This 2-part series first appeared in *Circuit Cellar Ink* magazine.

Part 1

Using Serial EEPROMs

If you have a project that needs a modest amount of nonvolatile, read/write memory, serial EEPROM may be the answer. These tiny and inexpensive devices are especially useful when you need to minimize the number of I/O lines, cost, or physical size.

Probably the most common use for serial EEfigROMs is to store various types of user data - settings for remote-control devices, phone numbers, security codes, or anything that used to be set with DIP switches. Other uses include storing error codes and other diagnostic information, usage records (times, dates, counts), and instrument readings.

In some cases, serial EEPROMs can even store program code. Parallax's Basic Stamp and similar products use serial EEPROMs to store user programs in the form of BASIC-language tokens.

This article is a guide to choosing and using serial EEPROMs. I'll compare the three major interface types: Microwire, SPI, and I²C, and the pluses and minuses of using each. Next time, I'll show how to program and read all three types from a PC's standard parallel port.

The Basics

Serial EEPROMs use a synchronous interface - both the EEPROM and the chip that controls it use a common clock, and clock transitions signal when to send and read each bit. For example, a sending device may write each bit on the rising edge of the clock, and the receiving device reads the bit when it detects the clock's falling edge.

Although some other synchronous serial chips require minimum clock frequencies, the clock for serial EEPROMs can be as slow as needed, and the clock signal doesn't have to be symmetrical. The controlling device can toggle the clock at its convenience, up to the maximum speed. There's no need for a fixed timebase

In contrast, in an asynchronous link, as in most RS-232 communications, both ends of the link must agree on a frequency, and each end provides its own timebase. When the receiving device detects a Start bit, it uses its own clock to determine when to read each of the following bits. If the two clocks vary a lot, the receiving end will misread the data.

Serial EEPROMs typically have just eight pins: power and ground, one or two data/address lines, and a clock input, plus up to three other control signals. Unlike parallel EEPROMs, which add pins as the number of address and data lines grows, a serial EEPROM's physical size doesn't have to increase with capacity. The only alternative that uses less space is when you're already using a microcontroller that has EEPROM or nonvolatile RAM on-chip. But this isn't always an option, or you may need more storage than the microcontroller has.

Capacities begin at 128 bytes. As with other memory chips, the maximum capacities available have increased over time: Microchip's 24LC64 is an example of an 8-kilobyte chip.

The EEPROMs use CMOS technology, so they consume very little power, with currents as low as a few microamps in standby mode and a milliamp when active.

The synchronous interfaces aren't intended for use over long distances (for that, use RS-232 or RS-485), but cables may be as long as 4 meters in some cases, or longer if you add stronger drivers and buffers.

Depending on the device, the maximum clock speed for accessing serial EEPROMs can be over 2 Megahertz. But because it takes eight clock cycles to transfer a byte, and the master also has to send instructions and addresses, the maximum rate of data transfer is no more than around 4 microseconds per byte.

Write operations actually take much longer, because the EEPROM needs several milliseconds to program a byte into its memory array. During this time, the master can't read or write to the chip, but it can go on to other tasks that don't involve the EEPROM.

With use, EEPROMs eventually lose their ability to store data, so they're not suited for applications where the data changes constantly. These days, most are rated for a minimum of 10 million erase/write cycles, which is fine for data that changes occasionally, or even every few minutes. (For unlimited read/write cycles, use battery-backed RAM.)

Besides EEPROMs, other components with synchronous serial interfaces include A/D and D/A converters, I/O expanders, clock/calendars, and display interfaces. Multiple devices can connect to one set of lines, with each chip having its own Chip-Select line or firmware address.

There are three major types of interfaces for serial EEPROMs: Microwire, SPI, and I^2C. These give you plenty of choices, but it also means that a single interface and protocol won't work with everything. The different types vary in speed, number of signal lines, and other details.

To see how the different interfaces compare, I'll describe a 4-kilobit EEPROM of each type. Table 1 summarizes the major features, and Figure 1 shows the pinout of each. All EEPROMs with the same interface behave in a

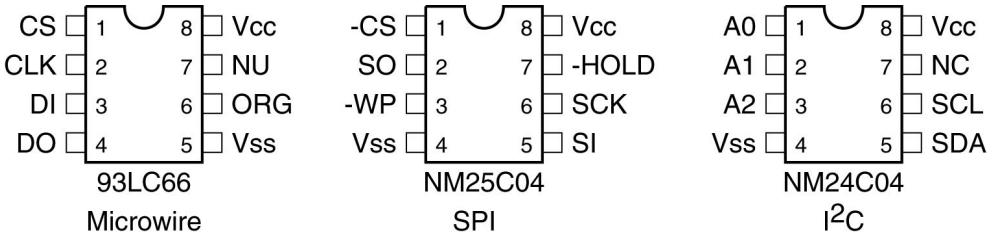

Figure 1. Pinouts for three 512-byte serial EEPROMs. The SOIC version of the 93LC66 is available in this and an alternate pinout. On the NM24C03, pin 7 is Write Protect. (NU = not used, NC = no connection)

similar way, though they may vary in the number of address bits and other details, such as whether or not there is a write-protect pin. To be sure, always read the data sheet for the chip you're using!

Microwire

Microwire is the oldest of the three interfaces. National Semiconductor introduced it, and other manufacturers now support it as well. National's COP888 is an example of a microcontroller with a Microwire interface built-in. Though it's commonly called a 3-wire interface, a complete link actually needs four signal lines plus a common ground.

