Tiny AVR Serial Port Programmer

Now you can program AVR microcontrollers with your PC's serial port for less than $10. Bruce's general-purpose, ATtiny15L-based AVR programmer plugs into a laptop's serial port and draws its power from the port's modem control lines.

Atmel makes a series of wonderful inexpensive general-purpose eight-pin microcontrollers that go by the name of ATtiny. One part in particular, the ATtiny15L microcontroller, has become one of my favorites because of its rich set of features and the design flexibility it offers. (No, I don't work for Atmel!) Atmel offers a wide variety of inexpensive flash-memory-based microcontrollers that all share the same AVR instruction set, which Atmel describes as RISC-based and optimized for high-level languages such as C.

The micro-architecture of Atmel's AVR microcontrollers is in fact clock-efficient in comparison to other microcontrollers such as Microchip's PIC line and the typical 8051 varieties. Evidence of this fact is that an AVR microcontroller running with a 1-MHz processor clock delivers close to 1 MIPS. This is because most instructions take just one clock to execute, and no instructions take more than three clocks.

I have been working with AVR microcontrollers for several years now. My favorite is microcontroller is the popular AT90S8515. (This part was renamed the ATmega8515 when moved to a new silicon process, which doubled its maximum clock rate to 16 MHz.) One of the things that drew me to the AVR product line in the first place was its flash program memory. I was also interested in the fact that the parts can be programmed in-circuit with the SPI port, using only four signal wires. One-time programmable (OTP) parts aren't for me. I guess I have always been a trial-and-error programmer. When my firmware gets into the hardware, I need a lot of trials to get it right!

For several years now, my colleagues and I have distributed the source code for a open-source Windows/Linux parallel-port command-line programming tool (PPPT). It uses a dumb five-wire cable connected to a PC parallel port with no active circuitry. There are many programs like this available on the 'Net.

Like all AVR parallel port programmers, the PPPT needs direct access to the PC's parallel port hardware I/O registers in order to wiggle the SPI port signal lines. This wasn't a problem with Microsoft Windows 95/98. But under Linux and Windows NT/2000/XP, direct I/O port access requires special user privileges and a custom driver. When using Windows XP, a special driver that grants the privilege to do direct I/O must be installed, making any program that accesses the driver a potential security risk. When using Linux, the typical parallel port programmer needs root privilege to give program direct access to the parallel port's I/O address space.

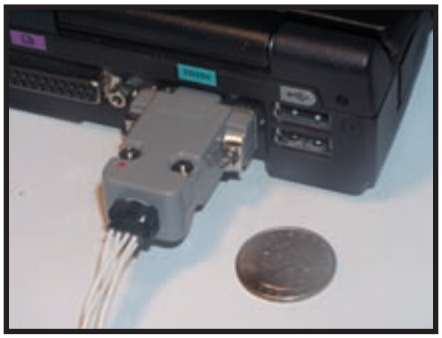

Photo 1—*My tiny AVR programmer plugs into a laptop's serial port. This diminutive device steals all of its operating power from the serial port's modem control lines.*

To make matters even worse, parallel ports have virtually disappeared from new laptops. The same applies to serial ports. Luckily, inexpensive USB-to-serial port dongles are available to add back the missing laptop serial ports that engineers in our line of work have come to depend on.

Given the above-mentioned problems, I decided to design an inexpensive, general-purpose, in-circuit programmer for AVR parts that doesn't require an unused PC parallel port and special operating system privileges. The ATtiny15L, coupled with a PC serial port, offered a perfect solution. The surface-mount version is so small and low in power that it can be placed in the hood of the DE9 female serial port connector and powered from a PC's RS-232 modem-control signals (e.g., RTS/CTS).

Photo 1 shows the tiny programmer plugged into a laptop PC's serial port. As you'll see, this device is something that's useful for other tasks as well. It's just a small matter of C programming!

ATtiny15L SOLUTION

The eight-pin ATtiny15L microcontroller is available in a conventional DIP package or a surface-mount SIOC package for less than $2 in small quantities. Because the part uses a factory-calibrated, stable, supply-voltage-independent, internal tunable RC oscillator for its 1.6-MHz processor clock, up to six of the eight pins can be used for I/O. If you want to program the part in-circuit without any

special voltages, the usable number of I/O pins drops to five, with the sixth I/O pin reserved for active-low chip reset.

Four of the I/O pins can be multiplexed into a built-in, 15,000 samples-per-second, 10-bit ADC, which offers a number of different programmable voltage reference options. Two of the ADC's input pins can be paired and used as a differential analog input with programmable gain. Also, two of the I/O pins on the part can be fed to an analog comparator.

