CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, or www.circuitcellar.com. Entire contents copyright ©2001 Circuit Cellar Inc. All rights reserved.



APPLIED PCs

by Fred Eady

Build a Coyote Protocol Converter

To prepare for this month's column, Fred devoted a lot of time to thinking about an appropriate project for Z-World's Rabbit 3000-based Coyote. After mulling over plans for sprinkler controllers and water heater monitors, he decided to leave the brainstorming up to you. This month, Fred surveys the Coyote's capabilities in an effort to prepare you for future projects.

Politicians like to get elected, mechanics like to fix things, musicians strive for melodic perfection, and engineers, well, they just engineer. I like to sit around studying widgets made of complex PCBs loaded with lots of itty-bitty parts.

There are times when an idea for an article just comes to me. Other times, readers send me their ideas. This month, I couldn't see the forest through the trees. And, to add insult to injury, I was lost in the forest as well. For a day or two, I pondered what to tell you about this neat collection of resistors, capacitors, ICs, and industrial-strength connectors spread across a couple of high-density PCBs. Project idea after project idea passed through my head. I decided that I really don't need another water sprinkler controller, and you probably don't want to read about one either.

So, there I was wandering around the Florida Room trying my best to come up with an interesting application for this gadget named after wild dogs that are famous for singing with the cowboys and howling at the moon. After I cleared my head, it occurred to me that I was walking the wrong dog. I decided to start from square one, and took an inventory of what the product I wanted to tell you about is made of.



Photo 1—*The heavy-duty headers and simple construction lend the Coyote to hazardous duty. The Rabbit 3000 core module on this Coyote is equipped with Ethernet.*

MY COYOTE

First of all, the Z-World Coyote is designed with an Ethernet interface that's supported by the latest Rabbit 3000 microprocessor. This doodad is as good as any, with Ethernet capability that makes me wag my tail and perk up my ears.

There's also a lot of brawny I/O in the Coyote's poundage, including nine filtered and protected digital inputs, eight digital outputs capable of sinking 200 mA at 36 VDC, and seven high-speed unfiltered protected digital inputs. So, I can turn on and off just about anything over an Ethernet link under precise microprocessor control.

A big green coin battery on the Coyote's I/O board suggests a battery-backed, real-time clock. Not only can I turn stuff on and off, I can do so on a schedule and in bad weather. An assessment of the available analog I/O reveals a single A/D input and a couple of D/A outputs. The A/D module could come in handy when making switching or data output decisions that are based on external voltages.

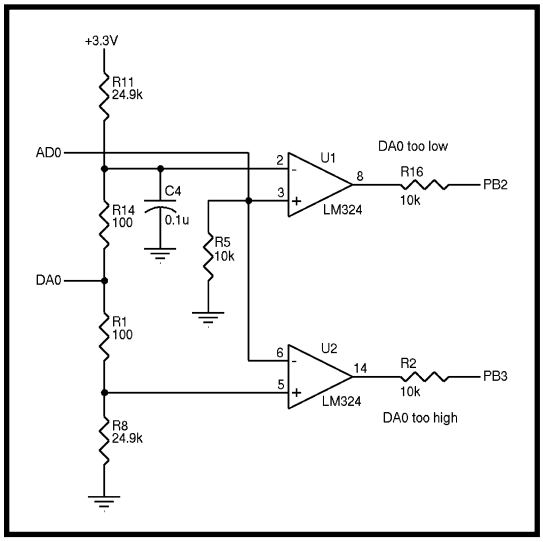


Figure 1—*Very clever! DA0 is used to feed the comparison voltage, which is ultimately used as the resulting voltage.*

The D/A output could be used in a voltage-control application like setting a machine tool motor's speed and direction. I could really make some motors sing or generate some weird auxiliary voltages using the two Coyote 9-bit PWM outputs.

In addition to the Ethernet port, my personal favorite, the Coyote has six serial ports. I should be able to do something with them; after all, serial ports are my life. In the memory corner, there's a 128-KB bucket of SRAM supported by a 256-KB washtub of flash memory that I can cram code into. Four programmer-accessible onboard LEDs act as debug indicators. You may use them to flash coded status to an end user.

The final feature that should have tipped me off regarding the application of this embedded system is the 100 mm × 100 mm board, which is perfect for DIN rail mounting. Photo 1 shows my *Canis latrans* (coyote) sans a leash.