Microchip's 93LC66 is a 4-kilobit serial EEPROM with a Microwire interface. It has two data pins: DI (data in) and DO (data out), a clock input (CLK), and a chip-select (CS).

Additional inputs are for memory configuration (ORG), which determines whether data format is 8 or 16 bits, and program enable (PE), which must be high to program the chip. Setting ORG high saves time because you can program and read two bytes with one instruction.

The EEPROM understands seven instructions: Erase/Write Enable and Disable, Write, Read, Erase, Erase All (sets all bits to 1), and Write All (writes one byte to all locations).

Table 1. Comparison of Microwire, SPI, and I²C serial EEPROMs

Interface	Microwire	SPI	I²C
Example device (512-byte capacity)	93LC66	NM25C04	NM24C04
Source	Microchip	National Semi	National Semi
Minimum number of interface (excluding GND)	4	4	2
Data width (bits)	8 or 16	8	8
Maximum clock speed (Mhz)	2	2.1	0.4
Write (busy) time (millisecs., max)	10	5	10
Maximum number of bytes programmed in one operation	2	4	16
Writes bit on (clock state)	rising edge	rising edge	low level
Reads bit on (clock state)	rising edge	falling edge	low level
Output low current (min.)	2.1mA@0.4V	1.6mA@0.4V	3mA@0.4V
Output high current (min.)	0.4mA@2.4V	0.8mA@Vcc-0.8V	(open collector)
Chip-select method	hardware	hardware	software
Write-protect method	software	hardware & software	none ('24C03 has hardware write-protect for upper half)

Figure 2 shows the timing for byte read and write operations. Each instruction must begin with a Start condition, which occurs when CS and DI are both high on CLK's rising edge. This occurs naturally when an instruction is written, because all of the instructions begin with 1. The master must bring CS low after each instruction, except sequential reads. When CS is high, the EEPROM is in Standby mode, ignoring all communications until it detects a new Start condition.

To write to, or program, the EEPROM, the master must first write an Erase/Write Enable instruction to DI, followed by a Write instruction, the

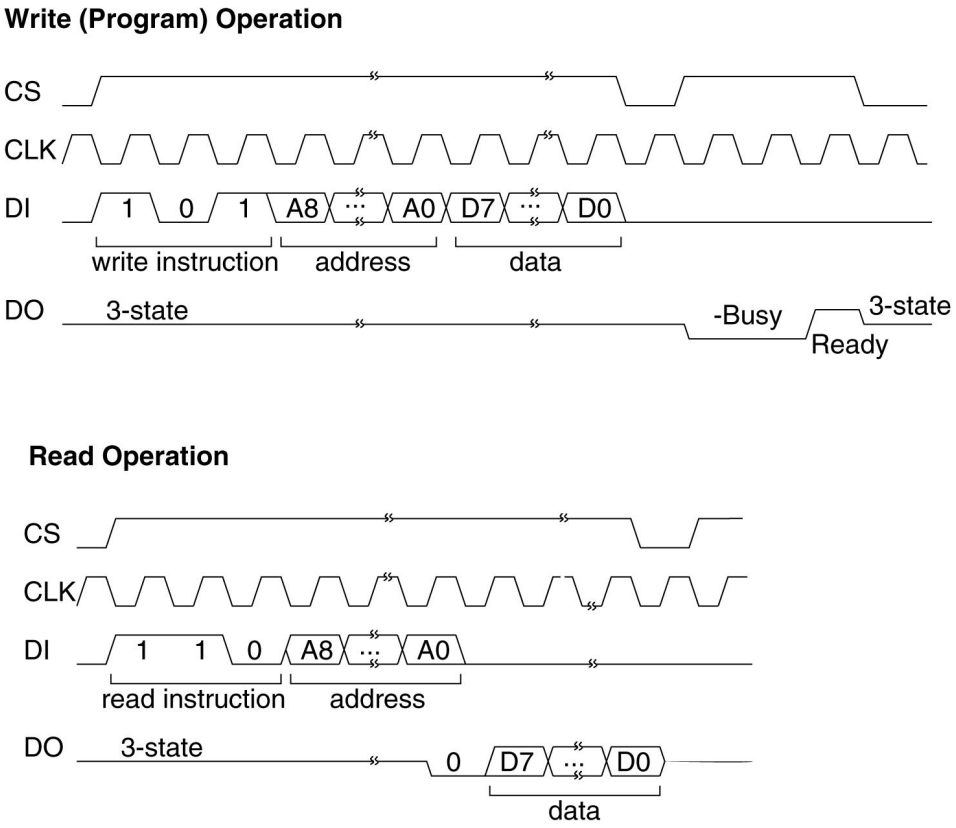

Figure 2. Byte read and write operations on a Microwire 93LC66 EEPROM, configured for 8-bit organization. CLK's rising edges latch inputs at DI and clock data out at DO.

master writes bits on CLK's falling edge, and the EEPROM latches each bit on the next rising edge.

After sending the final data bit in a programming operation, the master must bring CS low before the next rising edge of CLK. This causes the EEPROM to begin its internal programming cycle. (Some devices, such as Microchip's 93C76, don't require CS to go low here; instead, they begin programming on CLK's rising edge after D0.) The programming is self-timed; it requires no clock cycles. If CS returns high before the pro-

gramming cycle completes, D0 will indicate Ready/-Busy status. CS must then go low again to complete the write operation.

The master needs to send the Erase/Write Enable instruction just once per programming session. The device remains write-enabled until it receives an Erase/Write Disable instruction or power is removed.

