Tuesday Evening, 8:30 p.m. Client call
I received a call from a long-time client of mine. He was in a bit of a bind. He needed a way to do a little bit of data gathering for a demonstration of part of his system for his customers.

I understood his system well, most of it was either original work that I had done for him, or earlier design work that I maintained. Some of the work that I did for him involved creating designs that became volume products; other work was one-of-a-kind test equipment. It was the latter that he was interested in. He wanted a piece of hardware that would have probably a one-time use. “It can be held together with duct tape, for all I care,” he told me. “Just a quick design for you.”

He told me that he needed something that would operate on a laptop and gather about six different light levels. The device would need to sample about once a minute and log the data to the laptop hard drive. He described his plans. From what he told me I was making mental notes about this little project’s specification. A simple set of photoresistors would suffice for the levels of light that he would want to measure.

Six different channels - better make that eight, just to make sure — of measurement. The range and precision would not need to be great. For the relative measurements he was talking about 64 levels would suffice. Eight bits of analog to digital conversion would certainly work nicely.

His laptop only has a USB port, no serial or parallel ports. No problem, I was thinking. In fact I said this to him on the phone. He asked that I incorporate the data logging into a PC application that I had designed for him earlier. “No problem!” I repeated. “My demo is Thursday morning, I need hardware Wednesday by 4:00,” he said.” “No problem!” I’m not sure who said that, but it sounded a lot like me.

Tuesday Evening, 8:40 p.m. First plan
Perhaps it really wouldn’t be too bad. I know that I have some phototransistors in my junk box for testing. Eight ADC 0831 A/D parts reading the voltage levels from the phototransistors would convert the levels to digital values.

I have a board called the U401 that plugs into the USB port of a PC and can interface to SPI communication devices. My client already uses several of these boards.

So in the morning I will need to quickly assemble eight A/D chips, a set of screw terminals, and the USB to SPI board. This will mainly be made from rummaging through my parts bins. No problem.

As they say, ignorance is bliss. Had I made an effort to check my stock of converters before going to bed, I probably would have tossed and turned all night. But, I slept like a baby, oblivious to how my plans would be irrelevant come dawn.

Wednesday, 7:45 a.m. Frustration
My search turned up only a single A/D part. Some other project had consumed the other parts. I looked at several circuit boards imagining that I could steal enough A/D devices to make up for the missing seven parts. Nothing was found. My plans would need to change.

I couldn’t order additional A/D parts: even an overnight delivery would be too late for this fast-turn development effort.

My plan changed to using a multiplex chip to read the eight analog values with the single A/D converter. I can use the digital lines on the U401 to control the mux.

Wednesday, 9:00 a.m. Eureka!
While rummaging through bins to find a suitable multiplex chip, I had a stroke of luck. I bumped into an application note on SPI design for the Atmel AVR. I had used the AVR quite a bit to talk to the SPI A/D devices as a master. This app note was an intro to using the SPI on an AVR as a slave.

I was immediately filled with thoughts. Why not use a single AVR in place of the eight A/D converters? (Or what became a single A/D and a mux.) The AVR would be a slave SPI device, just like the A/D devices. There is an AVR that has eleven A/D pins. I used the ATTiny26 not so long ago.
I found an ATTiny26 immediately. In fact I found eight. Go figure. Now this will be the way that I will complete this design. But, the clock is ticking.

**Wednesday, 9:10 a.m.**
**Out of the starting gate**

Dropping a micro into the works is more complicated than just using the A/D parts. There is always more stuff to add to the design. A brownout detector, a reset device, and a crystal are needed in many little embedded designs.

But not with this AVR. The ATTiny26 has integral brownout detection. The reset pin will be connected to the USB to SPI board for in-system programming. And the microcontroller has an internal RC clock. This solution uses less hardware than the other approaches. There is, however, the matter of firmware. I needed to create firmware that will gather the analog values. SPI slave routines will let the U401 pull these values from the AVR.

**Wednesday, 9:30 a.m.** **Hardware assembly**

First things first. The hardware design was simple enough. [See the schematic] The AVR receives power from the U401, which is powered from the USB port of the laptop. The analog supply pin (AVCC) and the analog reference pin (AREF) are connected to VCC.

The eight lines on port A can be used for analog measurement. One of these lines, however, is the reference pin. Using a single line from port B for analog reading rounds out the eight total A/D channels.