WALKING THE COYOTE

It's pretty obvious that the Coyote is an industrial animal. I realized this when I tried to separate the RCM3010 RabbitCore module from the Coyote's I/O board. An antivibration screw holds the RCM3010 tightly in its socket. Another giveaway is the presence of heavy-duty Molex headers on the I/O board. The overwhelming presence of the two RabbitNet serial expansion connectors plus an additional RS-485 portal means this little wolf is intended to roam the shop floor.

After gathering all of these clues, my search for an application for the Coyote ended. You may download the pinout diagram from the *Circuit Cellar* ftp site. The various communications ports coupled with an ample amount of SRAM, a microprocessor, and a C compiler tailored to the I/O and communications facilities makes the Coyote a perfect protocol converter with ADC and DAC functionality.

The Coyote has polarized Molex terminals. All of the digital I/O is lined up along the bottom edge. J3 is a 10-position header on 0.1″ centers and pins out the digital outputs at positions zero through seven (OUT0 through OUT7).

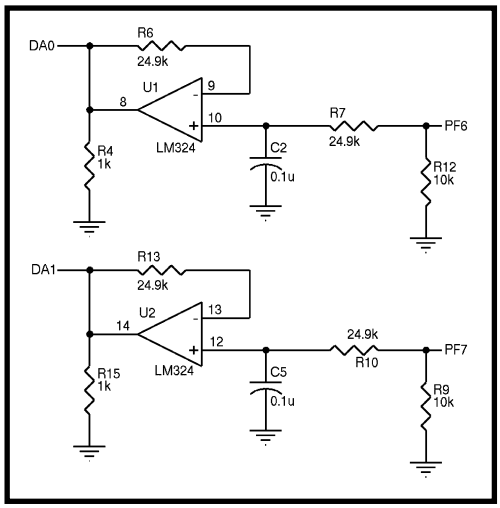


Figure 2—*PWM in, voltage out. It's that simple.*

J3's pin 9 is removed and pin 10—which is called pin 9 in the pinout diagram—is the K+ input. Connector J3's OUTX pins can each sink 200 mA. K+ is an external voltage input that ranges from 3.3 to 40 VDC.

If an external K+ voltage is used, it should be able to handle the entire current load for the OUTX pin or the pins being employed. Z-World engineers recommend that you use a K+ voltage for driving solenoids or relays. Doing so prevents voltage spikes from letting the life's breath out of your Coyote. Using the K+ line as a voltage source point puts a reverse biased relief diode across the load.

K+ and its associated ground are applied to the Coyote using a polarized two-pin 0.156″ center connector, J10. The J10 connection point is called the high-power driver source. The K+ pin on J3 is electrically connected to the J10 K+ pin.

Directly below J3 are the digital input pins on J11 and J12. IN00 through IN13 are protected inputs capable of handling input voltages between –36 and 36 VDC. IN14's input swings only between –36 and 5 VDC. Each INXX input is factory-set to pull up to 3.3 VDC. IN00 through IN07 can be pulled up to K+ or down to ground by moving an SMT 0-Ω jumper resistor. The filtered inputs—IN00 through IN07 and IN15—use a low-pass filter arrangement consisting of a 22-kΩ resistor in series with the

Listing 1—*If the Coyote had to use the "Big Router," then at least the* `MY_GATEWAY` *IP address would have to be supplied here. There are many levels to this routine, which are selected by functions and defined using* `TCPCONFIG`*. As you can see, this is* `TCPCONFIG 1`*.*

```
#if TCPCONFIG >= 100
      #use "custom_config.lib"
#else
//General configuration: You can specify your IP address, net
  mask, name server, and gateway at this location, and they will
      be used in each of the individual configurations below.
      #define _PRIMARY_STATIC_IP           "192.168.0.150"
      #define _PRIMARY_NETMASK             "255.255.255.0"
      #ifndef MY_NAMESERVER
            #define MY_NAMESERVER          "10.10.0.1"
      #endif
      #ifndef MY_GATEWAY
            #define MY_GATEWAY             "10.10.0.1"
      #endif

      #if TCPCONFIG == 1
//Configure 1: Simple static configuration of a single Ethernet
  interface.
            #define USE_ETHERNET           1
            #define IFCONFIG_ETH0 \
                        IFS_IPADDR,aton(_PRIMARY_STATIC_IP),
\
                        IFS_NETMASK,aton(_PRIMARY_NETMASK),
\
                        IFS_UP

      #endif
```

input on the J1X pin side and a 10-nF capacitor that's tied to ground on the port side of the 22-kΩ input resistor.

