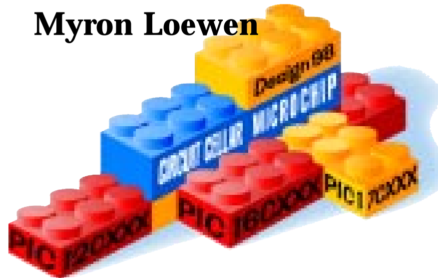
FEATURE ARTICLE

**Myron Loewen**




# Internet Appliance Interface

Internet appliances still aren't the most reasonable things out there. (Why pay hundreds more for a $20 toaster?) But, Myron uses a PIC and a 2400-bps modem to make an Internet interface that leads the way to less-expensive Internet appliances.

For years, we've been hearing about the promises of everything from coffee makers to lawn sprinklers being connected to the Internet for remote control and monitoring. Yet, none of these devices ever became a commercial product.

Two major roadblocks have prevented this dream from coming true—the cost of connecting to the Internet and the cost of an Internet terminal. No one wants to pay an extra $300 for a $40 appliance, plus $25 a month for the Internet service, just to have the ability to check on their home appliances from work.

But, things are changing in the market that could clear this roadblock forever. Cable modems are bringing continuous Internet connections into homes without the hassles of extra telephone lines and hourly billing.

Another recent change is free web access by local telephone numbers in some cities, if you can tolerate a little advertising. A dumb terminal like a coffee maker could care less how many banner ads it has to ignore. And the cost of the Internet interface is dropping to under $50 with the introduction of systems-on-a-chip complete with TCP/IP stacks.

In this article, I show how the Internet interface can be made even cheaper and simplified to run on a $2 PIC processor and a retired 2400-bps modem. The PIC dials and establishes a point-to-point protocol (PPP) connection with an ISP and exchanges data with remote servers. I don't have a cable modem or access to free web browsing yet, but there are many other applications that already need a small or cheap Internet interface.

Even though this project was conceived solely for the Circuit Cellar Design98 contest, it has since found use in everything from industrial controls to surveillance cameras. It's most suited for remote data collection, where the samples are stored until an alarm trigger point and then dumped to a central database through a local ISP. This has huge cost savings over leased lines.

My design, shown in Photo 1, uses the PIC12C672 to emphasize how small the device can be—only eight pins and under $2. Even with such a simple processor, I managed 2400-bps serial communications with the modem, three analog inputs, one digital output, and some of the Internet protocols.

The serial communications had to go through a software-emulated serial port (bit banging) because this device has no USART. The result was a demo Internet node with a remote-controlled red LED indicator and remote monitoring of three potentiometer settings with 8-bit resolution.

## SIMPLE INTERNET

Packing this kind of functionality into such tiny resources required a lot of tradeoffs. I studied a ton of Internet Request for Comment (RFC) documents, the public source code to Wat-TCP and Linux TCP/IP stacks, and *TCP/IP Illustrated* Volumes I, II, and III. Then, I whittled it down to the bare essentials, making many assumptions and forfeiting universal compatibility.

Here's a summary of how I implemented the Internet protocols and a description of the software. If you want to learn more, read a book like the one in the references and download

some of the Internet documentation listed in the Digging Deeper sidebar.

If you've read this far, you probably have quite an interest in TCP/IP protocols, but bear with me as I cover the basics. If it doesn't make sense to you, remember that this protocol stack is far from conventional. Try instead to focus on the flow of data instead of what software layers and state machines are missing.

I justify this reckless abandon of standards as necessary to shrink the code to the point where connections can barely be made with a majority of ISP servers. This will not endanger the standards of the Internet because these are end-user devices that perform no routing and are tweaked until they satisfy the end user. There are already inconsistencies between products from major brand names, which prevent the appliance from being able to log into some ISP servers.

Let's first look at how the Internet works and how data can cross such a maze of distant computers with varying physical connections and operating systems. Each computer gets a unique IP address, much like your mailing address. The data to be sent is broken into chunks that are stuffed in specially marked small envelopes that indicate the type of data (e.g., web page, e-mail).

