SPI and the Printer Port

by Karl Lunt

A small DOS-box computer doesn't leave you many options for attaching external devices such as sensors. Basically, you can use the serial ports or the printer ports. The serial ports work best for terminal-type equipment such as modems or RF links, but the printer ports can really shine when you need to hook up several medium-speed I/O devices.

In this article, I'll show you how you can add up to eight devices to a single printer port, using little more than a ribbon cable and some C software.

The SPI

Traditionally, hobbyists have used a printer port to drive ICs such as an eight-bit latch. This simple hookup lets you control eight latch output lines just by writing a value to the parallel port. But if you need more I/O capability, such as more latch output lines or even a few input lines, you end up with a much more complex design. The older parallel ports only had a few input lines, and even the newer Expanded Parallel Port (EPP) units aren't all that easy to use if you need a lot of input lines.

This I/O limitation hasn't stopped hobbyists from developing some clever designs, of course. One of the better projects that I've seen wired into a parallel port was a device that could read and write GameBoy game cartridges. I discussed this project in detail in a past *Nuts & Volts* Amateur Robotics column; you can look through your stack of back issues, or do a web search for GameBoy and ReadPlus (the name of the reader).

But to get the most mileage out of your parallel port, consider using it to drive a synchronous serial bus such as Motorola's Serial Peripheral Interface, or SPI.

Hooking up an SPI device, such as a latch, requires only three lines plus ground; a total of five if you include +5 VDC to drive the device. More importantly, each SPI peripheral needs only one dedicated printer output line. This means you can add up to eight SPI devices to a printer port.

Since the SPI bus is bidirectional, you can use any mix of input, output, or bidirectional devices you need. This means you wouldn't have any problem driving, say, 32 channels of A/D, a couple of eight-bit latches, and a pair of frequency synthesizers off of a single parallel port.

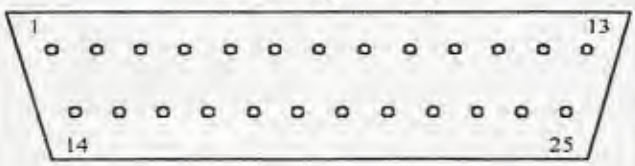
The SPI achieves this high capability because of the way it distributes data. All devices use the same serial input line, the same serial output line, and the same serial clock line; this last signal lets two devices synchronize bus operations. The device controlling the bus — known as the master — uses a dedicated line to each other device as a select line; bringing a select line low activates the device on that line. Only the selected device — known as a slave — will listen or respond to the host.

For example, if the master (your PC) wanted to exchange data with SPI device 3, it would bring printer output port line 3 low, leaving all other output lines high. Then, your PC could freely send commands serially over the common output line; only device 3 would process the commands. Similarly, your PC could receive data from the common input line, knowing that any data received would have been sent only by device 3.

The SPI exchanges data between two devices simultaneously. This means that each time the master sends a bit to the selected slave device, the master also reads a bit from the slave. The master device must provide the serial clock signal used by both devices for synchronizing this data exchange.

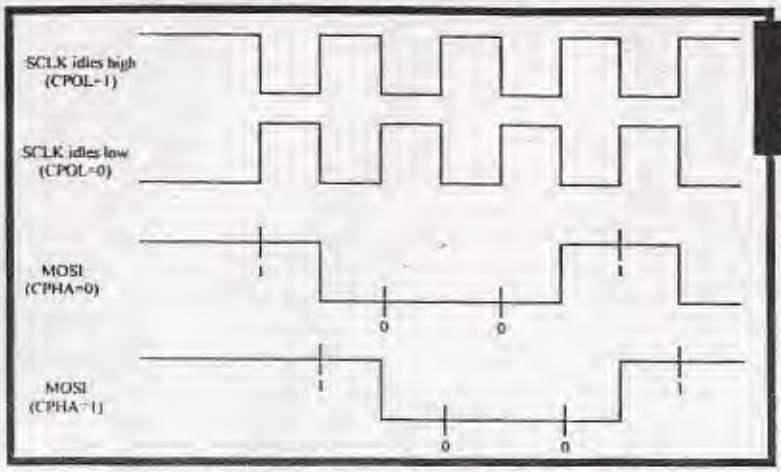
Putting this another way, no

DB-25 male (front view)