The analog gallery sits on connector J1, which is on the top right side of the Coyote. Although there are two D/A outputs, only DA1 can be used if the ADC is employed. Note that there aren't any specialized A/D or D/A ICs on the Coyote. As you can see in Figure 1, the A/D module is actually a pair of LM324 op-amps configured as a window comparator. The idea is to feed a voltage into DA0 until the op-amps' outputs (PB2 and PB3) are low. If both op-amp outputs are low, the DA0 input voltage is equal to the AD0 voltage.

Because the DA0 voltage is known, it's used as the AD0 input voltage value. Of course, the ideal situation (i.e., PB2 and PB3 being low at the same time) doesn't always occur, and that's when the 1-bit high or low readings occur. There's nothing unusual about that because even specialized conversion ICs exhibit this behavior. The DA0 swings between 0.1 and 3.1 V in 3.22-mV steps inside the 13.2-mV comparator window. This equates to 930 steps and more than 9 bits of resolution.

The DA0 voltage is raised or lowered according to which comparator output is high. For instance, if the PB2 output is high, the DA0 voltage is raised by the application. Conversely, PB3 in a true, or high, state would signal the application to lower the applied DA0 voltage.

You're probably wondering how this works without a reference voltage. Well, the answer is straightforward: Each Coyote is calibrated, and the calibration constants are retained in the user block data area of the flash memory. Programs are included with the Coyote to assist you in calibrating the ADC or saving and recalling calibration values. Each A/D conversion takes less than 100 ms via my Coyote running with a 29.4-MHz clock.

I don't like wasting silicon, and it seems as though Z-World engineers don't either, because they used the remaining op-amps in the LM324 package to assemble the D/A modules you see in Figure 2. Like the A/D module, the D/A module relies on the supply voltage and other factors (e.g., precision resistors) for accuracy without the support of a stable reference voltage. And, like the A/D module, the D/A module can be calibrated. It has software that supports calibration as well as the calibration value storage and retrieval process.

Looking at Figure 2, it's clear that the inputs to the op-amps are fed with a PWM signal. Because of my description of the A/D module's resolution, you already know that the Rabbit 3000 can supply a PWM signal with 10-bit resolution, which allows a D/A output resolution of 3.22 mV. With the D/A circuitry shown in Figure 2 and factory calibration constants stored in flash memory, the Coyote D/A module can deliver a resolution of 3.22 mV with a peak-to-peak ripple

Listing 2—*The `serX` defines can be used with any of the Rabbit's six serial ports. Only four of the six serial ports are pinned out directly on the Coyote. The RS-232 library makes implementing a buffered serial port easy on the Coyote. Note: Asterisks (*) denote code between fragments.*

```
//Serial port settings are 1 = E, 2 = F, 3 = D (RS-485)
#define SERIAL_PORT  3
#define BAUD_RATE    19200L
#define INBUFSIZE    1023
#define OUTBUFSIZE   1023
*****************************************************************
//Select serial port to use
#elif (SERIAL_PORT==3)

      #define serXopen      serDopen
      #define serXread      serDread
      #define serXgetc      serDgetc
      #define serXrdUsed    serDrdUsed
      #define serXwrite     serDwrite
      #define serXclose     serDclose
      #define serXwrFlush   serDwrFlush
      #define serXrdFlush   serDrdFlush
      #define serXwrFree    serDwrFree
      #define DINBUFSIZE    INBUFSIZE
      #define DOUTBUFSIZE   OUTBUFSIZE
*****************************************************************
if(state->open_timeout)
break;
}
serXopen(vs_info.baud);
ser485Rx();
*****************************************************************
if(bytes_to_write>0) {
 sock_read(socket,state->buffer,bytes_to_write);
 ser485Tx();
 serXwrite(state->buffer,bytes_to_write);
*****************************************************************
if(bytes_to_write>0) {
 ser485Rx();
 serXread(state->buffer,bytes_to_write,0);
 sock_write(socket,state->buffer,bytes_to_write);
*****************************************************************
else {
  ser485Rx();
  while(state->offset<VS_MAXOFFSET && (ch=serXgetc())!=-1) {
*****************************************************************
void main()
{
      brdInit();
      sock_init();

      vs_init(&vs_state);
      serXopen(vs_info.baud);
      ser485Rx();           //Disable transmitter, and initially
                              put in Receive mode
      serXwrFlush();
      serXrdFlush();
```

of 6 mV over a range of 0.1 to 3.1 V with a 17.5-ms settling time.