To read from the EEPROM, the master writes a Read instruction to DI, followed by the address to read. When the EEPROM receives the final address bit, it writes a "dummy zero" to D0, then writes the requested data on CLK's rising edges.

If CS remains high after a Read operation, additional clock transitions will cause the chip to continue to output data at sequential addresses. If CS goes low, the next read operation must begin with the Read instruction and an address.

A 2-line interface is sometimes possible by connecting DO and DI. (An isolation resistor between the lines is recommended.) The EEPROM's DO output is high impedance except when sending data or busy status. In a 2-line interface, the master's DI output must also be high impedance when DO is active. However, there is a brief bus conflict in Read operations, when the EEPROM outputs the dummy zero on receiving the final address bit, and the master's driver must be strong enough to pull the combined DO/DI line high when this occurs and the address bit is 1.

SPI

The second interface type, SPI (Serial Peripheral Interface), originated at Motorola, which includes an SPI interface on its 68HC11 and other microcontrollers. SPI is much like Microwire, though the signal names, polarities, and other details vary.

Like Microwire, SPI is often referred to as a 3-wire interface, though a read/write interface actually requires two data lines, a clock, a chip select, and a common ground.

The signal names differ on the master and slave devices. The data line MOSI (master out, slave in) on the master connects to SI on the slave, and MISO (master in, slave out) on the master connects to SO on the slave. The clock is called SPICK at the master and SCK at the slave. A Master may also have four outputs (SS0 through SS3) that each connect to a chip-select (-CS) on up to four slave devices.

As with Microwire, SPI EEPROMs write bits on the clock's rising edge, but unlike Microwire, they latch input bits on the falling edge. (The SPI protocol actually allows two different clock polarities, but the EEPROMs support only clock polarity = 0.)

An SPI device also has a choice of two phases. The EEPROMs support only clock phase 1, which normally means that the chip-select can remain low if the interface has only one slave device. On the EEPROMs, however, -CS still has to toggle to reset the serial logic between instructions, so you can't just tie it low. Notice also that the polarity of -CS (active low) is opposite from Microwire's convention.

National's NM25C04 is a 4-kilobit EEPROM with an SPI interface. In addition to the four lines mentioned above, the chip has two other inputs. -WP (write protect) must be high to program the device. And for interfaces with multiple slaves, the -Hold input enables the master to pause in the middle of a transfer in order to do something more urgent on the SPI link. The EEPROM ignores all activity on the SPI bus until -Hold returns high and both devices pick up where they left off.

The EEPROM understands six instructions: Set and Reset the Write Enable Latch, Read and Write to the Status Register, and Read and Write to the Memory Array.

The chip has several levels of write protection, which you can use to virtually guarantee that there will be no inadvertent writes to the device. If -WP is low, no changes to the data are allowed. If -WP is high, two nonvolatile bits in the chip's Status Register can block writes to all or a portion of the device. And finally, if -WP is high, before you can write to the Status Register or the portion of memory enabled in the Status Register, the EEPROM must receive a Set Write Enable Latch instruction.

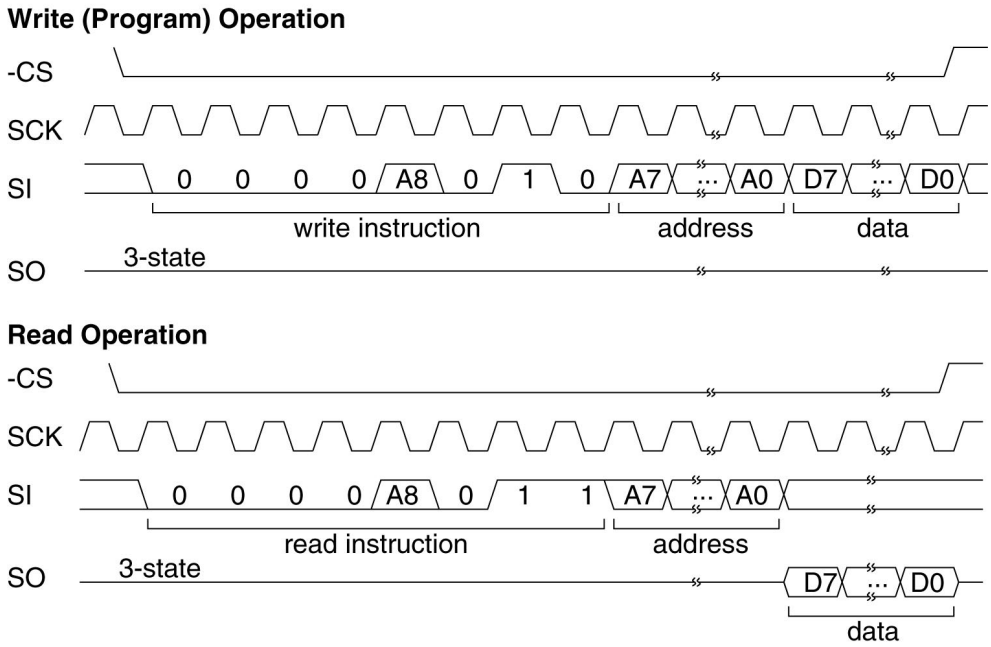

Figure 3. The NM25C04's SPI interface is similar to Microwire's. SCK's falling edges latch inputs at SI, while rising edges clock data out on SO.

