The first byte corresponds to various flags for button presses, sign bits, and overflow bits. The least-significant bytes of the x-axis and y-axis movement counters are then sent. Bit 3 of the first byte of the movement packet is always 1. No acknowledgement from the host is required after a packet is received.

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Byte 1	Y Overflow	X Overflow	Y Sign-bit	X Sign-bit	1	Middle Button	Right Button	Left Button
Byte 2	LSB 8-Bits of X Movement, since last packet.							
Byte 3	LSB 8-bits of Y Movement, since last packet.							

Figure 4: PS/2 Mouse Movement Packet

Handling Movement on the PIC

The most common way you will probably use x-axis and y-axis movement data is by storing an absolute position of a 'cursor' or some other point value in the PIC's memory. Every time a movement packet is received the PIC must add or subtract the relative movement with the current absolute position. Depending on the project, you must decide how sensitive the mouse will be, and how to handle overflows and underflows.

The demonstration in the sample code section stores the absolute position in two 16-bit unsigned integers, *xpos* and *ypos*. The code does not check for overflows or underflows, so the position will "wrap around" to the beginning/end. To reduced speed/sensitivity of mouse movement, the most-significant byte of *xpos* is driven on PORTD, therefore the LSB of *xpos* is used to "slow" movement. The coordinate information, such as button presses, sign flags, and overflow flags are stored in the unsigned char *coorinfo*. To access specific flags, bit-mask the coordinate information.

The sample code does not handle errors, such as Resend (0xFE) requests, parity checks, or timeouts while waiting for data. Depending on the needs and robustness of what you are designing you may want to implement error handling to varying degrees.

Specifications

PIC18CXX2 Data Sheet
http://ww1.microchip.com/downloads/en/DeviceDoc/39026c.pdf

Supplier

Part	Vendor	Part #	Price
4’ PS/2 Male/Female Cable	Digi-Key	AE-1118ND	$2.48
PS/2 Mouse	Microsoft	X03-67773	

www.digikey.com

Additional Resources

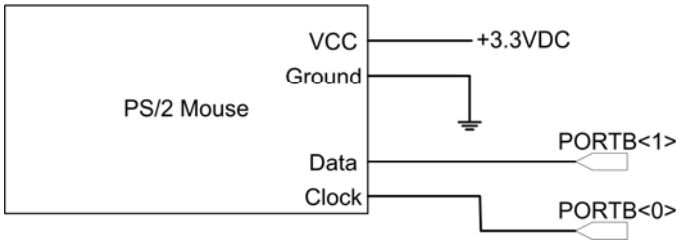
The PS/2 Mouse/Keyboard Protocol
http://www.computer-engineering.org/ps2protocol/

The PS/2 Mouse Interface
http://www.computer-engineering.org/ps2mouse/

Howstuffworks “How Computer Mice Work”
http://computer.howstuffworks.com/mouse3.htm