Pin	Signal	Direction (from PC)	Port & Bit
1	*Strobe	Output	Control, D0
2	D0	Output	Data, D0
3	D1	Output	Data, D1
4	D2	Output	Data, D2
5	D3	Output	Data, D3
6	D4	Output	Data, D4
7	D5	Output	Data, D5
8	D6	Output	Data, D6
9	D7	Output	Data, D7
10	*ACK	Input	Status, D6
11	*Busy	Input	Status, D7
12	PaperEmpty	Input	Status, D5
13	Select	Input	Status, D4
14	*AutoFeed	Output	Control, D1
15	*Error	Input	Status, D3
16	InitPrinter	Output	Control, D2
17	*SelectIn	Output	Control, D3
18-25	Ground		

THE SIGNALS AVAILABLE ON A PC PARALLEL PORT.



TIMING DIAGRAM SHOWING THE RELATIONSHIP BETWEEN CPOL and CPHA. MASTER SENDS BINARY 1001 TO SLAVE.

data are exchanged unless the master provides the necessary clocking pulses. This means that the master must always send something to the slave, even if the master just wants to read a byte of data.

Note that the SPI format doesn't prevent you from sending commands to more than one device at a time, should that be necessary. Just have the master pull all the necessary select lines low before sending any commands. However, this is a fairly rare occurrence. Generally, your software will deal with only one active device at a time.

There are some subtle timing requirements that you have to respect when using the SPI; refer to the sidebar for details. In a nutshell, though, the above paragraphs show that little is involved in moving data between a host device such as a PC and any of several different SPI devices on a bus. Anyone using the Motorola microcontrollers (MCUs), such as the 68hc11, likely will have already used or read about the SPI; it is built into almost all Motorola MCUs. Other chip makers, such as Atmel, also sell MCUs with built-in SPI.

The printer port

I've discussed the SPI in some detail, now I'll turn my attention to the printer port. In its simplest form, this is a multi-wire bidirectional port to the world; your PC software sees this port as three consecutive I/O registers. The first of these three registers is the data port, a byte-wide output port that your software can write to change the states of eight lines. The next higher register is the status port, a byte-wide input port that your software can read to sense the states of various signals from the printer. Finally, the control port is a byte-wide output port that your software can write to change the states of various signals to the printer.

Each printer port, known to your PC as LPT1 through LPT4, occupies three consecutive addresses beginning at any of three common I/O addresses, $3bc, $378, or $278. For example, if your PC assigns LPT1 to I/O address $3bc, then your software would use $3bc as the data port, $3bd as the status port, and $3be as the control port. The PC's BIOS

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <stdlib.h>

#ifndef TRUE
#define FALSE 0
#define TRUE 0xffff
#endif

unsigned int    dataport;
unsigned int    controlport;
unsigned int    statusport;
unsigned char   controlvalue;
unsigned char   datavalue;
unsigned int    waits;
unsigned char   buffer[128];
unsigned char   cpol;
unsigned char   cpha;
unsigned char   MSBFirst;
unsigned int    addata;                         // returned a/d value
float           fdata;
float           fullscale;

void SetMOSI(unsigned char value);
void ForceSCK(unsigned int v);
unsigned char ToggleSCK(unsigned char in);
void DeselectAll(unsigned char value);
void ToggleSelects(unsigned char mask);
void SetFormat(unsigned char phase, unsigned char polarity);
unsigned char ExchangeSPI(unsigned char value);

void main(int argc, char *argv[])
{
        unsigned int            n;
        unsigned char           c;