The ATtiny15L microcontroller has two 8-bit timers, including one that can be used a pulse-width modulator (PWM) up to 150 kHz. The part has no UART hardware, so one of the challenges I faced was providing bidirectional serial port communication with a PC serial port (using true RS-232 levels) at high data rates (up to 115,200 bps).

The ATtiny15L operates from 2.7 to 5.5 V while using only a tiny amount of current. When running at 1.6 MHz with a 3-V supply, it consumes only 3 mA. In deep sleep mode, it draws less than 1 µA. In fact, depending on the RS-232 serial port in use, it's possible to run everything from current stolen from the serial port's modem flow-control signal lines.

The ATtiny chip has two kinds of nonvolatile storage, 1,024 bytes of flash program memory and 64 bytes of EEPROM. That works out to be only 512 AVR instruction words, so you must be careful (and efficient) when creating firmware. Although the ATtiny15L has the standard complement of 32 general-purpose AVR registers, there isn't any additional RAM memory besides the register set. There's a three-level hardware call-return stack, but there are no push or pop instructions for data. On the surface, this makes using the AVR version of the GNU C compiler (AVR-GCC) impossible. However, I've discovered a number of C coding tricks that enables the AVR-GCC to work just fine with AVR parts without RAM, like the ATtiny15L.

DESIGN GOALS

My goal was to build an inexpensive, general-purpose device that required only a standard PC serial port to program any AVR device supporting Atmel's in-circuit programming algorithm. I wanted it to fit in the serial port's DE9 hood, and I wanted to be able to power it from the serial port (i.e., no external power supply required). Unlike other low-cost AVR serial port programmers, I wanted mine to provide true RS-232 signal levels.

Of course, I needed a workable design within the constraints of the ATtiny15L's limit of five I/O pins and no RAM. Because I was using GNU's open-source AVR-GCC C-compiler, all the firmware needed to fit within the ATtiny15L's limit of just 512 instruction words.

I wanted the programmer to achieve AVR firmware download times comparable to the existing parallel port programmers it would be replacing. I also wanted it to be reprogrammable in-circuit and useful for other embedded applications, besides programming other AVR processors.

As you'll see, I achieved all of these goals and more. The system is useful not only as an AVR programmer, but you can also use it for many other functions. For instance, you can use it as a stand-alone, general-purpose, AVR-based controller (using an external power source) or a port-powered intelligent analog/digital data-logging device (if you attach it to a PC).

Because signals for all eight ATtiny pins are brought out to an external connector, there is almost no limit to what can be done within the basic capabilities of the ATtiny15L part. If you need more than 512 AVR instructions and extra SRAM, you can substitute in a more capable eight-pin ATtiny part.

PROGRAMMING ALGORITHM

Table 1 is a summary of the AVR serial programming instruction set that's common to many parts in the AVR product line, including the ATtiny15L. In the case of a part like the ATtiny15L, this is known as the low-voltage instruction set because all of the in-circuit programming is done using the same voltage levels as the processor supply voltage in the range of 2.7 to 5.5 V.

Some AVR parts offer an alternative in-circuit serial programming option

Instruction	Instruction format				Operation
	Byte 1	Byte 2	Byte 3	Byte 4	
Programming enable	1010 1100	0101 0011	xxxx xxxx	xxxx xxxx	Enable serial programming (while RESET is active)
Chip erase	1010 1100	100x xxxx	xxxx xxxx	xxxx xxxx	Chip erase flash memory and EEPROM arrays
Read program memory	0010 H000	xxxx xxxa	bbbb bbbb	oooo oooo	Read H (high or low) data o from program memory at word address a:b
Write program memory	0100 H000	xxxx xxxa	bbbb bbbb	iiii iiii	Write H (high or low) data i to program memory at word address a:b
Read EEPROM memory	1010 0000	xxxx xxxx	xxbb bbbb	oooo oooo	Read data o from EEPROM at address b
Write EEPROM memory	1100 0000	xxxx xxxx	xxbb bbbb	iiii iiii	Write data i to EEPROM at address b
Write lock bits	1010 1100	1111 1LL1	xxxx xxxx	xxxx xxxx	Write lock bits LL
Read lock bits	0101 1000	xxxx xxxx	xxxx xxxx	xxxx xLLx	Read lock bits LL
Read signature bytes	0011 0000	xxxx xxxx	0000 00bb	oooo oooo	Read signature byte o at address b
Write fuse bits	1010 1100	101x xxxx	xxxx xxxx	FFFF 11FF	Set fuse bits FFFFF
Read fuse bits	0101 0000	xxxx xxxx	xxxx xxxx	FFFF xxFF	Read fuse bits FFFFF
Read calibration byte	0011 1000	xxxx xxxx	0000 0000	oooo oooo	Read RC oscillator calibration byte o (for OSCCAL register)

Table 1—*These bit patterns are shifted into the ATtiny15L microcontroller's SPI port in order to reprogram the device in-circuit. Note that* a *is the address high bits.* b *is address low bits.* H *is the low byte (0) and high byte (1).* o *is data out.* i *is data in.* x *is don't care.*

that applies 12 V to the chip's RESET pin during programming. This mode of serial programming is advantageous because the RESET pin can be reprogrammed and used as a general-purpose I/O pin. However, I didn't use this programming mode because I didn't want to deal with the complexities of generating and switching a 12-V signal.