Each of these goes into a medium-sized envelope, specially marked to get it to the right program on the remote computer. The type of data determines if the medium-sized envelope is a simpler UDP type or more robust TCP type. The TCP packet generates extra packets to open and close transmissions and resends packets that get lost.

The medium envelopes go into larger envelopes with source and destination Internet addresses on them. This is called the IP packet. It's like international mail; the address gets it to any destination on the Internet.

But, the Internet works more like passing notes in class. The large envelope goes in a bigger one with your friend's name on it. Your friend opens the big envelope, sees where the large envelope wants to go, and puts it into a new big one.

Your friend then passes it to another friend who is closer to the final destination, and the process repeats. The mail doesn't always take the same path and sometimes it gets lost along the way. With luck, the large envelope eventually ends up at its destination.

When it arrives, it is opened and the medium envelope is removed to see what program gets the data. That program then opens the little envelope to gets its data. Most OSs do this with a TCP/IP stack like WinSock. To save all the envelope handling, my algorithm puts on x-ray glasses and looks through all the layers for the data in the middle. The format of the envelopes or packets is shown in Figure 1.

There are protocols for the data following the IP header (e.g., ICMP for pinging, TCP for web browsing, and UDP for voice over Internet). You need to implement ping to test the Internet connection and send keep-alive packets to prevent the server from disconnecting. You also need a way to send data to the appliance and receive data back.

You're probably familiar with ftp for transferring files. It runs over TCP, which handles opening a high-level connection across the Internet with error detection and retransmission.

TCP has large RAM and ROM requirements to keep track of open connections and packets that have not yet been acknowledged by the remote computer. It turns out that there's another less popular file transfer protocol, called tftp, which runs over UDP. UDP is much simpler to implement than TCP because each packet is sent in response to the last one received and there is no retransmit buffer or table of connections to keep track of.

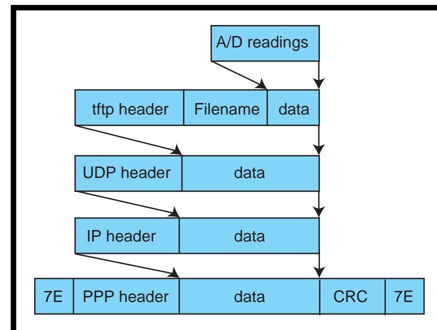



**Figure 1**—*The tftp packet shows how the A/D readings are wrapped in layers of packets to get a PPP packet.*

I couldn't choose tftp without client software for people to access their new appliances. With a quick search, I found several freeware, shareware, and demo tftp servers and clients for various OSs. The http links to these packages are available via the Circuit Cellar web site.

Another protocol I considered was SNMP. It is much more popular, has built in remote alarm monitoring and reporting, and lots of shareware clients, but is a little more complicated.

If you need it to be even simpler, just return data appended to pings. But then, you need to write a custom ping routine.

tftp has another advantage over ping and SNMP. It provides simple data logging of daily uploads on a central tftp server. tftp clients just have to upload files with unique filenames and they are stored on the server.

## NEGOTIATING PPP

A modem connection to the Internet is used because it's the most common method for remote data collectors. If we go back to the envelope analogy, the Internet appliance and the ISP

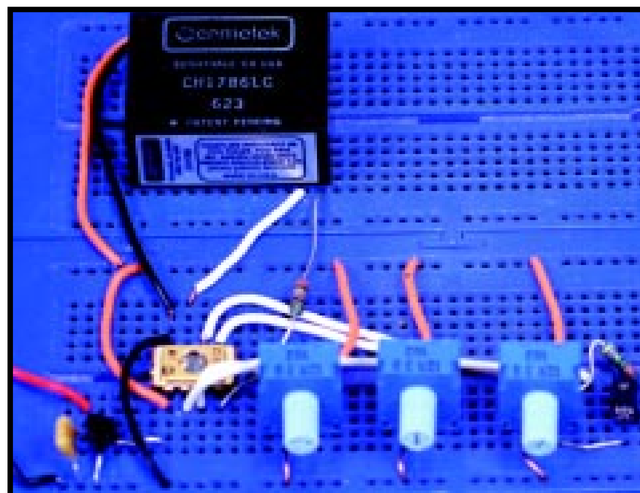



**Photo 1**—*The first prototype of the Internet Appliance uses three potentiometers for remote inputs and an LED to test remote control. The 2400-bps Cermetek modem provides a fast enough Internet connection to exchange the control packets.*

modem server are the close friends exchanging the big envelopes. The envelopes used in dialup connections come in two forms—SLIP and PPP. More acronyms are listed in Table 1.