        dataport = 0x378;                       // assume LPT1
        statusport = dataport+1;
        controlport = dataport+2;
        waits = 0;
        MSBFirst = TRUE;
        fullscale = 4.096;                      // for max1204 A/D
        if (argc > 1) {
                for (n=1; n<argc; n++) {
                        if (*argv[n] == '/') {          // if this is an argument...
                                c = *(argv[n]+1);       // get argument char
                                switch (c) {            // based on argument char...
                                        case 'w':
                                        waits = atoi(argv[n]+2);  // get number of wait
states
                                        break;
                                }
                        }
                }
        }
        DeselectAll(0xff);                      // all selects are active-low
        SetFormat(0, 0);                        // format: cpol=0, cpha=0
        while (!kbhit()) {                      // until user hits a key...
                printf("\r");
                ToggleSelects(0x01);                    // select device 0
                ExchangeSPI(0x8e);                      // ch=1 0, uni, single, internal clk
                addata = ExchangeSPI(0x00);             // get msb of data
                                        addata <<= 8;   // move msb to top
                addata += ExchangeSPI(0x00);            // add lsb
                addata >>= 5;                           // finally, align data properly
                ToggleSelects(0x01);                    // deselect the device
                if (addata == 0x3ff) {                  // if all bits set...
                        printf("Overrange!");
                }
                else {
// no, got valid data
                        fdata = fullscale * ((float)addata / 1023.0);
                        printf("Data: %7.3f ", fdata);
                }
        }
        getch();                                        // clear kbd buffer
        printf("\n");
}

void ForceSCK(unsigned int v)
{
        unsigned char                   n;

        if (v) {                                        // if SCK should be high...
                controlvalue &= 0xfe;                   // *STROBE = 1 (active-low)
        }
        else {
                controlvalue |= 0x01;                   // *STROBE = 0 (active-low)
        }
        outportb(controlport, controlvalue);
        for (n=0; n<waits; n++) {
                outportb(controlport, controlvalue);
        }
}

unsigned char ToggleSCK(unsigned char in)
{
        unsigned char           v;
        unsigned char           n;