Note that many AVR parts now offer Page mode read/write instructions for program memory. Although this project supports this mode of device programming, I won't cover it here.

The program and EEPROM arrays are programmed using the AVR chip's SPI bus while the chip's RESET pin is held active low. The serial programming interface consists of SCK (input), MOSI (input), and MISO (output) pins. Figure 1 shows sample AVR serial programming waveforms. The Programming Enable instruction must be executed before program/erase instructions can be executed.

A given programming instruction is sent to the AVR by shifting 4 bytes, 1 bit at a time, into the part using SCK and MOSI. Each bit shifted in results in a bit being shifted out on the MISO pin. Therefore, 4 bytes are shifted out for each programming instruction that's executed. For most instructions, these 4 bytes are ignored, except for read instructions, in which case the last byte shifted out is the read data.

Given the operation of the programming instructions in Table 1, a serial port programmer must be able to execute an arbitrary 4-byte AVR programming instruction and then return the last data byte read back from the chip. The other basic operation needed is to execute the programming enable sequence, including cycling RESET and SCK as required, in order to place the ATtiny15L part in Programming mode. With these two basic serial port programmer operations, a PC program can be constructed to use the ATtiny-based serial programmer to download firmware to any AVR microcontroller using its SPI in-circuit programming port.

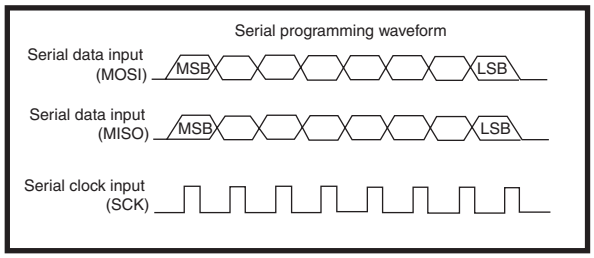

Figure 1—*ISP programming instructions are shifted into the SPI port's MOSI pin on the rising edge of SCK. Output data (on the MISO pin) is sampled on the falling edge of SCK.*

ALTERNATIVE DESIGNS

My programmer certainly isn't the only inexpensive serial port programmer for the AVR microcontroller product line. Information about constructing a PC serial port programmer using a 20-pin Atmel AT90S1200 microcontroller and a few transistors and miscellaneous passive components is described in Atmel's application note "AVR910: In-System Programming." As with my design, a PC program sends special commands over the PC's serial port to an AVR microcontroller. The microcontroller drives the SPI programming signals in order to reprogram a second AVR microcontroller.

In fact, Atmel's design involves circuitry similar to mine to generate bipolar voltage levels without the use of dedicated bipolar DC-to-DC converters. However, the positive RS-232 voltage level generated by the Atmel design can't be any greater than the AVR microcontroller's supply voltage. The positive RS-232 serial port levels technically should be higher than 9 V. This typically isn't a problem for modern PC serial ports, providing the AVR processor is powered from a 5-V source. However, if a 3.3-V supply (or even a 2.7-V supply) is needed to program a target AVR processor running at this lower supply voltage level, the Atmel design may not work at all, even with modern PC serial ports.

Because my design uses the serial port's modem control lines (DTR and RTS) to derive the positive RS-232 voltage level, the RS-232 level sent to the PC is independent of the AVR processor supply voltage, which can be derived from the same modem control lines. In other words, the design can be port powered, requiring no external power supply.

Another problem with the Atmel programmer's RS-232 level-shifting circuit is that it uses two microcontroller I/O pins for the RS-232 interface. With an eight-pin AVR part like the ATtiny15L, you don't have the luxury of using two pins for the serial port interface because you have a total of only five I/O pins to work with. The SPI programming port needs four pins: RESET, SCK, MISO, and MOSI. My solution uses only one bidirectional I/O pin to communicate in Half-Duplex mode over the serial port connected to the host PC.

It would be more of a challenge to fit Atmel's AT90S1200-based reference design inside the hood of a DE9 serial port connector because of its larger chip and required off-chip oscillator. The bottom line is that my ATtiny-based design is certainly smaller (and likely a little less expensive) than Atmel's reference design. Both designs cost less than $10 in parts. But, as you'll see, my design is flexible enough to have a number of other uses.