I only had room for one protocol, so I chose PPP because some ISPs prevent SLIP from establishing a connection. The PPP connection can also go straight to a local server's serial port to save the cost of a modem.

PPP is a protocol to encapsulate IP packets on a serial link. On an asynchronous link (like the modem serial connection), it requires the data format to be 8 data bits with no parity.

The sample packet in Figure 1 shows that character 0x7e starts and stops a packet. All other instances of 0x7e must be changed to the two-byte sequence 0x7d 0x5e to prevent false starts. The character 0x7d becomes an escape character that means complement bit 6 in the following byte.

Any original instance of 0x7d, 0x7e, or bytes inclusively between 0x00 and 0x1f, must be changed to 0x7d followed by the original XORed with 0x20. This bit stuffing eliminates false packet breaks, false characters, and RS-232 control characters under 0x20.

The PPP connection procedure can be broken into several phases. First, if the link is dead, carrier detect is one of the stimuli that moves us to the next phase. The link establishment phase uses LCP to detect and negotiate link options with the remote computer.

Next, the authentication phase verifies your password using PAP. Although it is not one of the phases, this is where ISPs generally configure data compression with CCP packets.

The final phase is the network-layer protocol (e.g., IP). Each protocol is configured with its protocol; IPCP in the case of IP. Of course, there's also the terminate phase to close the link.

The LCP, PAP, CCP, and IPCP packets look similar and negotiate options in the same way. Only the protocol field and the meanings of the options are different. Figure 2 illustrates what an LCP packet looks like and how options are negotiated across the serial link.

Basically, the packet can request, deny, and accept options. Both sides must issue an accept before the LCP negotiation is complete. Figure 3 depicts the packet exchanges.

Negotiation starts with one side requesting a list of options in a REQ request packet. Each option consists of a length byte, option number, and option parameters. The other side responds with an ACK if it accepts all the options. If it doesn't like an option's parameters, it responds with a NAK and a list of the options that it rejected with parameters that would be acceptable.

If required options are missing, it adds those to the rejected list in the NAK reply. If some options are not recognized or are considered nonnegotiable, the other side should respond with a REJ reply and list the bad options.

The first side resubmits updated option lists until it gets an ACK reply.

## Digging Deeper

Anyone trying to build an Internet appliance will find that this article barely scratches the surface of the Internet protocols. Fortunately, the Internet is fairly well documented and, best of all, the standards are free in the form of request-for-comments documents (RFCs).

RFCs are the working notes of the Internet research and development community. RFCs can be written by anyone to introduce a new protocol, modifications, new methods, or explanations. They are often updated by later RFCs with higher numbers, so make sure you use the latest revision and refer back to updated RFCs.

You can find RFCs on several Internet sites including www.cis.ohio-state.edu/hypertext/information/rfc.html or by emailing:

To: rfc-info@ISI.EDU
Subject: getting rfcs
Message body: help: ways_to_get_rfcs

Here are some helpful RFCs to get you started:

| | |
|---|---|
| RFC 768 | UDP specification |
| RFC 791 | IP specification |
| RFC 792 | ICMP specification, updated by RFC 950 |
| RFC 867 | Getting time and date from the server |
| RFC 1055 | Serial link IP (SLIP) |
| RFC 1071 | Internet checksums |
| RFC 1144 | Compressing TCP/IP headers |
| RFC 1157 | Simple network management protocol (SNMP) |
| RFC 1332 | PPP Internet protocol control protocol (IPCP) |
| RFC 1334 | PPP authentication protocols (PAP) |
| RFC 1350 | tftp version 2 |
| RFC 1547 | PPP requirements |
| RFC 1570 | LCP extensions, updates RFC 1548 |
| RFC 1624 | Internet checksum via incremental update; updates RFC 1141 |
| RFC 1661 | PPP, the protocol itself; obsoletes RFCs 1548, 1331, 1172, 1171, 1134 |
| RFC 1662 | PPP framing, the CRC checksum; obsoletes RFC 1549 |
| RFC 1663 | PPP reliable transmission |
| RFC 1700 | Assigned numbers, parameters, and keywords |
| RFC 1962 | Compression control protocol (CCP) |
| RFC 1989 | PPP link quality monitoring; obsoletes RFC 1333 |
| RFC 1990 | PPP multilink protocol (LCP stuff) |
| RFC 1994 | PPP challenge handshake authentication protocol (CHAP) |
| RFC 2153 | PPP vendor extensions |
| RFC 2484 | LCP international extension |

The other side can start negotiating its options at any time. The resulting link may have different options for each direction.

I didn't implement the terminate, code reject, protocol reject, echo, and discard packets. The terminate packets can be replaced by disconnecting the modem. The code and protocol reject packets are possibly required to connect to updated servers in the future. The echo and discard packets are for testing the serial path and can be ignored for most ISP connections.

The only LCP option that I accepted and required was number three for authentication using PAP. Authentication with PAP is as simple as sending a PAP request with a user ID and password and then waiting for an acknowledge. I had to force this option to avoid the alternative (CHAP), which appeared more complicated and required more RAM and ROM.

The MRU option was omitted because, although my receive buffer was only 49 bytes, all the packets I used were small. On top of that, I didn't have enough RAM or ROM to reconstruct fragmented packets.

My ISP wanted to negotiate the character-map option to reduce the number of characters under 0x20 that had to be bit-stuffed and sent as two bytes. They also wanted to compress the protocol, address, and control fields. Although these would have been good at the low 2400-bps bandwidth, it was not worth the extra software.

The magic-number option may be required by a few ISP servers, but I ignored it because it needed four extra bytes of RAM and a lot of ROM. There are a lot more options, and this is definitely one area where you'll need to play with the code to make it more compatible with your ISP servers.

Once the LCP options are agreed on, the PAP authorizes your user ID and password as I described. Then, the CCP compression options are negotiated. These options compress the entire serial stream and could easily use up the entire memory space by themselves. Because my packets are tiny and traffic is low, I chose to disable them and go under the puddle-jumper option number three.

The final group of options configure the IPP settings with IPCP. I disabled the TCP/IP header-compression option to keep the software simpler and because I don't intend to transmit any TCP packets. I do, however, use this protocol to get the IP address of the Internet appliance. After this, only IP packets are sent and there is no harm in ignoring the server's rare LCP packets.

| | |
|---|---|
| ACK | acknowledgement |
| CRC | cyclic redundancy check |
| CHAP | challenge-handshake authentication protocol |
| ftp | file transfer protocol |
| ICMP | Internet control message protocol |
| IP | Internet protocol |
| IPCP | Internet protocol control protocol |
| ISP | Internet service provider |
| LCP | link control protocol |
| MRU | maximum receive unit |
| NAK | negative acknowledgement |
| PAP | password authentication protocol |
| PPP | point-to-point protocol |
| REQ | request |
| REJ | reject |
| RFC | request for comment |
| SLIP | serial line Internet protocol |
| SNMP | simple network management protocol |
| TCP | transmission control protocol |
| tftp | trivial file transfer protocol |
| UDP | user datagram protocol |
| USART | universal synchronous asynchronous receiver transmitter |

**Table 1**—*These are the acronyms used in this article. Check the Digging Deeper sidebar for where to find full descriptions and technical details.*

## IP PACKETS

The IP packets start with a 20-byte header followed by data. The data is either an ICMP, UDP, or TCP header followed by its data. The IP header directs the data to the destination and keeps multiple data packets from arriving out of order.

The first four bits are the IP version. The next four bits are the header length, always equal to 20 in this application. The 8-bit type of service field sets the routing priority to minimize delay and cost as well as maximize throughput and reliability. Then comes the total 16-bit length of the header plus data, about 40 for most of the Internet-appliance packets.