        if (!cpha) {                                    // if CPHA = 0...
                SetMOSI(in);                            // need to set MOSI now
                v = inportb(statusport);                // get value of MISO
        }
        outportb(controlport, controlvalue ^ 0x01);
        for (n=0; n<waits; n++) {
                outportb(controlport, controlvalue ^ 0x01);
        }
        if (cpha) {                                     // if CPHA = 1...
                SetMOSI(in);                            // need to set MOSI now
                v = inportb(statusport);                // get value of MISO
        }
        outportb(controlport, controlvalue);
        for (n=0; n<waits; n++) {
                outportb(controlport, controlvalue);
        }
        return  (v ^ 0x80) & 0x80;                      // invert *BUSY and strip other bits
}
```

records the assignment of each printer port to its I/O address in a table stored in RAM at address 0040:0008.

To look at the printer assignments of your PC, go to a DOS prompt and fire up the DOS debug program. When you get debug's prompt, enter the command:

d 0040:0008 L8

debug will respond by printing out the eight bytes of data stored at that address. The first pair of bytes gives the I/O address of LPT1, the second pair gives the I/O address of LPT2, etc. Note that since the PC uses an Intel-style processor, the I/O addresses are stored LSB first, so you will need to reverse the order of the two bytes in each address to determine the true I/O address.

Knowing how to use this table means your software can look up the I/O address associated with any desired LPT port, even if the BIOS or some other program switches port assignments at some time. This is important, because to control SPI devices using a printer port, your software must perform low-level accesses to the I/O registers.

Obviously, you want your software to bang the lines of the correct port, lest your laser printer suddenly go wacko and start spilling paper all over the place.

With most of the basics out of the way, we can start looking at the available lines on the printer port, to assign these lines to the necessary functions we need to support an SPI bus. Refer to the accompanying table of signals available on the printer port for details.

The most important line in the SPI bus is SCLK, which acts as the system clock signal. We will be bit-banging all of the SPI signals from the PC, so we could choose any line we want as our SCLK signal, but probably the easiest to remember is *Strobe. This signal appears on the printer connector as pin 1, and in the parallel port registers as bit 0 of the control port. Note the leading asterisk in the signal name, *Strobe. This indicates that this signal is active-low.

From the software viewpoint, you have to write this bit with the inverse of the desired signal. Thus, to pull *Strobe low, your software must set bit 0 of the control port high. Similarly, writing a 0 to bit 0 of the control port will bring the *Strobe output line high. This can take a little getting used to, but one function will be used for all manipulations of *Strobe, so you only have to get this concept right once, then you can forget about it for the rest of the program.

Next up, we need a signal to act as MOSI, the master device data output line. I chose *AutoFd, which is bit 1 of the control port and pin 14 of the printer connector, for this function. As with *Strobe, discussed previously, this is an active-low signal, so you have to write the inverse of the desired value whenever your software manipulates this bit.

Then we have MISO, the master device data input line. I chose *Busy for this signal because the documentation I was using for my design indicated that this line was active-high, meaning that my software wouldn't have to deal with the inversion discussed above. Unfortunately, the documentation was wrong; *Busy is active-low. I only discovered this after completing the software and seeing the inversion in my tests.

Rather than rewrite the software and mod the hardware at this point, I just added the inversion to the code and left *Busy as my MISO line. If you decide to rewrite my software, you might switch lines for MISO; PaperEmpty or Select might make better choices. For now, my code uses *Busy, which is bit 7 of the status port and pin 11 of the printer connector.

All that remains is assigning the SPI select lines. This software uses output lines D0 through D7 as the eight device select lines

Data Port (I/O address offset 0)
Output only from PC

D7	D6	D5	D4	D3	D2	D1	D0

Status Port (I/O address offset +1)
Input only to PC

*Busy	*ACK	PE	Select	*Error	*IRQ	---	---

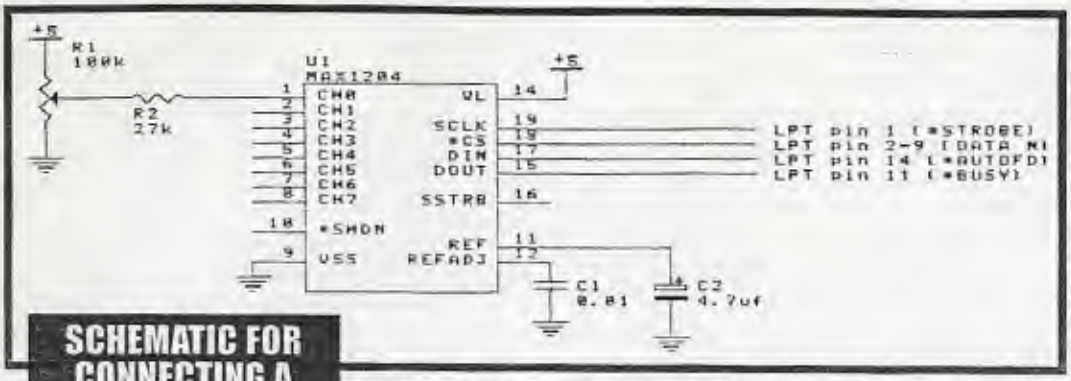
Control Port (I/O address offset +2)
Output only from PC

---	---	Dir	IRQEnb	*SelectIn	Init	*AutoFd	*Strobe

THE THREE REGISTERS OF A PC PARALLEL PORT.

```
/*
 * DeselectAll     bring all chip selects to deselected state
 *
 * This routine accepts an unsigned char that indicates all
 * devices are not selected.  After writing this value to the
 * LPT data port, this routine saves the value in a global
 * variable for later use.
 */

void  DeselectAll(unsigned char  value)
{
        unsigned char                   n;

        datavalue = value;
        outportb(dataport, datavalue);
        for (n=0; n<waits; n++) {                       // for all wait states...
                outportb(dataport, datavalue);
        }
}



/*
 * ToggleSelects     toggle one or more chip select lines
 *
 * This routine changes the chip select byte written to the
 * LPT data port.  Upon entry, the pattern in argument mask
 * is used to toggle bits in the current global value
 * datavalue.  This value is then sent to the LPT data port,
 * with waitstates.
 *
 * Note that you don't use this routine to turn bits on or
 * off, only to toggle them.  Thus, this routines serves as
 * both a device select and a device deselect routine.
 */

void  ToggleSelects(unsigned char  mask)
{
        unsigned char                   n;

        datavalue ^= mask;              // set the selected pattern
        outportb(dataport, datavalue);  // set it up
        for (n=0; n<waits; n++) {       // do the waiting
                outportb(dataport, datavalue);
        }
}