There's nothing remarkable or unique about the Coyote's serial ports. Currently, the RabbitNet ports don't have anyone to play with, and there's not much in the way of documentation for them. I know that they are SPI ports running RS-422. The clocked CMOS serial port on J9 gives me the idea that I²C may be another protocol conversion possibility; however, after some checking, the clocked CMOS port isn't an open-collector port, and the Ethernet port is using the open-collector port I would need for I²C operation (i.e., port D). If I²C were a requirement, I would opt for a Coyote without the Ethernet electronics on the Rabbit module (RCM3110). RS-485 access is also provided on J9, and the pair of RS-232 serial ports is pinned out on J6.

Listing 3—*Note that the data rate can be different for each interface. Also notice the absence of any user-written functions. I pieced this together using nothing but standard Rabbit RS-232 library calls. All I had to do to set up the serial ports was define them and their data rates.*

```
#class auto
#define INBUFSIZE     1023
#define OUTBUFSIZE    1023
#define ser485open    serDopen
#define ser485read    serDread
#define ser485putc    serDputc
#define ser485getc    serDgetc
#define ser485rdUsed serDrdUsed
#define ser485write   serDwrite
#define ser485close   serDclose
#define ser485wrFlush serDwrFlush
#define ser485rdFlush serDrdFlush
#define ser485wrFree  serDwrFree
#define DINBUFSIZE  15
#define DOUTBUFSIZE 15
#define ser232open    serEopen
#define ser232read    serEread
#define ser232putc    serEputc
#define ser232getc    serEgetc
#define ser232rdUsed  serErdUsed
#define ser232write   serEwrite
#define ser232close   serEclose
#define ser232wrFlush serEwrFlush
#define ser232rdFlush serErdFlush
#define ser232wrFree  serEwrFree
#define EINBUFSIZE    INBUFSIZE
#define EOUTBUFSIZE   OUTBUFSIZE
#ifndef _485BAUD
#define _485BAUD 19200
#endif
#ifndef _232BAUD
#define _232BAUD 19200
#endif
main()
{
   auto int nIn1;
   brdInit();                          //Initialize board
   ser485open(_485BAUD);               //Set data rate first
   ser485Rx();                         //Disable transmitter, and
                                         initially put in Receive mode
   ser485wrFlush();
   ser485rdFlush();
   ser232open(_232BAUD);
   ser232wrFlush();
   ser232rdFlush();
   while (1)
   {
       ser485Rx();
       if(ser485rdUsed())
       {
          nIn1 = ser485getc();
          ser232putc(nIn1);
       }
       if(ser232rdUsed())
       {
          nIn1 = ser232getc();
          ser485Tx();
          ser485putc(nIn1);
          while (ser485getc() == -1);  //Wait for echo
          ser485Rx();                  //Disable transmitter
       }
   }
}
```

ETHERNET TO RS-232

Like all Z-World development kits, the Coyote comes complete with its own special version of Dynamic C. Using the services of Dynamic C and the Coyote communications hardware, you can easily put together a shop-floor Ethernet-to-RS-232 serial converter.

The idea is pretty simple: All you need to do is enable the Ethernet interface and define all of the necessary IP addressing for the Coyote. Next, enable an RS-232 serial port and specify the data rate, parity, stop bit (or bits), and the number of data bits between the start and stop bits. Then, define a buffer tub for all of the bits and devise a scheme to move the bits between the tub and the RS-232 and Ethernet interfaces.

I figured this would take a day or two to code, debug, and test. Before I started the coding, I decided to take a look at the TCP/IP samples directory that comes with Dynamic C for the Coyote. I found a serialexa.c file. After reading the code description for serialexa.c, all I had to do was twiddle a few values in an information structure within the example code and fill in the Coyote IP blanks in the TCP_CONFIG.LIB file. After that came the Ethernet-to-RS-232 serial converter. As it turned out, I only had to fill in my Coyote's IP address (192.168.0.150) and network mask (255.255.255.0) in the TCP_CONFIG.LIB file to make things go (see Listing 1).