RS-232 LEVEL SHIFTING

A PC serial port uses RS-232 signal levels to communicate with external devices. RS-232 signal levels are inverted in comparison to typical digital signal levels. The Transmit Idle condition is signified by a continuous logic one (1), a negative voltage level between –4 and –12 V. The start of a character is signified by a logic 0, a positive voltage level between 4 and 12 V. (Note that levels in the range of ±25 V are technically possible, but few modern PCs use these extreme voltage levels.) A typical single-supply RS-232 level-translator chip such as the MAX202 puts out voltages in the ±9 V range.

After the start bit, the data bits are sent (typically 8 data bits in total) using this inverted logic. This is followed by at least 1 stop bit, a logic 1, or negative voltage level, which brings us back to the transmit idle state. The data bits are sent least-significant bit first. Because the RS-232 data signal

from the PC idles at the negative RS-232 voltage, you can use this same signal, along with some diode magic, to send negative voltage levels (logic 1) back to the PC.

The modem control signal lines from the PC—data terminal ready (DTR) and request to send (RTS)—are positive voltage levels (logic 0) when active. This is where I get my positive RS-232 voltage supply for sending logic 0 data back to the PC and, optionally, powering the regulator chip for the ATtiny15L's V_{CC} supply.

Figure 2 shows how it all comes together. I used a standard PC serial port to provide a half-duplex RS-232 communication path using a single ATtiny15L pin.

In Receive mode, the software UART in the ATtiny15L sets up pin 2 (PB4) as a digital input. RS-232 data sent over DE9 pin 3 (TXD) is read directly by the microcontroller on pin 2. Resistor R2 and diode array D2 limit the RS-232 input signal level sent to this logic-level pin to a safe range of 0 to V_{CC}.

At the same time, this same unipolar logic-level signal drives the gate of the N channel MOSFET QP1A. This inverts the logic-level signal and shifts it to V+, the larger positive voltage present on the DTR and RTS serial port modem control lines. The inverted signal is in turn applied to the gate of the P channel MOSFET QP1B, which again inverts the signal, either driving a positive voltage level onto DE9 pin 2 (RXD) or not. Pull-up resistor R5 ensures that the P-MOSFET QP1B is off whenever QP1A is off.

If the N-MOSFET QP1A isn't driving V+ onto RXD, then RXD is pulled to the same negative voltage present on TXD as negative current flows through R2 and the D1 diode array. This ensures that the PC receives true bipolar RS-232 signal levels on its serial port input pin 2 (RXD). D1 and R2 also serve to block positive voltages switched by the P-MOSFET QP1B from coupling back into the PC's serial output line TXD.

Because of double-inversion by the two transistors, serial data sent to the ATtiny15L is immediately echoed back to the PC without delay. This means that the PC receives a copy of every character that it sends to the programmer.

When it is time to transmit serial data from the ATtiny15L to the PC, pin 2 (PB4) is reconfigured to be an output pin by the software UART. At this point, the AVR I/O pin driver overrides the TXD signal path from the PC serial port to the programmer. The gate of QP1A is now under the software UART's control. Data follows the same path back to the PC as before, with transistors QP1A and QP1B serving as RS-232 level translators.

The source of positive voltage (V+) for the RS-232 driver QP1B is the PC serial port's RS-232 modem control lines DTR and RTS. The diode array D3 ensures that only positive voltages pass into the V+ storage capacitor C2. The low-dropout voltage regulator U1 converts this voltage into the supply voltage (V_{CC}) for the ATtiny15L microcontroller. Indicator LED D5 serves two purposes. It's a power-on indicator, and it guarantees a small current drain that prevents over-voltage buildup on the V_{CC} storage capacitor C1. Otherwise, the positive voltage path from TXD through the current-limiting resistor R2 and then through the V_{CC} voltage-limiting diode D2 onto C1 would tend to build up excess voltage on C1.

Note that there's a provision to power the ATtiny15L microcontroller via an external DC voltage source using the miniature power connector P2 and diode D4. This enables the programmer to be powered via an external power source in situations where the PC serial port modem's control signals don't supply enough current to run the ATtiny15L. Because the design's typical current requirements are less than 5 mA, most PC serial ports have plenty of current to spare, and an external power supply isn't required. Photo 2 shows the front and back of the assembled programmer.

