by Peter Montgomery

Serial Network Hub (Part 2)

Circuit Design and Usage

You are well on your way to building a six-port RS-485 hub that can implement a network packet system. Here you learn the details about the software and hardware, as well as some hidden "gotchas" present in RS-485 chips.

In the first part of this article series, I described my need for an RS-485 hub as part of the hardware running my yearly Halloween display. A quick visit to my website (www.socalhalloween.com) will show you the level of sophistication in my computer-controlled show, which includes multiple animatronic figures having conversations as well as physical effects, sound effects, and more. The system is run by a laptop connected to a series of daisy-chained hardware nodes that communicate via RS-485. Normal RS-485 network topologies run either half-duplex—with all transmitters and receivers connected to a single wire pair—or full-duplex—with a master node transmitting to all slave nodes on one wire pair, while listening to all slave nodes on a separate wire pair. In either case, only a single slave node can transmit at a time, and some mechanism (polling, token passing, etc.) must be implemented to prevent multiple slaves from transmitting simultaneously and destroying each other's packets. I originally implemented a "store and forward" scheme on each slave node to allow slaves to transmit whenever they needed to, but it required extra circuitry, complicated the software running on each node, and introduced increasing time delays for slave nodes sending messages to the master node.

My ultimate solution was to build a six-port RS-485 hub that implemented my Halloween network packet system and allowed me to greatly simplify the wiring of the nodes. The design criteria included writing as much code as possible in C, minimizing the number of UART chips (the hub has six ports) on the PCB, building a device that has its own power supply in the case (no wall warts!), and building a device that is small and professional-looking. Did I succeed? Let's find out.

CIRCUIT DESIGN

Like everyone else who does this kind of stuff, I have my personal favorite processors. My processors of choice are

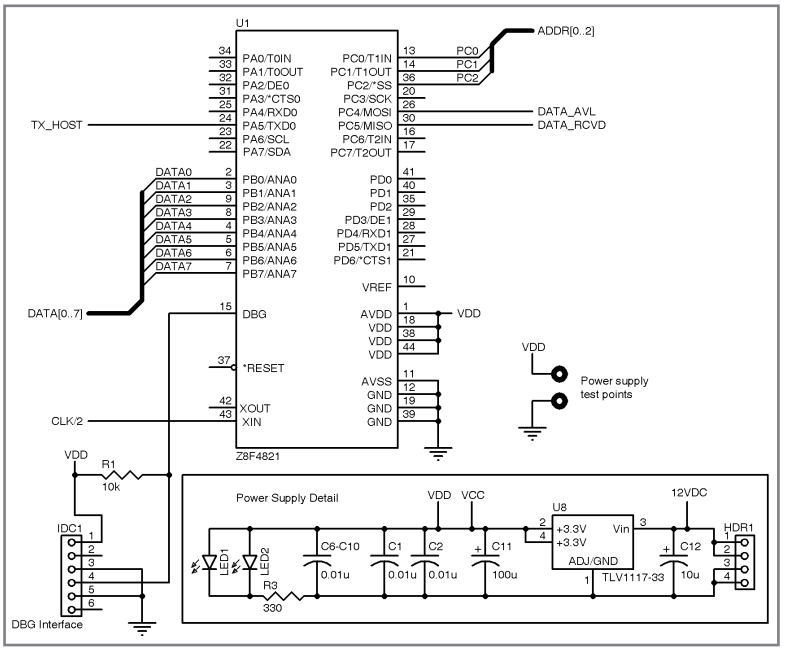

Figure 1—This is the Z8 portion of the Serial Network Hub. It manages all the packet buffering and communication.