Schematics

Schematic 1: PS/2 Mouse Connected to PIC



Sample Code

Listing 1: ps2mouse.c

```
/* ps2mouse.c:  Demonstration on how to use a PS/2 Mouse.
 * Author: Nathaniel Pinckney <npinckney@hmc.edu>
 * Date: March 2005
 *
 * Assumes 20MHz clock.
 */

#include <p18f452.h>
#include <delays.h>
#include <timers.h>

#define clk PORTBbits.RB0
#define dataline PORTBbits.RB1
// Remember: TRIS has 0 for output and 1 for input.
#define clk_tris TRISBbits.TRISB0
#define dataline_tris TRISBbits.TRISB1
// one microsecond delay, for Delay10TCYx
#define US_DELAY 0.5

#define MS_RESET                                0xff
#define MS_DATA_REPORTING_ENABLED               0xf4

#define MS_SUCCESS                              0xaa
#define MS_DEVICE_ID                            0x00
#define MS_ACKNOWLEDGEMENT                      0xfa

#define COORINFO_XSIGN                          0x10
#define COORINFO_YSIGN                          0x20


/* Prototypes */
void wait(unsigned char t);
void sendData(unsigned char data);
unsigned char recvData(void);


void sendData(unsigned char data) {
       char i;
       unsigned char parity = 1;
       dataline_tris = 0;            // Drive data pin
       clk_tris = 0;                 // Drive clock pin.
       clk = 0;                      // Drive clock pin low.
       Delay10TCYx(100*US_DELAY);    // Wait a bit, 100 us.
       dataline = 0;                 // Drive data pin low.
       Delay10TCYx(10*US_DELAY);     // Wait 10 us.
       clk_tris = 1;                 // Stop driving clock.

       for(i=0; i < 8; i--) {
```

```
                while(clk) Delay10TCYx(10*US_DELAY); // Wait until clock is low.
                dataline = data & 0x1;               // Send bit.
                parity += data & 0x1;                // Calculate parity.
                data = data >> 1;                    // Shift data.
                while(!clk) Delay10TCYx(10*US_DELAY); // Wait until clock is high.
        }
        while(clk) Delay10TCYx(10*US_DELAY);  // Wait until clock is low.
        dataline = parity & 0x1;              // Send parity
        while(!clk) Delay10TCYx(10*US_DELAY); // Wait until clock is high.
        while(clk) Delay10TCYx(10*US_DELAY);  // Wait until clock is low.
        dataline = 1;                         // Send stop bit.
        while(!clk) Delay10TCYx(10*US_DELAY); // Wait until clock goes high

        dataline_tris = 1;            // Stop driving data pin.
        while(dataline);              // Wait until acknowledgement bit.
        while(clk);                   // Wait until clock goes low.
        while(!clk || !dataline);  // Wait until mouse releases clk and dataline.
                                   // (both go high.)
}



unsigned char recvData(void) {
        unsigned char data = 0x0;
        unsigned char parity;
        char i;

        // Start bit
        while(clk) Delay10TCYx(10*US_DELAY);  // Wait until clock is low.
        while(!clk) Delay10TCYx(10*US_DELAY); // Wait until clock is high.
        // 8 bits of data
        for(i=0; i < 8; i--) {
                while(clk)
                        Delay10TCYx(10*US_DELAY); // Wait until clock line is low.
                data = data >> 1;             // Shift buffer.
                data += dataline * 0x80;      // Read next bit into buffer.
                parity += data & 0x1;         // Update parity.
                while(!clk) Delay10TCYx(10);  // Wait until clock is high.
        }
        // Parity bit.  Calculated but not checked.  Just compare
        // to value on dataline and handle accordingly.
        while(clk) Delay10TCYx(10*US_DELAY);  // Wait until clock is low.
        while(!clk) Delay10TCYx(10*US_DELAY); // Wait until clock is high.
        // Stop bit.  Not checked.
        while(clk) Delay10TCYx(10*US_DELAY);  // Wait until clock is low.
        while(!clk) Delay10TCYx(10*US_DELAY); // Wait until clock is high.
        return data;
}

void main(void) {
        unsigned char coorinfo;       // Mouse coordinate information
        unsigned short xpos=0,ypos=0; // Absolute x and y positions.
        TRISD = 0;                    // PORTD as output, to display x-axis.
        INTCON2bits.RBPU = 0;         // Enable weak pull-ups on PORTB
        clk_tris = 1;                 // Clock as input. (Open collector)
        dataline_tris = 1;            // Dataline as input. (Open collector)

        sendData(MS_RESET);           // Reset mouse.

        while(recvData() != MS_SUCCESS);      // Wait for self-test success.
        while(recvData() != MS_DEVICE_ID);    // Wait for mouse device id to be
                                              // sent.
        sendData(MS_DATA_REPORTING_ENABLED);  // Enable data reporting.
        while(recvData() != MS_ACKNOWLEDGEMENT); // Wait for acknowledgement
                                              // that mouse is in data reporting
                                              // mode.

        while(1) {
                coorinfo = recvData();
                // The tertiary operator is to determine if we should sign extend
                // the movement data.
```

```
                xpos += (COORINFO_XSIGN & coorinfo) ?
                                0xFFF0 | recvData() : recvData();
                ypos += (COORINFO_YSIGN & coorinfo) ?
                                0xFFF0 | recvData() : recvData();
                PORTD = xpos >> 8;     // Drive MSB of xpos on PORTD.
        }
}
```