void  SetFormat(unsigned char  phase, unsigned char  polarity)
{
        cpol = polarity;                                // record clock idle state
        ForceSCK(cpol);                                 // set clock to proper idle state
        cpha = phase;                                   // record CPHA setting
}


void  SetMOSI(unsigned char  value)
{
        unsigned char                   n;

        if (!value) {                                   // if sending a 0...
                controlvalue |= 0x02;   // set MOSI to 0 (active-low)
        }
        else {                                          // nope, must be 1...
                controlvalue &= 0xfd;   // set MOSI = 1 (active-low
        }
        outportb(controlport, controlvalue);
        for (n=0; n<waits; n++) {               // wait it out...
                outportb(controlport, controlvalue);
        }
}


unsigned char  ExchangeSPI(unsigned char  value)
{
        unsigned char                   v;
unsigned char                   n;
        unsigned char                   indata;

        indata = 0;
        if (MSBFirst) {                                 // if sending MSB first...
                for (n=0; n<8; n++) {   // for all bits in byte...
                        v = ToggleSCK(value & 0x80);    // clock it
                        indata <<= 1;                   // move new data left one bit
                        if (v)  indata += 1;            // add the bit we just read
                        value <<= 1;                    // now move out byte left
                }
        }
        else {
                for (n=0; n<8; n++) {   // no, sending LSB first...
                        v = ToggleSCK(value & 0x01);    // clock it
                        indata >>= 1;                   // move new data right one bit
                        if (v)  indata += 0x80;         // add the bit we just read
                        value >>= 1;                    // now move out byte right
                }
        }
        return indata;
}
```

C PROGRAM



SCHEMATIC FOR CONNECTING A MAX1204 A/D CONVERTER USING THE PC'S PARALLEL PORT

supported by this project. This makes it easy to remember which device goes to which connector pin. If you need even more SPI devices hooked to your port, you can write extra code that uses the last two control lines, *InitPrinter and *SelectIn, to add two more devices.

Now it's time to get into the software. Refer to the accompanying listing of my C program. This is not a finished piece of software, so I've cut a few corners. I encourage you to build on this program to make it better for your own application.

I compiled this code using Borland C/C++, version 4.52, as a DOS standard application using the large memory model. The only non-ANSI feature that I'm aware of in this program is the use of the outportb() library function, which writes a byte to the desired I/O port, but most compilers support a similar function. Refer to your compiler's manual for details.

The program's main() function begins by assigning default values to several important variables. The variable **dataport** holds the base address of the parallel port's I/O registers. The value written here, 0x378, serves as the I/O address of my system's LPT1. As you can see, I have not implemented the BIOS lookup scheme described above. Feel free to add the lookup yourself, if you choose. If not, at least use a utility such as Microsoft's MSD to locate the I/O address used by your target printer port, and change the value written to **dataport** accordingly.

The variable **waits** is vital to the proper operation of this program, and deserves study. Most SPI devices run at a top clock speed of one to two Mbits per second. The newer PCs, however, can pump data out the printer port at much higher rates than the SPI devices can handle. In order to slow the newer machines down so the SPI devices can keep up, I've added a wait-state feature. **waits** holds the number of wait-states to insert in any SPI bus operation.

On my little 486 DOS-box, which I use for data collection, this program can only change the SPI SCLK line at about 500 KHz, well within reach of most SPI devices. The default value of **waits**, which is 0 in this code, is good enough for my little system. You can change the value of **waits** on the command line by using the **/wxxx** option, where **xxx** is the number of wait-states you want to insert.

Adding wait-states gets a little tricky. You can't use a simple counting loop, as some compilers will optimize out such loops, and the newer machines run too fast to make such loops meaningful.

To insert wait-states, my code simply repeats the most recent I/O operation **waits** times. I/O operations are timed independently of the PC's CPU, so each wait-state will actually occur, and will be of a known duration.

For an example of how the wait-states are inserted, look at the code in ForceSCK(). It is an instructive exercise to hook an oscilloscope to your printer port's *Strobe line, then run this program with different values for **waits** and watch the effect of the wait-states on the SCLK pulse length.