The ATTiny26 is programmable in-circuit (ISP). The first three lines from port B are used for the SPI in-circuit programming as well as for the slave SPI operation. ISP works when the AVR is in reset; the reset line is also part of the ISP connection.

As a slave device the MOSI, MISO, and SCK lines are used for communication. When the firmware is running, this would suffice to move data in and out of the chip. I have, however, also included a “chip select” feature, implemented on port B bit six. This pin can be set to allow an external interrupt.

I assembled the components on a small board. Screw terminals were lined up in a row along one edge for the client’s connection to his photoresistors. There are pull-up resistors for each of these eight devices. The resistors pull up to VCC.

The board is sparsely populated. I added a transistor driving an LED from a spare port pin. A couple of test pins and “standard” AVR ISP header round out the parts. This is certainly a minimum design, as can be seen in the photo.

**Wednesday, 10:45 a.m.** **Rapid tools**

Now the pressure is on. It would have taken longer to assemble eight A/D parts, but once assembled I could start the PC application development. Unfortunately with a microcontroller there needs to be a middle step: firmware development.

I knew this when I chose the AVR as the solution to this problem. I had hoped that the trade-off would be even. In other words I would have a piece of hardware that would take less time to assemble because of fewer parts. But the time that I save would be consumed developing firmware for the AVR.

Luckily I didn’t start at ground zero. I had previously used many AVRs, the ATTiny26 specifically. That was why I had some in my bin.

And I wasn’t beginning the process of learning this architecture. I already had programmer tools, and, more specifically, a Rapid Application Development tool: the Imagecraft C compiler and development environment. The Imagecraft IDE has a tool that speeds development, the Application Builder. This mini application, selected from the “tool” menu of the IDE, helps build code for the peripherals of the AVR.

If you run Application Builder the initial page (CPU) allows you to select the target AVR device. The latest version of the compiler that I downloaded has support for the ATTiny26 by selecting “T26” under “Target CPU”.

I chose a crystal speed of 8MHz. When implementing timer functions, or using the UART peripherals, selecting the appropriate clock speed here will let the Application Builder automatically determine the best values for the often hard-to-calculate registers. Just getting the right values into UART registers can be a nightmare.

I used port B bit 6 for a chip select. I didn’t want to poll the pin to know when the U401 has selected the AVR. I wanted an interrupt to be triggered when the AVR is selected. Under “External Interrupts” I checked “INT0” and picked “Falling Edge” for the detection of the signal.

The selections “Watchdog”, “Pin change”, and “EEPROM” remained untouched.

In the second Application Builder page, Ports, selection of the port direction and initial values are made. I chose values appropriate for the hardware design.

The lower section of the Application Builder screen, the part that does not change when you change pages, has a “Preview” button. Selecting this button will bring up a sub screen containing the generated code.
The generated code can be inserted into a text editor to become part of the project code. The code below was that which was generated by these settings. Somewhere in your code, likely very early in “main()” you will need to make a call to the function that App Builder has created called “init_devices”. init_devices in turn calls routines that initialize the peripherals that you intend to use.

Listing 1—See Code Patch page 48

So far, the RAD tool has made code to initialize the port lines and interrupt registers. When the time came, the App Builder was used to initialize the SPI and A/D converter, as well as set up a template routine for the SPI byte reception and the external interrupt.

The Application Builder helped more with the project, but when I got to this point my goal was to make a first initial test. Something simple just to make sure that I have my ducks in a row.

Wednesday, 11:15 a.m. Rapid blinking

The board has an output device. The transistor/LED combination is there to provide a “heartbeat” — an indication that the processor is alive and kicking.

The LED was also the first part of the project to show any life. It made a nice sanity check to make sure that I was getting code into the processor, that it was running the code, and that it was not stuck in reset.

To the initialization code that was automatically generated, I added a “main()” routine. The intent of the routine was to turn the LED on, delay, turn it off, delay, and repeat forever.

Listing 2—See Code Patch page 48

I compiled the code and turned to the tools menu item for the programmer. The IDE comes with an “In System Programmer” to program the AVR. If you have programming hardware compatible with this tool, then you can program the AVR without leaving the IDE.

I actually use my U401 to program the AVR via a USB port. I have added my programming software to the IDE menu, so that I can enjoy the same convenience. The U401 application will run when the menu item is selected. From the U401 AVR programmer application I can program and erase the AVR, as well as communicate with it via SPI. I set the fuse settings of the ATTiny26 for 8 MHz operation, and loaded the AVR with code. I then selected “Run”.