Figure 3 shows the timing for byte reads and writes for the '25C04. To write to the EEPROM, the master writes a Set Write Enable Latch instruction to SI, followed by a Write instruction (which contains address bit A8), then the lower eight address bits, then the data to write. The master may send up to four data bytes for sequential addresses in one operation. After clocking the final data bit, with SCK low, CS must go high to begin programming the byte into the EEPROM.

While the EEPROM is programming the data, the master can read the EEPROM's Status register. When bit 0 of the Status Register is 0, the EEPROM has finished programming, and the next write operation can begin. The chip is write-protected after each programming operation, so each write must begin with a Set Write Enable Latch instruction.

To read the EEPROM, the master sends a Read instruction, which contains bit A8 of the address to read, then bits A7-A0. The EEPROM responds with the data bits in sequence on SO. As with Microwire, additional clocks will cause the EEPROM to send additional data bytes in sequence.

For larger capacities, rather than embedding address bits in instructions, the master sends a 16-bit address.

I²C

I²C (Inter-Integrated Circuit Bus) is the final interface type. This one originated with Philips/Signetics, whose 8XC528 (8051 family) is an example of a microcontroller with an I²C interface built-in.

The I²C interface requires just two signal lines, plus a common ground. Serial Data/Address (SDA) is a bidirectional line that requires open-collector or open-drain outputs. Serial Clock (SCL) is the clock. Instead of a chip-select line, the master sends a slave address on SDA. Figure 4 shows the timing for I²C transfers.

An I²C bus can have up to about 40 devices, with the limit determined by a maximum bus capicitance of 400 pF. Each device on the bus can have an address of up to 7 bits.

The open-collector/open-drain interface means that any logic-low output will pull SDA low. A device releases the SDA line by writing 1 to its output.

Unlike Microwire and SPI, which are edge-sensitive, I²C is level-sensitive. Data and address bits on SDA may change only while SCL is low, and the receiving device reads bits after SCL goes high.

There are two occasions when SDA changes state while SCL is high. A Start condition signals the beginning of an operation, and occurs when the master brings SDA low with SCL high. A Stop condition signals the end of an operation, and occurs when SDA goes high with SCL high.

Unlike Microwire and SPI, which offer no feedback except the success or failure of an entire read or write operation, I²C devices send Acknowledge

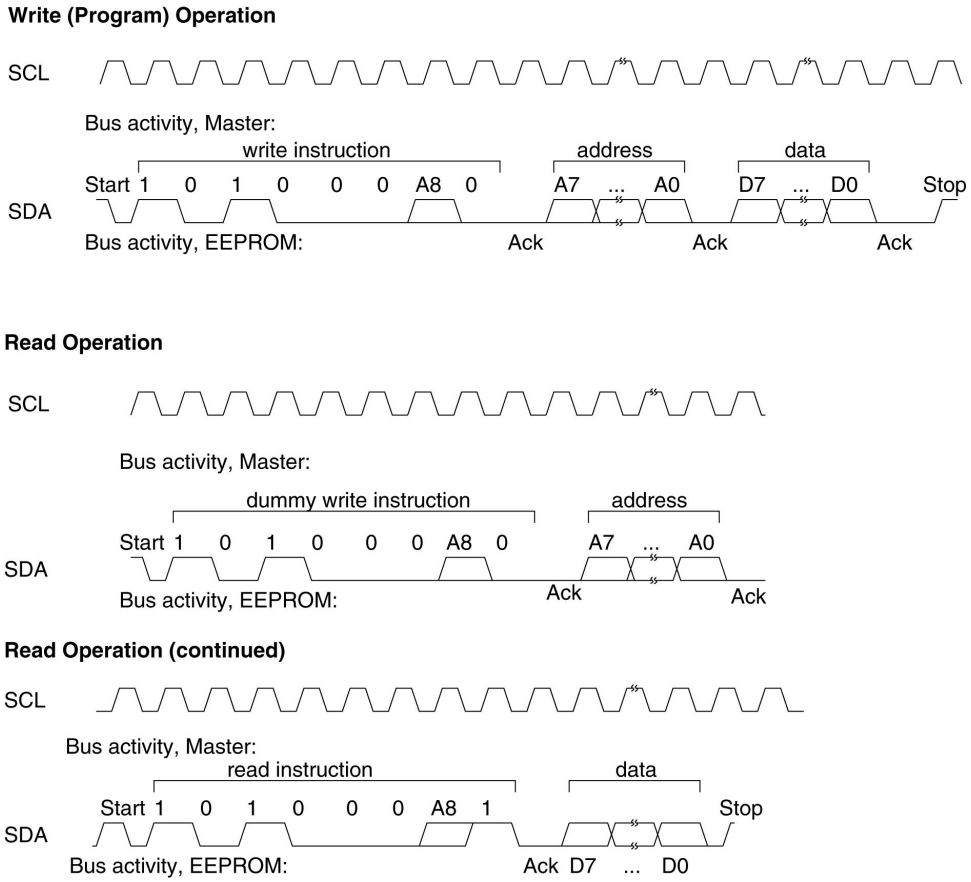

Figure 4. The NM24C04's I²C interface uses a single bidirectional signal line.

signals after receiving eight bits. After most transmissions, whether an instruction, address, or data byte, during the ninth clock cycle, the transmitting device releases SDA and the receiving device pulls SDA low to acknowledge that it received the bits. If the master doesn't see the acknowledge, it knows that something isn't right. The receiving device releases SDA on SCL's next falling edge.

An I²C bus can have multiple masters. If more than one master tries to control the bus at once, an arbitration protocol defined by the I²C standard determines which one wins.