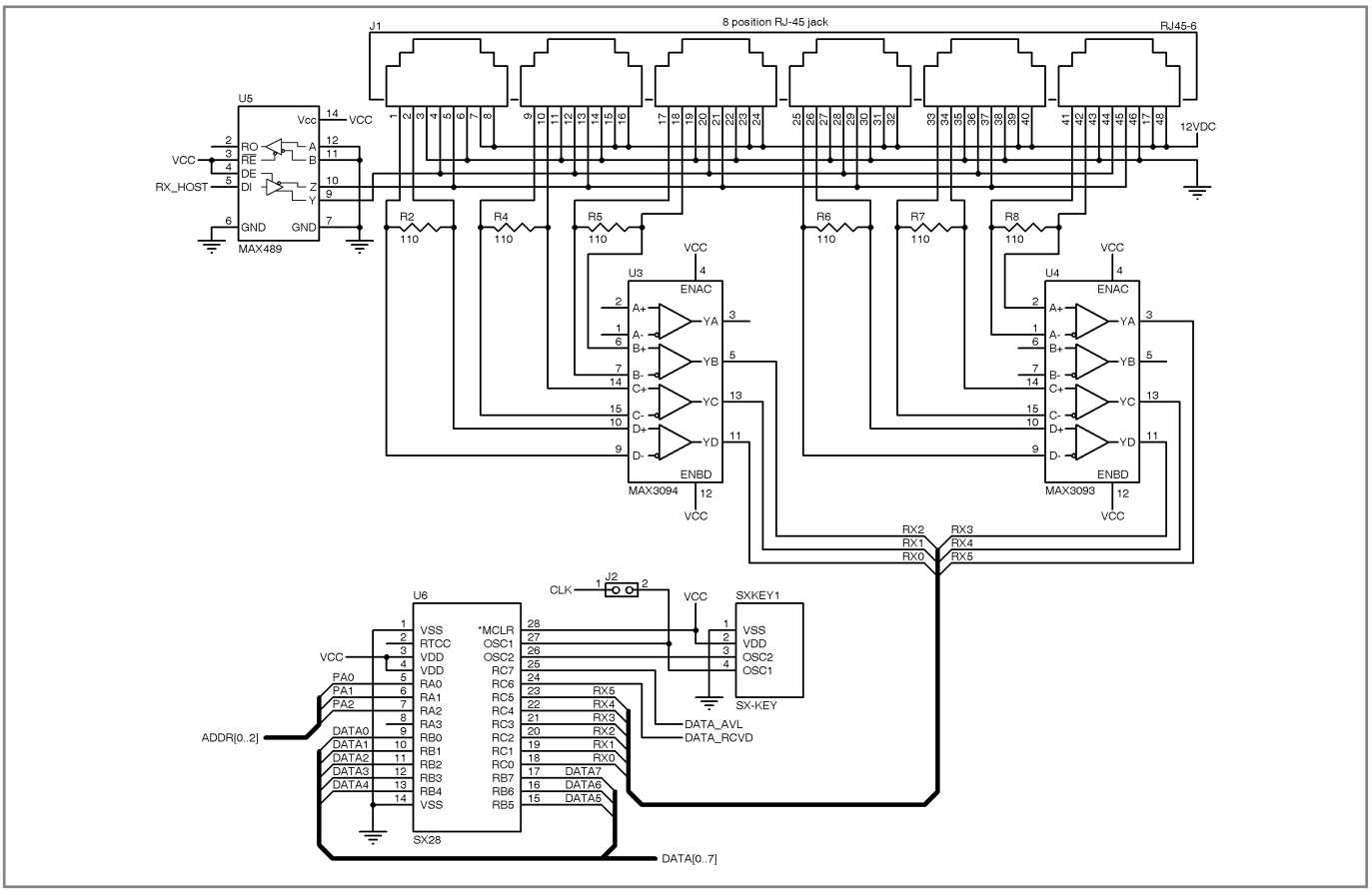

Figure 2—This shows the S portion of the hub, which is responsible for implementing a six-port UART in software. The MAX3083 chips handle the RS-485 level conversion.

the Zilog Z8 Encore! and the Parallax SX chips. The Zilog Z8 is an 8-bit chip that lets me code in C, using a free IDE and compiler provided by Zilog. The IDE and chips support hardware debugging at the C source or assembly language level. The parts range from eight-pin to 68-pin devices, with EEPROM from 1 to 64 KB and up to 4 KB of RAM. They come in through-hole or surface-mount parts, and nominally run at 3.3 V while accepting a range from 3 to 3.6 V.