I've included the routine SetFormat() so you can quickly change the active SPI format. You might have devices hooked to your printer port that require different SPI polarities or phases, and this routine lets your software adjust the system's format before beginning a transfer. I've also added routines for controlling the eight printer data lines, used as SPI device select lines.

The function DeselectAll() allows you to write a value to the data port that turns off all SPI devices. Since your design might use a mix of active-high and active-low SPI devices, DeselectAll() allows you to pass a value that constitutes all devices off. Another routine, ToggleSelects(), lets your software change the state of particular SPI device select lines. Note that you don't specify whether the line goes high or low, only that it changes. Your software can blend usage of DeselectAll() and ToggleSelects() to control the SPI devices without having to know what state any device select line should be in.

Taking the bus for a SPIn

All of this software doesn't mean beans until you actually hook up a device. You have a bewildering assortment of SPI devices available to you. Perhaps the most commonly used SPI device is the 74hc595 serial-in/parallel-out octal latch. I've done a couple of articles on hooking this device to the SPI; see some of my earlier *Nuts & Volts* Amateur Robotics columns for details on using this chip to drive, for example, a liquid-crystal display (LCD).

But hooking up yet another '595 seemed so boring, considering the number of other possible choices. So I plugged into Maxim's web site at **www.maxim-ic.com** and started looking around for cheap A/Ds. After a few minutes, I settled on the MAX1204, an eight-channel 10-bit serial A/D that needs a single +5 VDC supply to operate.

Maxim has built some nice features into this chip. One feature I like in particular allows you to configure it as either eight single-ended analog inputs, or four differential inputs. You can even mix and match, should you need, say, five single-ended and one differential.

You can order a couple of these devices directly from Maxim, via their samples desk. The device is also available from Digi-key for about $4.50 each.

Inside the SPI

Motorola's Serial Peripheral Interface (SPI) uses three bus wires and additional chip-select lines to implement a synchronous bus that runs at speeds in excess of one megabit per second. Widely used in the embedded control industry, the SPI makes a good choice for moving high volumes of data across short distances (less than one foot) or among several different devices. The following paragraphs describe the various signals used in the SPI bus.

SCLK (or SCK) is the main clock signal for synchronizing data transfers between the master device and the selected slave device. This signal is provided by the master device. SCLK may idle either high or low. To generate a clock pulse, the master device momentarily brings SCLK to the active state, then returns it to the idle state. Each such clock pulse constitutes a single bit time, used to synchronize the exchange of one data bit.

MOSI (master-out-slave-in) is the SPI output line from the master device. At the correct point in each clock pulse, the master device outputs a level on MOSI corresponding to the bit value to send to the slave device. MOSI is connected to the input lines on all slave devices on the bus. The selected slave device will sample the value of MOSI at the correct point in each clock pulse to determine the value of the data bit sent by the master device.

MISO (master-in-slave-out) is the SPI input line to the master device. MISO is connected to the output lines of all slave devices on the bus. At the correct point in each clock pulse, the selected slave device outputs a level on MISO corresponding to the bit value to send to the master device. The master will sample the value of MISO at the correct point in each clock pulse to determine the value of the data bit sent by the slave device.

***CS** is the chip-select line used by the master to select a slave device. Generally, the master must provide a single chip-select line for every slave on the SPI bus, though some bus configurations can use one chip-select line to control multiple slave devices. By convention, SPI slave devices use an active-low chip-select line, though some SPI devices, such as the Dallas DS1305 real-time clock, use an active-high select line. The master leaves all chip-select lines in their inactive states until a data exchange is required. At that time, the master drives the proper chip-select line to its active state, selecting that slave device. After exchanging one or more bytes of data with the slave device, the master returns that select line to its inactive state, deselecting the slave device.

The timing for data exchange on the SPI bus depends on the SCLK signal and the agreed-upon format between the master and slave devices. The two devices may use any of four different timing formats, based on the idle state of SCLK (either high or low) and which edge of SCLK (either leading or trailing) marks the presence of valid data. Motorola refers to these two criteria as CPOL and CPHA. CPOL, or clock polarity, is 0 if SCLK idles low or 1 if SCLK idles high. CPHA, or clock phase, is 0 if data are valid on the leading edge of SCLK or 1 if data are valid on the trailing edge. Thus, an SPI bus that uses the format CPOL=0 and CPHA=0 relies on SCLK idling low, with data valid whenever SCLK changes from low to high.