The sample code for the Ethernet-to-RS-232 serial interface is based on a state machine that uses the socket status (INIT, LISTENING, OPENING, and CLOSING) as states to determine how to handle the data transfers. The serial

port is always open. The buffer tub of bits is composed of SRAM buffers for both the serial and Ethernet ports.

The Coyote Ethernet-to-RS-232 serial program initializes a socket, starts the serial port, and listens for a connection. I used telnet to establish a session with the Coyote.

For the serial-side transactions, I ran Tera Term Pro on a separate PC and tied one of the Tera Term Pro PC's serial ports to the Coyote's serial port E. As simple as this seems, it didn't work right out of the box. After a few hours of scratching my head and running traces, I found that the J6 RS-232 serial port designations were incorrect. After lots of pin swapping, I finally consulted the Coyote's schematic and found the GND pin to be J6 pin 9 rather than J6 pin 10. After I straightened this out, I lashed up a corrected DCE cable, using words that a victorious Navy fighter pilot would scream after a dogfight, "Splash one."

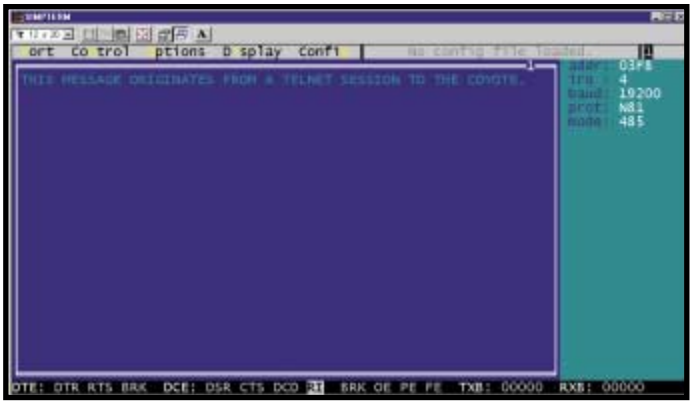


Photo 2—*I took advantage of some freeware from B&B Electronics, who also supplied the RS-485 card I used in the PC. The RS-485 card has four ports that can be switched between RS-232, RS-422, and RS-485 among all of the COM ports and IRQs.*

ETHERNET TO RS-485

The Coyote RS-485 interface is a two-wire, half-duplex configuration. My RS-485 card in the RS-485 PC is a four-wire, full-duplex configuration. So, I had to do some more cabling.

After checking the pinout diagram, I saw that the labels for the RS-485 port (J9) matched the pin designations. To turn the PC's four-wire interface into a two-wire interface, I tied the RS-485 "A," or negative, transmit and receive pins together, and then did the same for the "B," or positive, transmit and receive pins.

To keep the wiring tangle factor low, I used a standard eight-pin RJ-45 female connector, Cat 5 cabling, and one of the Molex female headers and pins that comes with the Coyote. I punched down a short between the female RJ-45 connector's pin 1 (TDB+) and 3 (RDB+), and 2 (TDA–) and 4 (RDA–).

Because the PC's RS-485 interface is an RJ-45 female, I opted for a standard 568B-wired Cat 5 patch cable to connect the Coyote's RS-485 interface to the RS-485 card. This placed my PC's RS-485+ on pins 1 and 3, and the RS-485– on pins 2 and 4. I used pins 1 and 2 of the Cat 5 patch cable to carry the RS-485 signals and wired the corresponding pins into the Coyote's J9