The 16-bit identification identifies each packet and is used with the following flags and fragment offset to reassemble fragmented packets. Following all that is the time-to-live byte, which sets the maximum number of hops this packet can be routed toward the destination before giving up. The next byte indicates what type of protocol is riding in the IP data.

Next, comes a 16-bit checksum of the 20-byte IP header. Be careful—this is a 16-bit one's complement checksum, not your ordinary math (described later). After that is the 32-bit source IP address and finally the 32-bit destination. You're probably used to seeing these addresses in a form like 10.97.123.67.

The easiest to understand protocol that rides the IP header is ICMP, which is used to ping Internet nodes. Named after the sonar method for locating objects, it sends out a packet and waits for a reply.

You need ICMP in this application to keep the Internet connection alive and respond to others that are looking for the health of your Internet node. The test is usually repeated several times indicating pass or fail and response time.

The ping packet is 20 bytes of IP header, then 8 bytes of ICMP header and some data to echo. The first two bytes of the ICMP header are type=8 and code=0 for the ping request. Type is 0 for the ping reply.

After that comes a 16-bit one's complement checksum of the ICMP header and echo data. Then, an identifier for multiprocessing systems and a sequence number increment each iteration.

The other protocol essential here is UDP, which is used to send data over tftp. Because of its simplicity, this protocol is used for everything from H.323 multimedia applications to SNMP network management.

Each output operation produces only one packet. Like IP headers, it can't ensure that the data gets to the destination, but it does check the data for errors.

The protocol simply takes the IP header, adds four 16-bit parameters and a string of data. The four parameters

are a source-port number, destination-port number, length, and one's complement checksum.

The port numbers identify which process gets the data in a multiprocessing system. The length is redundant with data in the IP header—eight for the UDP header plus the number of data bytes. You may want to bury your own CRC checksum in important data because UDP and TCP only use an inferior checksum.

TCP is the protocol for transferring web pages and e-mail across the Internet. TCP negotiates a connection, transfers the data, does error checking, retransmits bad or missing packets, and closes the connection.

A more complex protocol has to handle a lot of special cases and keep track of lots of data, which takes more RAM and ROM than is available in the PIC. But, it would be an interesting project to see just how small TCP could be implemented in another processor.

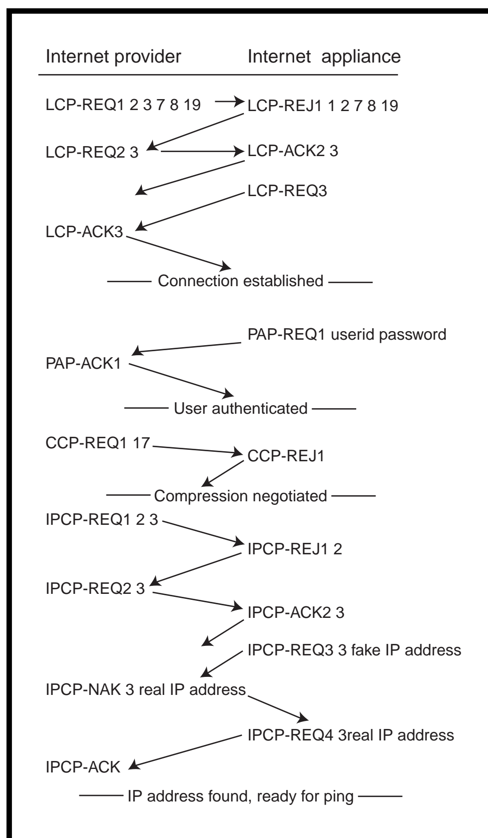



**Figure 3—***This is a typical exchange of option requests in the negotiation of a PPP connection. Each request (REQ) is numbered and the replying acknowledgment (ACK) or reject (REJ) must reference the same number. The list of numbers at the end of a packet identify each option as highlighted in Figure 2.*

## EXCHANGING DATA

I already indicated how data could be transferred using the echo data on the ping command and why I chose tftp instead. This section is a look at how tftp works and how the Internet appliance uses it.

tftp is intended for bootstrapping diskless workstations, so it is small and easily fits on a ROM. Each data exchange begins with the client asking the server to read or write a file. There are five packets used to transfer the data, Read Request (RRQ), Write Request (WRQ), Data, Acknowledge (ACK), and error.