The Parallax SX is a more unique beast. Running at speeds up to 75 MHz (and up to 75 MIPS) and featuring a jitter-free, deterministic interrupt scheme, they are almost FPGA-like in their abilities. The interrupt system is where the power of this chip lies. The SX chips have an 8-bit real-time clock counter (RTCC) with prescaler that generates interrupts at pretty much any interval you desire. Unlike pretty much every other CPU on the market, the SX always takes three clock cycles to enter or exit the RTCC interrupt regardless of what instruction the processor is executing at the time. Further, there is an exit opcode which lets the chip automatically update the timer value to account for the number of cycles used during the interrupt handler, regardless of whether the interrupt code path was longer or shorter for a given interrupt. Imagine that you've set the RTCC interrupt to occur every 129 clock cycles, and that the interrupt handler takes anywhere from 51 to 69 clock cycles depending on what it does during the interrupt. Regardless of the exact number of clocks that the interrupt handler uses on a given interrupt, the special exit opcode will automatically reload the RTCC counter with the correct value to make the next interrupt happen exactly 129 cycles from the previous interrupt. These two features (fixed interrupt latency and the special exit opcode) provide a completely jitter-free interrupt period. Because of this feature, the SX allows you to create virtual peripherals that are rock steady. For example, if you want a PWM generator, you write one that runs in the interrupt handler. Want a UART? Same approach. Want six UARTs? Just make sure the code to execute them can complete in the time between interrupts and you can have them. Writing software lets you make the chip do what you need it to do, rather than have a lot of extra peripherals on board that are either insufficient or over abundant for the needs of a particular design.

A Zilog Z8F4821 CPU controls and coordinates everything (see Figure 1). The circuit shown in Figure 2 has a Parallax SX-28 acting as a six-receiver port UART. Both parts will run on a 3.3-V supply, so I didn't need both a 5- and a 3.3-V source. The power is supplied by a Texas Instruments TLV1117 linear power regulator in a SOT-223 surface-mount package. Extra PCB copper was used to provide heatsinking. This was especially important since the input voltage to it was 12 V, which meant dissipating more power

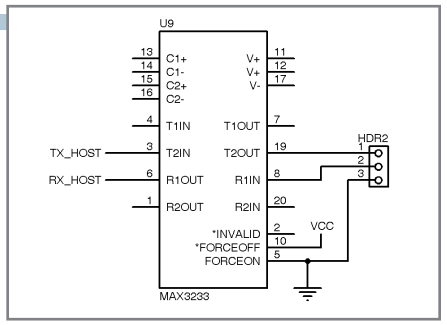

Figure 3—A MAX3233 provides conversion from RS-232 level to 3.3-V logic without requiring external capacitors for the charge pump.

in the form of heat. Signal conversion from logic to RS-485 levels was handled by a pair of Maxim MAX3094 quad receivers and a MAX3070 transceiver. A MAX3233 chip provided logic to RS-232 level conversion, and required no outboard capacitors for the charge pump, reducing component count (see Figure 3).

I spent a bit of time working out the best approach for clocking the two CPUs. The SX can be run up to 75 MHz, but the Z8 tops out at 20 MHz.

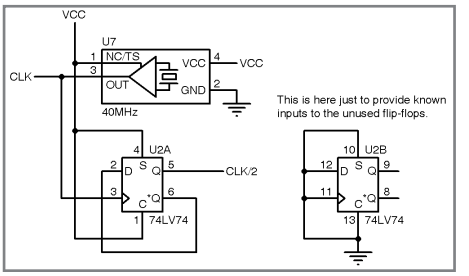

Figure 4—The project needs a 40-MHz clock for the SX and a 20-MHz clock for the Z8.

I wanted to run the Z8 around 20 MHz for maximum performance, but if the SX ran at that speed, it would have insufficient speed to run all six UARTs in software. I also wanted to keep power consumption down where I could, so I didn't want to run the SX at too high a speed. After writing the code for the SX, I could determine how many cycles the interrupt took. Using this information, I decided to run a 40-MHz master clock, which feeds the SX directly, but pass it through a simple divide by two flip-flop circuit to give the Z8 a 20-MHz clock (see Figure 4).