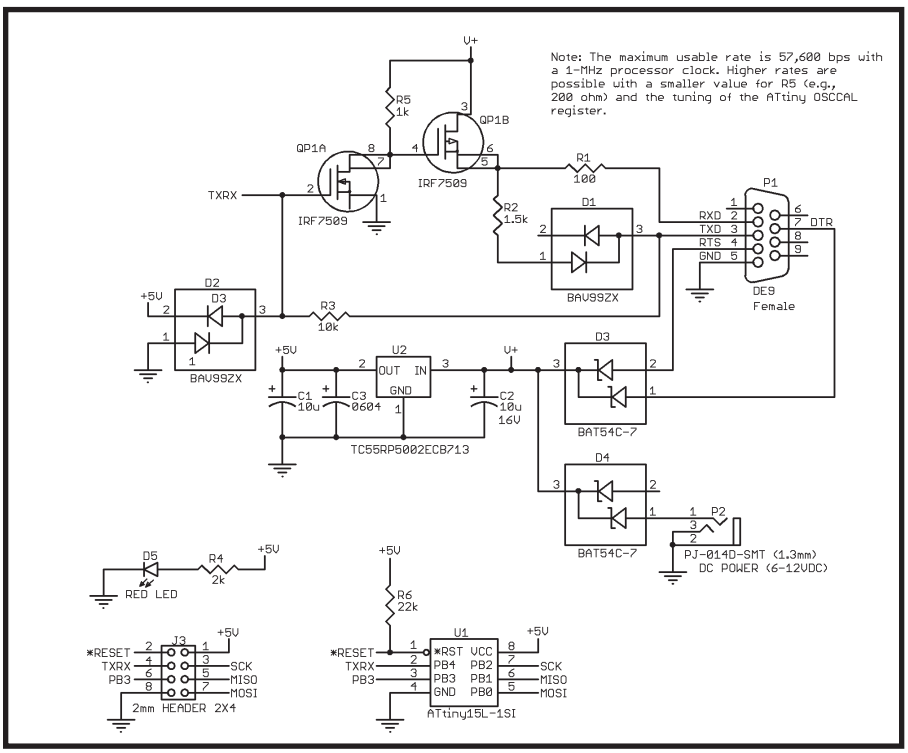

Figure 2—*My tiny AVR programmer steals its power from the serial port. The RS-232 interface needs only one of the ATtiny15L's eight pins. By using all surface-mount parts, everything easily fits inside the DE9 connector hood.*

TRANSISTORS GONE BAD

One of the purposes of this project is to provide an RS-232 serial interface that would run at the highest possible data rate. In the case of the ATtiny15L, which runs with a calibrated 1.6-MHz processor clock, the most code-efficient way to achieve precise UART timing is to use soft-

ware delay loops and not one of the 8-bit timers. Although the ATtiny15L's factory-calibrated internal RC oscillator makes for a stable, predictable source of timing for the software UART, I found that the prototype wouldn't run reliably at higher data rates (e.g., 57,600 and 115,200 bps).

The problem was diagnosed as being the fault of the circuit's transistors. The transistors' turn-off times were much too slow, even though the bit time at 115,200 bps is a whooping 8.7 µs. My original design used PNP and NPN switching transistors to provide the double-inversion and level shifting from unipolar logic levels to bipolar RS-232 levels. Apparently, the original transistors were capable of switching at higher than 100 MHz. As I discovered, this isn't true in a relatively low-current application such as this one.

Because the transistors were being driven into full saturation and were switching extremely low currents (less than 1 mA), their turn-off times were measured in microseconds! I was running with turn-off delays that were several of orders of magnitude slower than I had first expected. This was the result of the Miller effect. Note that this delay isn't symmetrical with respect to the two edges of a serial port data pulse. Only one edge (the falling edge of RXD) is delayed by as much as 5 µs. Therefore, at 115,200 bps, where bit times are only 9 µs, the serial data was distorted enough to completely garble the data received by the PC.

The improved circuit shown in Figure 2 uses P channel and N channel logic-level MOSFETs to achieve the same function as the earlier PNP/NPN transistor design. In fact, this design still exhibits significant FET turn-off delays on the order of 1 to 2 µs for QP1A and QP1B combined, depending on the values chosen for pull-up resistor R5 and pull-down resistor R2. This effect can be ignored at data rates of 38,400 bps and lower.

At higher data rates, a couple of no-operation instructions in the software UART code can be used to help balance the delay. However, this fix doesn't remove edge-timing distortion from the echoed characters sent by the PC to the programmer. The only fix in this case is to lower the value of R5. A value under 470 Ω clears up this problem at 115,200 bps. The trade-off is current consumption while the serial port is active versus the maximum serial port speed. Remember that you want to power everything with power stolen from the serial port's modem control lines. As a result, you need to switch as little current as possible.