National Semiconductor's NM24C04 is a 4-kilobit EEPROM with an I²C interface. SDA, SCL, power, and ground use four of the eight pins. Three other pins are seldom-used page-address inputs (A0, A1, A2). These allow multiple low-density EEPROMs on a single interface. For example, a link could have eight 256-byte EEPROMs, each with a different hard-wired page address. On the '24C04, A0 is unused and A1 and A2 can select any of up to four 512-byte devices. But it's simpler and cheaper to use a single 2-kilobyte chip.

On some EEPROMs (NM24C03), the remaining pin is a hardware write-protect for the upper half of the memory array.

To write a byte to the EEPROM, the master issues a Start condition, then writes an 8-bit slave address. The address consists of a 4-bit type identifier (1010 for EEPROMs), followed by the selected page (000 or 001), and a request to read (1) or write to (0) the device.

The type identifier is defined by the I²C standard. The page address is a way of specifying address bit A8 (0 or 1). The other two bits in the page address are unused unless there are multiple EEPROMs. In the clock cycle following the slave address, the EEPROM pulls SDA low to acknowledge.

The master then sends an 8-bit address, and the EEPROM acknowledges. (When necessary for larger capacities, the master can send two address bytes.) The master sends the byte to write, waits for an acknowledge, then issues a stop condition. The EEPROM then programs the data into its memory array. When programming is complete, the EEPROM acknowledges.

To write to up to 16 bytes to sequential addresses, instead of issuing a Stop condition after the first data byte, the master may continue to send data bytes, waiting for an acknowledge after each. After sending all of the bytes, the master issues the Stop condition, and the EEPROM programs the bytes and acknowledges. Some I²C EEPROMs (Microchip's 24LC04) can pro-

gram all 16 bytes in parallel, for much faster programming. On others, you can write 16 bytes in sequence, but the chip programs them one at a time.

To read a byte, the master begins as if doing a write operation, sending a slave address followed by a byte address. When the EEPROM acknowledges the byte address, the master issues a new Start condition, followed by the slave address with the final bit set to 1 (read). The slave acknowledges, then writes the data to SDA. On receiving the data, the master doesn't acknowledge, but instead issues a Stop condition.

To read sequential addresses, the master acknowledges receiving the data byte, and the EEPROM responds by sending the next byte in sequence. The EEPROM will continue to send bytes until it receives a Stop condition instead of an Acknowledge.

Decisions

Which EEPROM type should you use? Sometimes the choice is obvious, for example, if you're using a microcontroller with a built-in interface, or you want to use a specific A/D converter in the link.

All three types are easily available and inexpensive. Digi-key has 512-byte devices of each type for under $3 in single quantities.

I^2C is best if you have just two signal lines to spare, or if you have a cabled interface (I^2C has the strongest drivers).

If you want a clock faster than 400 kilohertz, use Microwire or SPI.

For more on using serial EEPROMs, I recommend browsing the manufacturers' pages on the web, especially these sites:

National Semiconductor
http://www.national.com/design/
Many application notes on Microwire.

Motorola Semiconductor
http://www.mcu.motsps.com/mc.html
Microcontroller references contain SPI documentation.

Microchip Technology
http://www.microchip2.com/appnotes/appnotes.htm
Notes on SPI use.

Philips Semiconductor
http://www.semiconductors.philips.com/
I^2C information.

Next time, I'll present the design of an EEPROM programmer that runs from a PC's parallel port, with Visual-Basic program code.

Jan Axelson is the author of *USB Complete* (available 11/99), *Serial Port Complete, Parallel Port Complete* and *The Microcontroller Idea Book*. You can reach her by email at jan@lvr.com, or via her web site at http://www.lvr.com.

Part 2

Programming Serial EEPROMs from a PC's Parallel Port

Serial EEPROMs are popular devices for storing user settings, measurement data, and other changeable, nonvolatile information. In a typical use, a microcontroller acts as a master that controls communications with the EEPROM. Many microcontrollers have built-in ports that are compatible with the EEPROMs' synchronous serial interfaces.

But a microcontroller isn't the only way to communicate with these devices. A PC's parallel printer port is an inexpensive and flexible interface that you can use for programming and reading serial EEPROMs.

With a parallel-port interface, the only added hardware required is some generic buffers and drivers and a cable. On the programming side, you can use any language that enables you to read and write to ports. Software can emulate all three of the popular synchronous interfaces, and the code is easily modified when needed to handle device variations.

This article describes circuits and Visual-Basic code for a parallel-port programmer for serial EEPROMs of each of three interface types: Microwire, SPI, and I^2C. The article builds on the information in my previous article about serial EEPROMs.

I tested the programmer with a 4-kilobit device of each type. With minimal modifications, you can use the interfaces and program code to communicate with just about any serial EEPROM. You can also use the code and circuits as a starting point for communicating with other chips that use similar interfaces.

About the Programmer

One drawback of using the parallel port is that it's software-intensive. If you use a microcontroller or expansion card with a built-in interface, the hardware takes care of most of the details of generating the clock and chip-select signals at appropriate times, dividing each byte to send into bits, and combining received bits back into bytes. If you use the standard parallel port (or any generic I/O port), you have to do all of this in software.

There are a variety of ways to connect the EEPROMs to the parallel port. Although many ports now include features for high-speed, bidirectional communications, every PC's parallel port can emulate the original port's design, with all bits under software control. The signals are eight Data outputs (bits 0-7) at the port's base address, five Status inputs (bits 3 - 7) at base address + 1, and four Control outputs (bits 0 - 3) at base address + 2.