The SX was the perfect chip for me to create a six-port UART. In my design, I run the SX with an interrupt rate of 307.692 kHz. When you're running the interrupt at over 300 kHz, you know you're using a fast chip. The interrupt code executes six software UARTs sequentially, each with an 8-byte ring buffer, and each operating at 4× oversampling to detect the start bit properly. The main code running in the foreground monitors the ring buffers. When a buffer for a UART has added a byte to its ring buffer, the foreground code sends it to the Z8 using an 8-bit-wide parallel transfer using a lightweight handshake scheme.

The handshake scheme simply uses an input and output bit on each processor that is cross-wired (input to output) between the two chips. The scheme doesn't use the absolute value of the bits, but instead whether a bit has changed state to indicate an event. At start-up, each processor's output bit is set to a known state. Each processor reads that initial state and stores it. It then polls the bit and compares the current state to the stored state. When a processor changes the state of its output bit, the other processor detects the

Listing 1—This code for the Z8 to SX data transfer handshake scheme runs on the Z8. It shows extensive use of macros to simplify code readability and ease of modification.

```
// This macro generates a bitmask for the passed bit number
#ifndef THIS_BIT
#define THIS_BIT( bit ) ( 1 << (bit) )
#endif // THIS_BIT

// Port E, bit 4 is data ready signal from the SX
#define DATA_RDY_IN_PORT PEIN
#define DATA_RDY_BIT 4
#define DATA_RDY_FROM_SX DATA_RDY_IN_PORT & THIS_BIT(DATA_RDY_BIT)

// Port E, bit 5 is the acknowledge output bit
#define ACK_OUT_PORT PEOUT
#define ACK_BIT 5

#define ACK_TOGGLE            ACK_OUT_PORT ^= THIS_BIT(ACK_BIT)

   char CurrDataReady; // Current state of data ready line from SX
   char PrevDataReady = 0; // Previous state of dtat ready line from SX

   while(1){
      CurrDataReady = DATA_RDY_FROM_SX; // Get state of ready line from SX
      // Has the SX Data ready line changed state?
      if (CurrDataReady != PrevDataReady){
         ACK_TOGGLE; // Yes, so tell SX we got the data
         PrevDataReady = CurrDataReady; // Note state for next time
      }
   }
```

change by comparing the new input state to the previously stored state. This new state is then stored, and the process repeats.

Since data flows unidirectionally from the SX-based UART to the Z8, the SX changes state on its output bit when it has a byte to transfer. The Z8 sees the change, reads the byte, and then changes the state on its output bit. This change from the Z8 lets the SX know it is safe to put a new value on the 8-bit transfer port as soon as one is ready. Using some simplified test code, I clocked the two processors sending data synchronously at 186 KB per second. The code snippet in Listing 1 shows the C language code used by the Z8. Note the extensive use of macros. Since talking to ports, setting bit directions, and other actions are very hardware-specific, I used macros to isolate

the logic from the chip-specific details. If I move to a completely different processor, I can update the code by changing only the macros. These changes are then propagated to the rest of the code during compiling, rather than manually hunting through the source code for the hardware-dependent sections. It also helped me when I moved from the development hardware to the final hardware and completely changed the port and pin assignments in the process. The macros also make the working part of the code incredibly easy to read.