As you can see from the accompanying schematic, the wiring is dirt simple; you get eight channels of A/D for little more than a couple of caps and a socket. I've added a trimpot (R1) and current-limiting resistor (R2) to channel 0 so I can dabble a little.

Note that the *CS line — pin 18 — connects to only one of the Dn lines on the printer port connector; if you want to use this schematic with the software listed here, hook *CS to printer connector pin 2 (D0). Also make sure you hook at least one of the ground lines on the printer connector (pins 18 through 25) to the ground line in the schematic. You will also need to supply a source of +5 VDC for this MAX1204; this can be a wall-wart or a set of batteries with a suitable voltage regulator.

The physical layout of this circuit is not critical; you could make up a little printed circuit board (PCB), or just use one of the RadioShack experimenter's boards. I chose to build my circuit on one of those white plastic prototyping blocks, offered by a number of mail-order vendors.

The tricky bit involves hooking wires to the printer connector. I opted to start with a 18-inch long 26-pin ribbon cable, available surplus nearly anywhere for a buck or so; mine even had a female 26-pin IDC connector on one end.

I stripped back the 26th wire from the opposite end of the cable, then pressed a male IDC DB-25 connector onto that end. This gave me a ribbon cable of suitable length, with one end I could plug into the LPT port of my PC and another end that I could plug into a 26-pin male dual-row header.

All that remained was coming up with a dual-row header that I could plug into my prototyping block. I started with a 26-pin wirewrap header gleaned from my junk box. Working carefully with a set of needle-nose pliers, I bent each of the long 26 pins to the proper shape. When complete, I had widened the gap between the two rows of long pins so the header would straddle the wide channel down the center of my prototyping block. When I plug the header into the block, each pin is isolated from the others.

Now I can plug the 26-pin female connector on the end of my ribbon cable into the prototyping block and complete my wiring. Note that such modified dual-row headers are also available from several mail-order houses, if you don't feel like taking the time to construct your own.

With the wiring complete, I just needed to add some code to my program that is specific to the MAX1204 A/D. See the code in routine main() for details. After some setup, all of the code for reading and processing the MAX1204 is handled by the large while-loop at the bottom of main().

To take a reading, my software first toggles select line 0, then uses ExchangeSPI() to send a read command to the MAX1204. The command sent, 0x8e, takes a reading from channel 0 in unipolar, single-ended mode, using the MAX1204's internal clock.

To collect the data from the MAX1204 after it finishes reading, my software must send two bytes of 0 and save the responses. This is done with the two successive calls to ExchangeSPI() and the manipulation of variable **addata**.

Finally, my code examines the value saved in **addata**. If the variable holds 0x3ff (all bits set), the code assumes an overrange and prints out an error message. If the value returned is valid, my code converts it into a floating-point number, scaled to a maximum of +4.096 VDC, and displays the result. This loop of read, collect, and display continues until the user presses a key to halt the program.

That's a wrap

As you can see, hooking SPI devices to the PC's printer port requires very little hardware and only a moderate amount of software. The program given here should get you well on your way.

Originally, I had intended to do a single, general-purpose program that could handle everything, but the different configurations of SPI devices are simply too great. In the end, I chose to do a collection of functions that you could use to accommodate nearly any SPI device. You could also transcribe this code to any of the various dialects of PC BASIC, should you be more comfortable with that language.

The SPI bus is a natural for grafting onto the PC's parallel port. The wide variety of devices, the ability of the bus to handle many different devices easily, and the simplicity of the software involved add up to a potent combination. Give this technique a try on your next data collection or robot control project. I think you'll like the results.

Much of my information on the line-printer port came from a superb web site maintained by Peter H. Anderson, a professor in the department of Electrical Engineering at Morgan State University.

He and his students have developed many different printer-port projects, and their web-site is loaded with terrific information and project designs. You can even buy books and circuit boards for several projects.

Check this page out at **www.et.nmsu.edu/~etti/fall96/computer/printer/printer.html**. **NV**