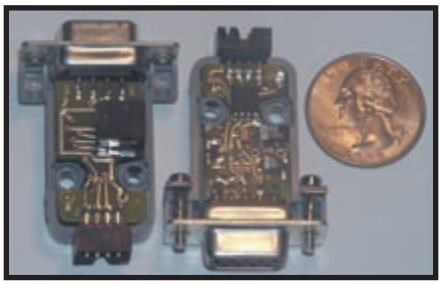

Photo 2—*When I say tiny, I mean tiny. Everything fits inside a DE9 hood.*

C WITHOUT RAM

After switching to C code from assembly language for firmware development, it's hard to go back. You still have to know the target processor's machine code to be an effective firmware developer for resource-limited microcontrollers like the ATtiny15L, but I find that being able to program in C makes me more productive and results in a much more maintainable product. Plus, with careful work and planning, there's no net loss of efficiency with respect to how much functionality you can stuff into the target microcontroller.

My C compiler of choice has always been GNU's open-source GCC compiler, which is supported in some form on almost every operating system. It targets virtually every instruction set in the universe, and it has generated good machine code every time I've used it. There is an AVR version of GCC with a full complement of support tools, including an assembler and a linker. For years, I've used AVR-GCC to develop firmware for Atmel AVR chips, using both Windows and Linux. The machine code it has generated has been especially good. However, when it was time to start working with the ATtiny15L, I was disappointed to discover that AVR-GCC didn't support the low-end Atmel microcontrollers like the ATtiny15L that have no RAM. What was I to do?

I have years of experience with the AT90S8515 microcontroller, which has only 512 bytes of RAM, so I knew I could write C code for the chip, which uses almost no RAM. In fact, the hard part about using AVR-GCC for AVR microcontrollers like the AT90S18515 is writing C code in a way that conserves the precious small amount of RAM.

The AVR-GCC generates AVR code that uses the 32 8-bit hardware registers for local variables. It passes parameters to and from subroutines and functions using the same registers. Therefore, it's reasonably easy to write complex C programs that use little or no RAM. The reward for doing this well is tightly coded machine code that tends to run fast. Like many firmware developers, I always generate a mixed listing from the compiler that shows the compiler's machine code interspersed with my C source code lines. This way I know exactly what the compiler is doing. By reviewing this output on a regular basis, I can learn what the compiler does well and what it has problems with.

My experiences with the code generator from the AVR-GCC taught me that, in fact, the compiler should work with a microcontroller without RAM. The key is making sure that AVR instructions that reference RAM never make it into the target microcontroller's executable image. I solved this problem by putting together a simple Perl script that post-processes the C compiler's output and checks for illegal instructions.

I use the GNU make program (supplied with the AVR-GCC) to cause the assembler to automatically generate a mixed listing and then check it for bad instructions using the Perl script. If problems are detected, the Perl script outputs an error message, which includes the C source line that generated the problem instruction along with the bad assembly language instructions. The make file logic then deletes the resultant object file and aborts the entire process. Therefore,

bad code never makes it to the linker.

With these checks in place, it's now a simple matter of coding in such a way as to completely avoid any use of RAM. Also, remember that the ATtiny15L has only a three-deep call-return stack.

Let's consider some tricks for programming AVRs without RAM. These tricks tend to yield fast and efficient machine code, no matter which AVR processor your C program targets.

Constant data that would otherwise need to be placed in RAM is stored in program memory using some special compiler attributes. It's then retrieved with the load program memory (LPM) instruction when needed.

The main program is declared naked to inhibit the generation of subroutine entry/exit code. The project is structured to include no more than two (or three) levels of subroutine calls (depending on whether or not interrupts are enabled). Without this declaration, registers used for local variables would be pushed and popped from the (nonexistent) stack.

Any storage needed in the interrupt handler is globally declared both in the interrupt handler and in the main program and subroutines:

```
register unsigned char foo
  asm("r2");
```

Note that `r2` specifies one of the AVR hardware registers. In this case, the register selected in this way must not be one of the registers reserved by the AVR-GCC for special uses, such as parameter passing or temporary storage.

Use a special in-line assembly macro in situations where the AVR-GCC uses a RAM location and an `lds` instruction to load constant data into one of the AVR lower registers (r0–r15). For example:

```
foo = 1; // Will likely generate a
  "lds"
foo = BYTE(1);// Uses "ldi/mov"
  instead
```

SPPT Options	Description
-d	Set Debug mode
-r	Reset only
-a	Leave programmer SPI signals active on exit
-comx	Look for program cable on serial port COMx
-b rate	Set serial port data rate (default=38400)
-ce	Chip erase
-en	Enable chip (de-assert RESET)
-lp file	Load program memory from file
-le file	Load EEPROM from file
-sp file	Save program memory to file
-se file	Save EEPROM to file
-dp n1 n2	Display program memory in range n1–n2
-de n1 n2	Display EEPROM in range n1–n2
-dd	Display device codes
-ap addr w	Alter program memory at addr with 16-bit word w
-ae addr b	Alter EEPROM at addr with 8-bit byte b
-wp	Write protect program/EEPROM memory
-rp	Read/write protect program memory/EEPROM

Table 2— ***Bruce: Please write a 2- or 3-sentence caption.***

The in-line assembly macro `BYTE()` causes the compiler to first load the constant into one of the high registers (r16–r31) using an `ldi` instruction. It then moves the data into the variable's assigned low register.