Eureka! Blinking light. Time for a quick lunch break.

I added photoresistors to the small circuit. At least I was right about having a lot of these in my parts bin. They don’t all match, but for testing they will suffice. Once again I turned to the App Builder to quickly help put the A/D operation in place. The App Builder “Analog” page is for A/D selections. The ATTiny26 A/D channels are ten bit. For the sake of simplicity (and speed) I only wanted to read eight bits. One very nice feature of the A/D subsystem lends itself well to doing this. The ten bits can be placed into an eight-bit register either justified to the right, or left. I certainly didn’t need the lower two bits of the ten. I was interested only in the most significant eight. This feature is realized by selecting “Left adjust”.

I also chose “ADC enable” and “Free-run select”. I wanted the analog conversions to “just happen”. A call to the A/D initialization routine, “adc_init”, was added to the “init_devices” routine.

I added code to the main loop. I left the light blinking in place, but added code to read the A/D register, and advance to the next A/D channel.
The A/D channels stored in the “ADValue” array would need to be accessed by the slave SPI routine. For the final time, I let the App Builder help.

On the USI page I selected 3-wire (SPI) mode with no clock source. When there is a full byte in the receive register an interrupt can be generated. A template for the interrupt is created by selecting “Counter ovf interrupt”. Once again the USI init code is added to the “init_devices” routine.

The SPI master device, the U401, will be sending two bytes to the AVR. The first byte addresses the A/D channel number. The second byte sent is a dummy byte to clock the A/D value out of the AVR. I wanted to keep the two SPI devices in sync. The external interrupt routine is triggered when the U401 selects the AVR. The routine only needs to clear the pointer to the reply array.

The USI ovf interrupt will place the A/D value indexed by the first byte in the message into the USI data register after the first byte has been received. This byte is then clocked back to the U401 by the transmission of the dummy byte.

Initially I added code to select channel zero, and display its value. The pointer just barely moved. I covered the sensor, shined light into the sensor. The results never changed. I even tried yelling at the sensor. Still no change.

Out of desperation (and panic) I selected channel one. The slider jumped to the top. I covered the sensor and the slider slid down. Nice. It turned out that the first sensor was damaged. After replacing it I added code to scan all of the sensors. I covered the sensors with a pattern of fingers to get some random results.

Wednesday, 2:30 p.m. Application test

Only an hour before I need to hit the road, I started the PC application changes. I created a sub-screen to my client’s application just for this testing. The application displays the eight A/D values on sliders. There is a button to transmit the SPI message to the AVR, and gather the A/D values. The final application logs the readings to a file every 60 seconds.

The VB application opens a handle to the U401. Commands are sent from the application to initialize the U401. The SPI subsystem of the USB board is enabled, and a digital line is set to output for the AVR chip select.

Once initialized, a command from the application sets the AVR chip select line low. The external interrupt in the AVR senses this edge and resets the data pointer (index) of the array. The application then writes a SPI command, and returns the chip select high.

The application performs this action eight times, once for each channel. The AVR sends back the reading associated with the channel.

Initially I added code to select channel zero, and display its value. The pointer just barely moved. I covered the sensor, shined light into the sensor. The results never changed. I even tried yelling at the sensor. Still no change.

Out of desperation (and panic) I selected channel one. The slider jumped to the top. I covered the sensor and the slider slid down. Nice. It turned out that the first sensor was damaged. After replacing it I added code to scan all of the sensors. I covered the sensors with a pattern of fingers to get some random results.

Wednesday, 3:20 p.m. Delivery

I copied the files I needed to a CD, and carefully placed the hardware into an anti-static bag. I hopped into the batmobile, (Well, O.K. Minivan.) and headed out to my client’s office.

About 18 hours earlier I had made up my mind to solve the problem with eight serial A/D parts. I found myself in quite a bind when I realized that I didn’t have the parts.

But a shift in my approach led me to use the AVR to replace the eight A/D converters. It was certainly simple using a RAD tool to quickly generate the initialization code, time that I would otherwise waste pouring over the datasheets.

My client was oblivious to the rapid pace of development, but liked the demo. “Nice,” he said. “I can’t wait to demo it next week.”

“Next week?” I asked.

“Yes. Next week Thursday, like I said.”