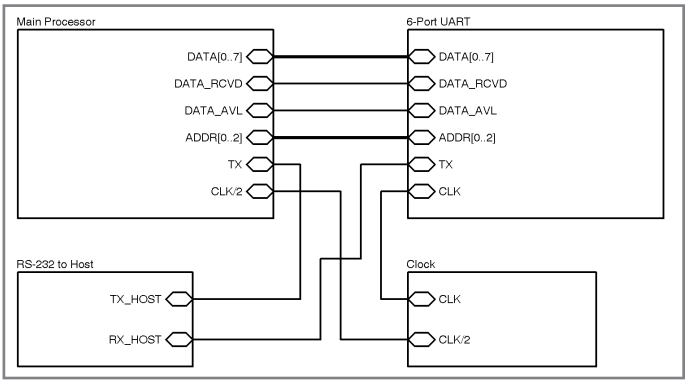

Figure 5—Top-level connections for the serial network hub

The Z8 was a great chip to use as the CPU in charge. With 48 KB of EEPROM and 4 KB of RAM, I had plenty of room for code and buffering. It turns out that the code for the Z8 barely made a dent in the EEPROM, using only 3 KB. However, it wasn't really an option to use a smaller part as the smaller Z8s have less RAM, and RAM was something I wanted to have plenty of. The Z8 has to retrieve each byte of data from the SX and store it internally in a ring buffer dedicated to that UART. It then monitors each ring buffer to see if it has a completed packet. When it finds a complete packet, the Z8 uses one of its hardware UARTs to send it to the laptop. Since I wanted to give each UART a 256-byte buffer, the RAM needed for buffering alone took up 1.5 KB.

CONSTRUCTION

Just as I did with the Halloween Remote, I wanted the serial network hub to look as good as it worked (see Photo 1). Looking good meant having an attractive enclosure with well designed and machined panels. For the enclosure, I chose a small extruded aluminum unit from Hammond. It's a black anodized enclosure that's only 4″ × 4.75″ and 1″ tall. For the front and rear panels, I used Front Panel Express once again. Their software is quite easy to use, and the total cost for the front and back panels was less than $75. This was for a pair of blue anodized panels with custom machined openings for the various connectors and switches, as well as yellow text and a skull logo I designed in white.

The power supply for the hub was something I didn't design. I needed the smallest power supply I could find. I selected a MEAN WELL Enterprises NFM-20-12 switching power supply that provides 12 V at 1.8 A in a board 0.75″ tall and only 3.5″ × 2″ in size. I wanted

to make sure this was a safe device to operate, so I used a breaker in line with the power supply. Oddly, this was one of the trickier elements to find. Most small panel breakers have a tiny face but are over 1″ long. Since I was trying to cram this project into a small case, the typical breaker would collide with the power supply. I finally found a small breaker (made by E-T-A Circuit Breakers) that was wide behind the panel but not deep.

Photo 1—The board in the middle is the MEAN WELL power supply, while my PCB with a six-port RJ-45 jack is on the right. Note the attention to detail on the internal wiring.

The PCB layout was a bit tricky. While the board needed to be the full 4″ wide to fit in the slots of the extruded aluminum, the length was another story. The power supply took up 2″ of the space inside the enclosure, while the switches, breakers, and power input took up almost another 1″, leaving a maximum of about 1.75″ for the length of the PCB. Complicating things further was the fact that the RJ-45 jack required 0.69″ of space on the PCB, thus cutting my room for components down to about 1″.

On the plus side, I didn't have a ton of components. On the negative side, I didn't have a lot of room. The solution? Surface-mount components and a generous use of both sides of the board for component layout. Rather than the typical component-side/solder-side style of board layout, I put components on both sides of the board as needed. I kept the top side populated with the through-hole-only devices, such as the jacks and headers, as well as the surface-mount processors and linear power regulator parts. The bottom of the board was for all the RS-485 interface chips and terminating resistors. So, I had a finished PCB. The result was tricky to lay out, but worth it for the space savings it generated.

PCB LESSONS

I have used the SX-28 chip in numerous projects, but it has always been the through-hole DIP package. When I decided to make this PCB an all surface-mount design, I checked the documentation for the SX-28 to see if the pinout was the same for the DIP and surface-mount packages. In the documentation I read, the pinout was identical for both the DIP and SOIC packages. Simple. I

proceeded to lay out the board and then sent it to be manufactured. When I got it back and started adding parts, all the testing went great until I added the SX-28 to the PCB. Suddenly, the power LED was dim (indicating a heavy load) and the SX-28 got extremely hot very fast. I had overlooked a minor, but critical, detail. The SX-28 comes in two different surface-mount packages—SOIC and SSOP. While it's true that the DIP and SOIC packages use identical pinouts, I was using a chip in the SSOP package. Sadly, the SSOP has a very different pinout. I ended up having to correct the parts in the schematic and PCB software, modify the layout design, and then get a new PCB manufactured.