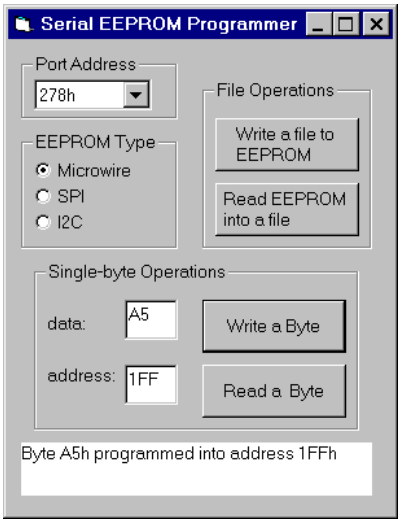

Figure 1. The PC's parallel port provides a simple interface for communicating with serial EEPROMs.

I tried several hardware configurations for the programmer. Figure 1 shows the interface I settled on because it was straightforward and usable on any PC's port. Each EEPROM uses two or three of the Data outputs and one Status input. The Control port and two Status bits are unused. The eight ground returns in a standard 25-wire parallel cable all connect to signal ground in the EEPROM circuit.

Circuit construction and cable design aren't critical. I used an ordinary 10-foot ribbon cable, with the circuits on a solderless breadboard.

Microwire and SPI Interfaces

The interfaces to Microwire and SPI EEPROMs are similar. Each line uses one of a 74LS244's buffer/drivers.

The DO and SO outputs aren't intended for driving long cables; they're guaranteed to sink at most a couple of milliamps at 0.4V, so I added a driver

for each. In the other direction, again because of the cable length, I used '244 buffers to add some hysteresis at the EEPROM's inputs.

The pullup at the Microwire's DO output is required only if you might try to read the chip's Busy status after a programming operation has completed. You don't need it if you read the status during a programming cycle, or if you skip the Busy check entirely and just wait 10 milliseconds to access the chip after programming it.

The choice of buffer/drivers isn't critical; a 74HCT244 or similar will work as well. If your cable is very short, you may get by without any added buffers or drivers at all.

The ORG, HOLD, and Write-Protect inputs are all tied inactive. If you want to control these in software, you can connect them to the unused Control-port outputs.

The I²C Interface

The I²C interface differs because it uses a single, bidirectional data line (SDA). The EEPROM's SDA output is open-drain, and the master's SDA output must be open-drain or open-collector as well, so that either the master or the EEPROM can pull the SDA line low.

The I²C standard also specifies that SCL's output should be open-drain, to enable multiple masters to take turns providing the clock signal. With a single master, you can use other output types.

One way to connect the SDA line to a PC's parallel port is to use one of the bits of the parallel port's Control port. On the original PC and most of its descendants, the Control bits are open-collector, with 4.7K pullups typical (On many newer ports, for faster switching, the Control outputs switch to push-pull type when the port is configured for a high-speed (EPP or ECP) mode, but revert to open-drain when emulating the original port.)

But there are a few ports that don't have the open-collector/open-drain outputs, so I decided not to assume that these would be available. Instead, as

with the other interfaces, I used a Data output and a Status input. SDA's input buffer is a 7407 open-collector driver with a 4.7K pullup.

When the PC is writing data, addresses, or instructions to the EEPROM, SDA's output is off and SDA follows bit D6. During Read operations, D6 must be high, to enable SDA to control S5.

You can use just about any LSTTL or HCTMOS driver/buffers at S5 and D7 (SCL). I used the 7407s only because I was already using the chip and had the extra drivers. And of course, if you don't use open-collector/open-drain devices for these, you don't need the pullups.

If you decide to use one of the parallel port's Control bits to communicate with SDA, be aware that bits 0, 1, and 3 in the parallel port's Control register read the inverse of the logic state at the connector. If you use one of these bits, remember to use inverting buffer/drivers or invert the bit in software.

Using the Programmer

Figure 2 shows the user screen for the programmer application. I created the software with Visual Basic 4, and it will load and run in either the 16-bit or 32-bit edition of VB4, under Windows 3.1 or Windows 95. (It won't run under NT, which requires a kernel-mode driver for port accesses. If you have a driver for port I/O under NT, you can modify this program's routines for use with it.)

The complete program code is available on my web site at http://www.lvr.com and on the Circuit Cellar Ink BBS and web site.

A drop-down list box allows users to select any of the three most common base addresses for parallel ports (378h, 278h, 3BCh). You can use other addresses by adding them to the list box's code.

The application will program and read individual bytes, or files. To program a byte, you select the EEPROM type, enter the byte and an address in the text boxes, and click on the corresponding Program command button.