Keep all subroutines and functions simple enough that they don't require any local variable storage other than those registers reserved by the AVR-GCC for parameter passing and temporary storage. Basically, this includes 12 of the 16 upper 8-bit AVR registers (r18–r27 and r30–r31). Plus, avoid non-leaf C subroutines and functions that include any state. A routine with no local storage meets this requirement, which is needed to avoid push and pop instructions needed to save and restore this state before and after calling another routine.

Use static in-line routines and in-line assembly macros liberally. Such code is expanded in-line by the compiler. It completely eliminates subroutine calls, which serves to avoid some of the aforementioned restrictions. In fact, it's more efficient for routines that are called from only one place.

Of course, your other option is to use an ATtiny device that has RAM. That would be way too easy for me! Besides, not having RAM tends to make the microcontroller cheaper, and in the embedded world, and cheaper is always better.

PC SOFTWARE

The simple console-mode Windows C program serial port programming tool (SPPT) controls my ATtiny15L-based programmer. You can compile the program with either Microsoft Visual C++ or Borland's C compiler (BCC). Unlike its earlier parallel port cousin PPPT, this program doesn't need any special operating system privileges to run. The only hardware resource it requires is one of the PC's standard RS-232 serial ports.

Like all AVR-GCC tools, the program SPPT is a command-line program that takes its direction from parameters supplied with the program's run string. For example, the following command will erase and then download the foo.hex file (containing an AVR program memory image) into a target AVR processor:

```
sppt -ce -lp foo.hex -en
```

The program's command-line options are shown in Table 2.

The SPPT program does all of its work using the ATtiny15L microcontroller as a kind of proxy. The PC program tells the ATtiny15L what to do by sending a few simple commands over the PC serial port. As you know, these command characters are echoed back because of the way the half-duplex serial interface works. The ATtiny15L then executes the commands. For many of the commands, one or more response bytes are sent back to the PC. When the command is complete, the ATtiny15L sends back a prompt character (`*`) and waits for a new command.

The ATtiny15L's serial port commands are shown in Table 3. Note that the minimal set of commands needed to reprogram a target AVR processor includes only the first four commands in the list (i.e., `e`, `r`, `R`, and `>`). In fact, the initial version of SPPT used only these four commands.

The `a`, `f`, `m`, `F`, and `M` commands

were added to improve the speed of the programming operation. The program memory and EEPROM write bytes commands F and M activate AVR firmware to poll the AVR memory arrays for write-completion during reprogramming and to perform a read-back verification for each byte written. The AVR program memory and EEPROM arrays are self-timed. In order to write at the maximum speed, polling is needed. By using these commands with the PC serial port set for 38,400 bps, the program SPPT, just like its PC parallel port relative, can reprogram an AVR target processor very close to the maximum theoretical speed.

Command	Description
e	Enable programming mode on target
r	Reset target chip
R	Release target chip from programming mode
> a b c d	Send bytes a, b, c, and d to programmer (returns 1 byte)
a L H	Set current memory address to H:L
f n	Read n bytes from flash memory starting at current address
m n	Read n bytes from EEPROM starting at current address
F n a b ...	Write n bytes to flash memory starting at current address (n < 9)
M n a b ...	Write n bytes to EEPROM starting at current address (n < 9)
v	Show firmware version
n	No operation

Table 3— ***Bruce: Please write a 2- or 3-sentence caption.***

TIME TO PROGRAM

My programmer's eight-pin dual-inline 2-mm header J3 has one pin for every signal attached to the eight-pin ATtiny15L microcontroller. Therefore, the chip can be programmed in-circuit using the standard AVR low-voltage serial programming algorithm with practically any AVR programmer, including another ATtiny programmer. You can provide power with an external power supply and a custom-built programming cable (see Photo 3). Or, depending on the programmer, regulated power can be supplied by the programmer through J3. Alternatively, unregulated voltage can be supplied via P2, or the board can be self-powered via the serial port.

If power is supplied via the programming cable, then the device can be programmed using any voltage in the range of 2.7 to 5.5 V. Otherwise, the programmer needs to supply and accept SPI port logic levels in a safe range based on the V_{CC} voltage provided by the ATtiny programmer's on-board voltage regulator, U2.