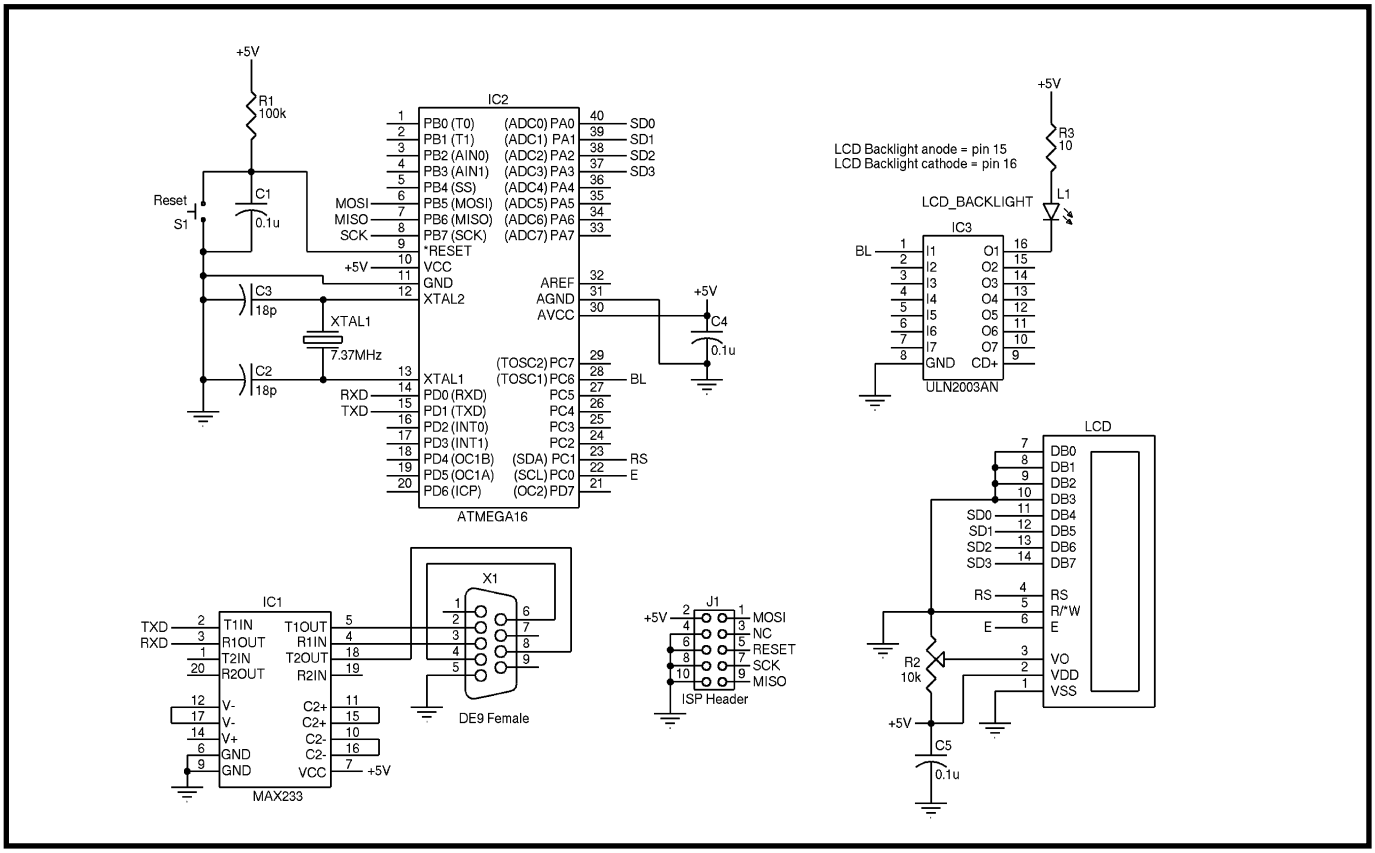


Figure 3—*You can get the Coyote schematic from the Z-World web site. I used this string of parts to make a thingamajig to bark with the Coyote.*

Molex connector.

I also added a connection on the J9 Molex connector ground pin to pin 8 of the Cat 5 patch cable, which is ground on the PC's RS-485 card. Technically, I could run the RS-485 connection without a ground, but it's always a good idea to include the third wire.

I used the same example program, serialexa.c, to build the Ethernet-to-RS-485 converter firmware. All I had to do was add a define block for the RS-485 port (serial port D) and insert the `ser485Rx()` and `ser485Tx()` functions ahead of every serial read (receive) and write (transmit) call in the serialexa.c code.