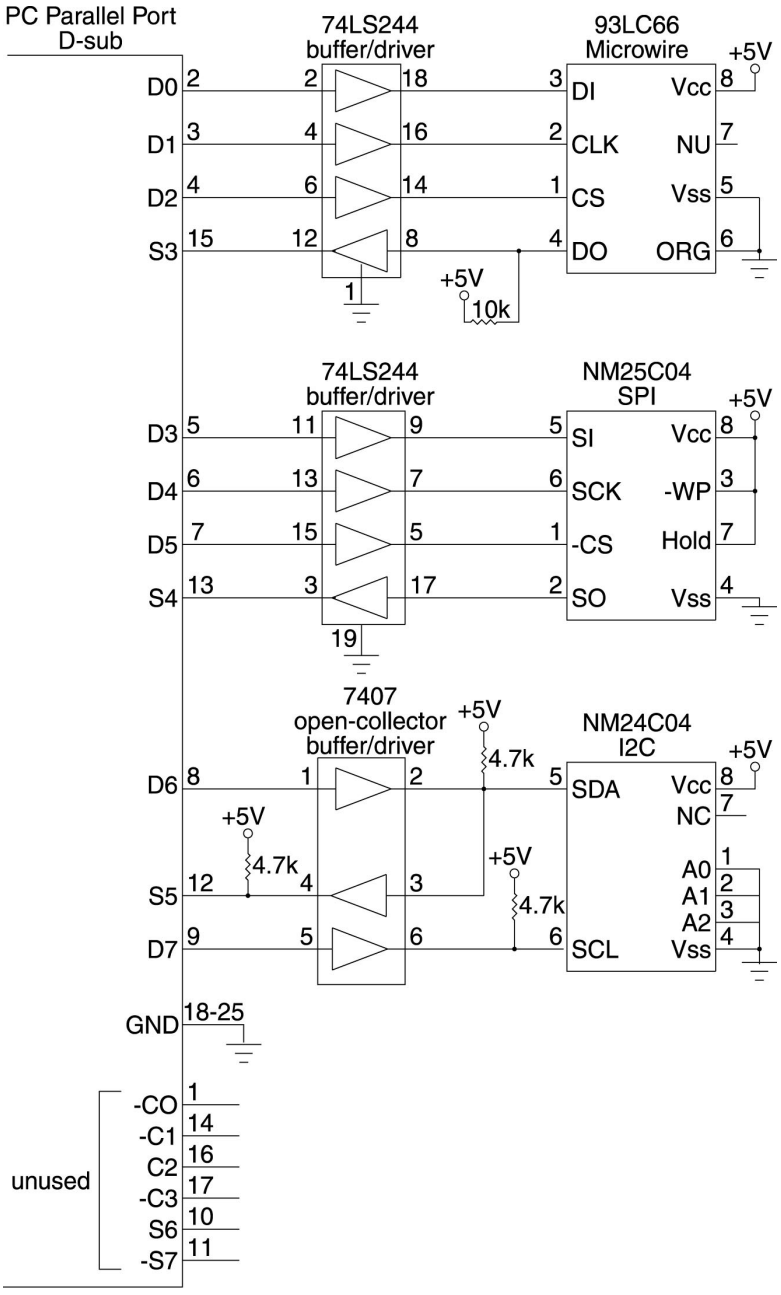

Figure 2. With this Visual-Basic program and Figure 1's circuit, you can read and program Microwire, SPI, and I²C serial EEPROMs.

After programming, the software reads the EEPROM's status, waiting for it to return a "not busy." The text box at the bottom of the window tells you when the programming operation is completed, or that the programming operation has timed out without receiving the expected response from the EEPROM. If the file to program is longer than 512 bytes, the software programs the first 512 bytes.

To read a byte from an EEPROM, you select the EEPROM type, enter an address in the text box, and click on the corresponding Read command button. The program reads the requested byte and displays it in the text box. If it's an I^2C interface and the software doesn't receive the expected Acknowledge signals, the read operation times out and displays a message. The Microwire and SPI interfaces have no timeout for read operations, because the EEPROMs don't send any acknowledges.

To program the contents of a file into an EEPROM, or to write the contents of an EEPROM into a file, use the File Program and File Read command buttons. Each brings up a Common Dialog box that enables you to select a file to read or write to.

A completed programming operation doesn't guarantee success. To verify, read the byte(s) back and compare with the original.

Accessing Ports

The first challenge to accessing the parallel port in Visual Basic is that VB doesn't include BASIC's usual Inp and Out for accessing I/O ports. A solution is to use an "Inpout" DLL (dynamic linked library) that adds these routines to VB. The DLL reads and writes directly to the selected port.

The EEPROM programmer will use either of two DLLs, depending on whether the program is running under the 16-bit or 32-bit edition of VB. As with all DLLs, the DLL itself must be present on the system running the program, and the program must declare the routines it calls.

The syntax for using the DLL's Inp and Out is the same as in QuickBasic:

```
ByteRead = Inp(PortAddress)
Out PortAddress, ByteToWrite
```

```
VB also allows this alternate syntax for Out:
Call Out (PortAddress, ByteToWrite)
```

Another option for accessing ports under Windows 3.x or 95 is to use a VxD (virtual device driver), which enables an application to block port accesses from unauthorized sources. A VxD has other benefits as well, such as the ability to respond more quickly and to use system features like DMA.

But both Windows 3.x and 95 allow direct port reads and writes as long as another driver hasn't blocked access to the port. If other applications have no need to use the port, and if you don't need to use a VxD for other reasons, direct I/O is a quick and inexpensive solution.

Inside the Software

The program itself consists of many short routines. One set handles the user interface, including reading the option buttons and text boxes and responding to button clicks. Other routines handle tasks common to all three EEPROM types, such as extracting a bit from a byte and displaying timeout messages. And for each EEPROM type, a set of routines sends instructions, addresses, and data, and reads data in the required format.

I designed the program to work with an example 512-byte EEPROM of each type. With modifications, you can use it with EEPROMs of other capacities, or make other changes a specific device may require.

Listing 1 shows the routines for writing and reading one bit with each type of EEPROM. For each EEPROM signal, I defined a constant equal to the signal's bit number at the parallel port. If you want to use different bit assignments, just change the constants to match.