Firmware for my ATtiny programmer, including the complete source code, is posted on the *Circuit Cellar* FTP site. Also posted is the AVR-GCC development environment V. 2.95, including a make file and all the WinAVR executables needed to rebuild the ATtiny programmer's firmware ROM files from source code. Source code and executables for the SPPT are included as well.

All of the electronic components I used for this project are available from Digi-Key, my favorite online electronic parts distributor. The tiny two-sided, surface-mount PCB shown in Photo 2 was built with ExpressPCB's free PCB layout software. The three-day PCB fabrication service is nice. The ExpressPCB layout file is also posted on the FTP site so you can make your own boards.

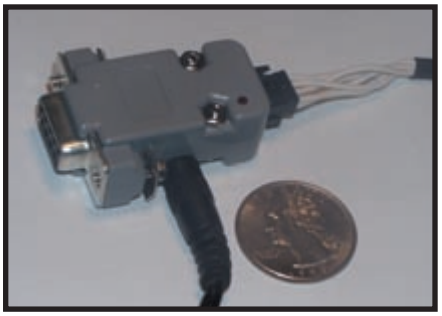

Photo 3—*External power is optional but useful for programming the programmer or using it without the serial port.*

TRULY TINY

I set out to build yet another serial port-based AVR programmer, and I did just that. I met all of my design goals. The programmer is truly tiny. It's arguably as small as can be for a device that plugs into a DE9 serial port, because it's housed completely within the nine-pin serial port connector hood. The inexpensive programmer features the ATtiny15L, which is one of Atmel's least expensive eight-pin ATtiny microcontrollers, and just a thimble full of other surface-mount parts.

In most applications, my programmer requires no external power supply because it's thrifty enough to steal current from a serial port's modem control outputs. For applications that require external power, you can power the programmer with an external wall wart using a dedicated DC power plug. Or, regulated V_{CC} in the range of 2.7 to 5.5 V can be supplied over the device's eight-pin 2-mm header connector.

My flexible programmer isn't limited to programming AVR microcontrollers. That's just one of its many potential applications. You can easily reprogram the programmer's ATtiny microcontroller in-circuit to function as a stand-alone general-purpose AVR-based controller. You can also use it as a serial-port-powered intelligent A/D data-logging device attached to a PC.

Because all eight of the ATtiny15L's pins are accessible from an external eight-pin connector, application programs (written in C or assembly) have access to all of the microcontroller's on-chip peripherals, including two 8-bit timer/counters with separate prescalars, one of which offers a 150-kHz, 8-bit, high-speed PWM output. Among other things, the PWM output can function as a digital-to-analog output.

Depending on the configuration (stand-alone or attached to a serial port), the device offers three or four channels of analog-to-digital conversion with 10 bits of resolution at up to 15,000 samples per second with programmable reference voltages. In Stand-Alone mode, the device even offers one differential ADC input with optional 20× gain. There are also two available analog comparator inputs. By changing the on-board DC regulator chip, you can set up the device to use a V_{CC} voltage between 2.7 and 5.5 V.

I use my programmer to digitize the audio from a Radio Shack scanner and send the data into my PC via the serial port. Besides some C code inside

the ATtiny15L, all this involves is a capacitor.

This project proves that you can develop a nontrivial application for a 512-instruction microcontroller without RAM completely in C language. With the supplied design files, source code, and GNU GCC development environment, you should be able to duplicate this feat. Now it's up to you to make it just a little bit better!

Bruce Lightner (lightner@lightner.net) works for Lightner Engineering in La Jolla, California. He discovered computers several decades ago and has been building hardware and software ever since. Bruce holds more than a dozen patents in the fields of computer architecture and telematics. His most recent venture is Networkcar, which produces wireless diagnostics and tracking devices for vehicles of all sizes.

PROJECT FILES

To download the code and additional files, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2006/190.

RESOURCES

Atmel Corp., "ATtiny15L: 8-bit Microcontroller with 1K Byte Flash," rev. 1187E, 2002.

———, "AVR910: In-System Programming", rev. 0943C, 2000, http://www.atmel.com/dyn/products/app_notes.asp?family_id=607.

B. Dean, AVR Downloader/Uploader (AVRDUDE), www.nongnu.org/avrdude/.

PicoWeb Parallel Port Programming Tool (PPPT), Lightner Engineering, www.picoweb.net/downloads.html.

S. Ball, "A Design Logic 2001 Primer", *Circuit Cellar* 127, 2001, www.circuitcellar.com/pastissues/articles/ball127/text.htm.

SOURCES

ATtiny15L Microcontroller
Atmel Corp.
www.atmel.com

PCB layout software
ExpressPCB
www.expresspcb.com

WinAVR Development tools
http://sourceforge.net/projects/winavr/