The `ser485Rx()` and `ser485Tx()` functions enable the functionality of the RS-485 transmit-enable signal that's assigned to PA4 (i.e., I/O port A, pin 4). Because the Coyote is configured for Half Duplex mode, the `ser485Xx()` functions provide the code to enable and disable the RS-485 transmitter on the Coyote's Sipex SP483 RS-485 driver IC via PA4. Otherwise, the data transfers resemble RS-232. A fragmented look at my modifications for RS-485 operation is shown in Listing 2. The RS-485 portion of the Ethernet-to-RS-485 converter session is shown in Photo 2. "Splash two."

RS-232 TO RS-485

I'm on a roll. So, let's move on to converting the RS-232 and RS-485 serial streams.

You can make this as complicated as you want, however, I refuse to breed complexity. Again, using constructs from here and there, I pieced together some code that turns the Coyote into a serial pipe. RS-232 in yields RS-485 out, and vice versa. All of the code in Listing 3 uses the services of the standard Rabbit 3000 RS-232 library. The RS-232 function calls allow me to sit and wait for incoming data from either serial port, and redirect the incoming serial data to an opposite outgoing serial port. Simple yet effective. "Splash three."

CLOCKED COYOTE

I returned to the Z-World web site to see if I could squeeze out some additional scoop on RabbitNet, but was unsuccessful. So, I decided to concentrate on the Coyote's clocked CMOS communications port. But, it seems as though what I call "clocked" is more like the SPI-based RS-422 RabbitNet port than the clocked CMOS port on the Coyote. As it turns out, the clocked CMOS port is simply another asynchronous communications port without RS-232 conversion circuitry.



Photo 3—*The Atmega16 may be overkill for this application, but I figure I'll be adding more gadgets to this board, and having all of the AVR functionality plus a pack of I/O pins will make life easier later on. The RS-232 port is just below the SP233ACP, which is removed when the clocked CMOS port is connected to the three-pin connector to the left of the serial port.*

To test that theory, I put together a serially controlled 4 × 20 LCD module based on an AVR Atmega16 (see Figure 3). The idea was to connect the clocked CMOS serial port to the AVR board and send a message to the LCD from a telnet session aimed at the Coyote.

The coding was a piece of cake. I simply added RS-232 library defines for port C to the existing serialexa.c program. On the AVR side, I stuffed the Atmega16 with an interrupt-driven asynchronous communications application that manipulates the LCD. I tested the AVR board using a Sipex SP233ACP RS-232 converter IC attached directly to a Tera Term Pro terminal emulation session on the Tera Term-equipped PC. I used the ImageCraft AVR C compiler to architect the Atmega16 firmware, and implemented an Atmel AVR ISP programmer pod to push the finished code onto the AVR.

After debugging the AVR code with Tera Term Pro, I removed the Sipex SP233ACP IC and connected the AVR/LCD module's USART directly to the Coyote's clocked CMOS pins. I did this in a three-wire, null-modem fashion (i.e., Coyote TX to AVR RX, and AVR TX to Coyote RX, plus a common ground). You may download the AVR code from the *Circuit Cellar* ftp site. The companion AVR hardware is depicted in Photo 3.

Even with the three-wire RS-232 converterless communications success, I still wondered about the CLKC line. Further study indicated that the clocked CMOS port could be used in SPI mode. The Coyote's clocked CMOS port is supported in SPI mode by a Dynamic C SPI library. So, the clocked CMOS port can be used to interface to external specialty ICs that communicate using SPI. And, oh, yeah, "Splash four."

"BEEP, BEEP"

Well, the Coyote has arrived and like that fast bird we all know and love, I've got to go. The Coyote I've examined has a belly full of Rabbit, which gives it the power to measure, control, and communicate using a variety of protocols on just as many physical interfaces. On second thought, the Coyote would make a perfect sprinkler controller or hot-water heater monitor/controller, but I'll leave the final applications to you. My job is done, and once again I've proven that even products based on floppy-eared varmints and named for wild dogs don't have to be complicated to be embedded.

Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.

PROJECT FILES

To download the code and pinout diagram, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2003/156/.

SOURCES

ATmega16 AVR
Atmel Corp.
(408) 436-4314
www.atmel.com

3PXCC4A RS-232/485 Card
B&B Electronics Manufacturing Co.
(815) 433-5100
www.bb-elec.com

AVR C compiler
ImageCraft Creations, Inc.
(650) 493-9326
www.imagecraft.com

SP233ACP Converter IC
Sipex Corp.
www.sipex.com

Coyote
Z-World
(530) 757-3737
www.zworld.com