To keep the software as flexible and easy to understand as possible, I designed the code so that individual bits in a byte can be changed without having to keep track of the states of all of the others. A form variable (DataOut) holds the last value written to the Data port, and a BitWrite function sets or clears a selected bit in a byte. To toggle a bit at the Data port, the code first sets or clears the desired bit in DataOut, then writes the result to the port.

A challenge in getting this software working was that the serial links don't provide much in the way of feedback. The only way to know if a byte programmed successfully is to write the byte and read it back. If it doesn't verify, there's no way to know if the problem was in the programming or read operation. A single missing or extra clock pulse, or a mistake in an instruction or address means that the intended operation won't complete.

I²C does send an Acknowledge to let the master know when the EEPROM has received something. Even here, the master may read (logic-low) Acks when none have been sent (if the circuits aren't powered up, for example). So only a successful verify, not the lack of an error message, indicates a success.

The good news is that with all three interfaces, you can toggle the clock as slowly as you want. To troubleshoot, I single-stepped through the routines and verified that each signal was doing what I expected at each step.

Listing 2 shows routines for writing Read instructions to each EEPROM type. Again, nothing is automatic. The software has to provide all of the clock transitions and write each bit of the instructions and data at appropriate times.

Modifications and Enhancements

Although the program is functional as it stands, chances are that you'll want to make changes and enhancements, such as the ability to use other EEPROM sizes or the addition of instructions such as Erase All. Other enhancements might include saving of program settings such as default EEPROM types and file directories, and more robust error-checking.

Jan Axelson is the author of *USB Complete* (available 11/99), *Serial Port Cmplete, Parallel Port Complete* and *The Microcontroller Idea Book*. You can reach her by email at jan@lvr.com, or via her web site at http://www.lvr.com.

Listing 1. Each serial-EEPROM type uses a different protocol for sending and receiving bits. When using the PC's parallel port to communicate with the EEPROM, the software has to provide the clock transitions at appropriate times. These routines write bits to the EEPROMs.

```
'Bit numbers of output signals at the parallel port's Data port.

'MW (Microwire):
Const DIn = 0
Const CLK = 1
Const CS = 2

'SPI:
Const SI = 3
Const SCK = 4
Const nCS = 5

'I2C (no hardware chip-select):
Const SDAout = 6
Const SCL = 7

'Bit numbers of input signals at the parallel port's Status port.

'MW:
Const DOut = 3

'SPI:
Const SO = 4

'I2C:
Const SDAIn = 5

Private Sub I2CWriteBit(BitToWrite%)
'Write the bit with SCL=0,
'then bring SCL high to latch the bit into the EEPROM.

DataToWrite = fncBitWrite(DataToWrite, SCL, 0)
Out OutputPortAddress, DataToWrite

DataToWrite = fncBitWrite(DataToWrite, SDAIn, BitToWrite)
Out OutputPortAddress, DataToWrite
```

```
DataToWrite = fncBitWrite(DataToWrite, SCL, 1)
Out OutputPortAddress, DataToWrite

End Sub

Private Sub MWWriteBit(BitToWrite%)
‘Write the bit on CLK’s falling edge,
‘then bring CLK high to latch the data into the EEPROM.

DataToWrite = fncBitWrite(DataToWrite, DIn, BitToWrite)
DataToWrite = fncBitWrite(DataToWrite, CLK, 0)
Out OutputPortAddress, DataToWrite

DataToWrite = fncBitWrite(DataToWrite, CLK, 1)
Out OutputPortAddress, DataToWrite

End Sub

Private Sub SPIWriteBit(BitToWrite%)
‘Write the bit on SCK’s rising edge,
‘then bring SCK low to latch the data into the EEPROM.

DataToWrite = fncBitWrite(DataToWrite, SI, BitToWrite)
DataToWrite = fncBitWrite(DataToWrite, SCK, 1)
Out OutputPortAddress, DataToWrite

DataToWrite = fncBitWrite(DataToWrite, SCK, 0)
Out OutputPortAddress, DataToWrite

End Sub
```

Listing 2. Each EEPROM type responds to a small instruction set. The routines below send a Read instruction to the EEPROM. The instruction is followed by the address to read, and the EEPROM responds with the requested data. The I²C and SPI instructions include address bit 8, while the Microwire instruction sends all nine bits following the instruction.

```
Private Sub I2CSendReadInstruction(A8%)
'The read instruction consists of the device identifier (1010),
'two don't cares, address bit 8, and 1 (Read).

Call I2CIssueStartCondition
Call I2CWriteBit (1)
Call I2CWriteBit (0)
Call I2CWriteBit (1)
Call I2CWriteBit (0)
Call I2CWriteBit (0)
Call I2CWriteBit (0)
Call I2CWriteBit (A8)
Call I2CWriteBit (1)
Call I2CWaitForAck

End Sub

Private Sub MWSendReadInstruction()
'Sends the Start bit (1) and Read instruction (1,0):

Call MWWriteBit(1)
Call MWWriteBit(1)
Call MWWriteBit(0)

End Sub

Private Sub SPISendReadInstruction(A8%)
'Sends the Read Instruction:

Call SPIWriteBit(0)
Call SPIWriteBit(0)
Call SPIWriteBit(0)
Call SPIWriteBit(0)
Call SPIWriteBit(A8)
Call SPIWriteBit(0)
```

```
Call SPIWriteBit(1)
Call SPIWriteBit(1)

End Sub
```