Another lesson learned involved the MAX3094 RS-485 quad receiver chip. On page 1 of the datasheet, it says: "When active, these receivers have a fail-safe feature that guarantees a logic-high output if the input is open circuit." This definitely made life easier since I didn't have to worry about what the chip would do when nothing was attached to the inputs of the chip. After I got the revised PCB, I proceeded to add parts to it and test it as I went along. Electrically, the board was solid, with no parts overheating or exhibiting strange behavior. I then downloaded the code into the SX-28 and Z8 Encore! chips and immediately discovered strange behavior.

If I plugged a device into the hub but the device didn't send any data, the SX-28 was seeing a stream of zeros flooding in on that UART. Some time spent debugging the hardware revealed that until the first byte was received, the MAX3094 receiver was stuck with the output low, causing the UART to view it as a stream of zeros. I reread the datasheet in detail and discovered (buried on page 10) that the super-handy "fail-safe" behavior actually fails if the input is terminated. Since it is an absolute requirement of the RS-485 spec that the last receiver is terminated and a hub is by default the last receiver, then all the RS-485 receivers in the hub exhibited this bad behavior.

It seems to me that calling something "fail-safe" which is actually guaranteed to fail simply by following the electrical spec of RS-485, kind of implies they need to come up with a different name to describe the "feature." I was shocked that this condition wasn't properly handled by the chip. Maxim provides a fix for this that requires adding two more resistors for each input, but that pretty much defeats the whole purpose of the "fail-safe" feature of the chip. Fortunately, since all my MAX3094 chips acted the same way, I was able to work around this bug using a firmware fix, but the correct solution will involve a new layout and PCB with the extra resistors added.

USING THE HUB

The hub performed completely up to my expectations. Not having to daisy chain every device for the display simplified the system layout and wiring. I used one of the ports to run a line to a far corner of the house. This let me plug in the remote without having to drag a monstrous cable behind me while using it. I am especially pleased with the look of the finished device. It looks like something commercially manufactured. Since every single packet has to pass through this device, failure would have caused the entire show to stop running. The strong physical construction combined with an internal power supply meant that the hub just chugged along all night, passing packets and keeping the show alive.

Peter Montgomery (PJMonty@csi.com) spent 12 years working as a visual effects supervisor on films such as Mortal Kombat *and* Ace Ventura: When Nature Calls *before becoming a director. He has directed dozens of commercials and made the transition to episodic television with The Disney Channel's* Lizzie McGuire. *Peter is self-taught in both programming and digital hardware design.*

PROJECT FILES

To download the code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2010/237.

SOURCES

NFM-20-12 Power supply
MEAN WELL Enterprises Co. | www.meanwell.com

SX-28 Proto Board
Parallax, Inc. | www.parallax.com

TLV1117 Linear power regulator
Texas Instruments, Inc. | www.ti.com

Z8F4821 Microcontroller
Zilog, Inc. | www.zilog.com

NEED-TO-KNOW INFO

Knowledge is power. In the computer applications industry, informed engineers and programmers don't just survive, they *thrive and excel*.

For more need-to-know information about topics covered in Peter Montgomery's Issue 237 article, the *Circuit Cellar* editorial staff highly recommends the following content:

—

Wireless Mobile Robotics
by Scott Coppersmith
***Circuit Cellar* 224, 2009**

Scott used an MCU, an embedded Ethernet board, and a wireless router in a robot control system. A webcam can transmit real-time pictures to a laptop. Topics: Robotics, Wi-Fi, Webcam, Motor Control

Go to: www.circuitcellar.com/magazine/224.html

—

MiniEmail
A Compact MCU-Based Mail Client
by Alexander Mann
***Circuit Cellar* 204, 2007**

This easy-to-build system uses an MCU and an Ethernet controller to monitor email. When mail arrives, you can respond via a standard keyboard. Topics: Network, E-mail, WinAVR, LCD, GCC

Go to: www.circuitcellar.com/magazine/204.html