The packet is laid out as 20 bytes of IP header, 8 bytes of UDP header, and the opcode for type of packet. Opcode 1 is for RRQ and is followed by a null-terminated filename, and a null-terminated ASCII or binary transfer mode. Opcode 2 is for WRQ and it has the same format as an RRQ packet.

Opcode 3 indicates a data packet and is followed by the block number and up to 512 data bytes. Opcode 4 is for ACK and is followed by the block number being acknowledged. Opcode 5 indicates an error and is followed by the error number and a null-terminated error message.

All data packets must have 512 bytes of data except the last packet in the file. A packet length under 512 bytes indicates the end of the file. Lost packets are detected when the sender has a time-out and the last packet is resent.

Just like ftp, there is no security. Instead, the tftp server usually limits transfers to files with world-read and world-write permissions.

For this Internet appliance, you want to read three analog inputs and set one digital output. There's even a way to do both in one operation.

Reading a filename that ends with a 0 clears the output. If the filename ends with a 1, the output is set; otherwise the output is unchanged. The returned file contains the output setting and the three A/D readings in ASCII format.

Many variations of this process can be implemented. If the output was a PWM analog output, the digit in the filename can be extended to

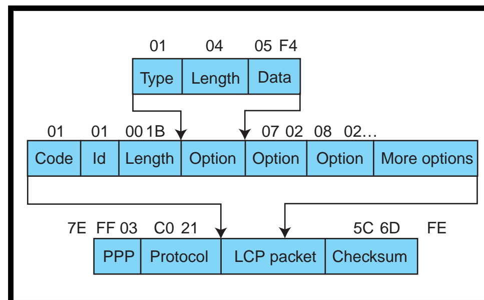



**Figure 2—***The LCP packet shows how the configuration options are sent in a PPP packet. This example highlights the MRU option.*

a three-digit value. The digits can be moved anywhere in the filename. The number of outputs and inputs is only limited by the transmit buffer size and maximum filename length.

The appliance can dump its data to the server by writing a file with a unique filename based on date. The server would have a directory of files logging the daily data of the appliance. The appliance doesn't need a real-time clock because the date and time can be retrieved from most Internet servers from UDP port 13 [1].

## THE HARDWARE

I chose to implement this project in the smallest processor I could find (the 8-pin PIC12C family) to emphasize how compact the code is. In the PIC12C family, I chose the device with the most RAM and ROM (at the time), the PIC12C672. Although it only has 128 bytes of RAM, that should be enough for the variables, a small transmit buffer, and a small receive buffer.

At first I was scared that the code wouldn't fit and it would force me to a larger processor. The 2 KB of ROM turned out to be plenty and left the possibility of a more robust PPP stack, EEPROM routines, SNMP routines, and other options. If I could start over, I'd choose a processor with a USART to simplify and speed up the serial I/O.

The other main component is the modem. I had plenty of old 2400-bps modems to choose from, but I went with the Cermetek for its small size (and because there were several laying around the lab from old projects). If you're not as lucky, you should still be able to locate a 2400-bps external modem. You could add RS-232 drivers to the circuit or, even better, bury the

whole circuit inside the modem case. The schematic for this project is shown in Figure 4.

Open the external modem case, remove the RS-232 driver chip, attach the circuit, connect the Tx and Rx pins to the correct driver chip pins, and connect power and ground. Be sure to stay away from the high voltages on the incoming power and telephone lines.

Also, use potentiometers that fit and mount nicely on the modem case. You won't need the resettable fuses and overvoltage limit components because the modem already has them.

I used 10-kΩ potentiometers from Tocos. These are nice for breadboarding because they have through-hole leads and a nice knob for days when you can't find your pot tweaker. You can use any LED to indicate the digital output, including one already in the case if you use the modified external modem.

The circuit is quite simple so I breadboarded it instead of waiting for a PCB. You could go even smaller and make it all surface mount.

Don't worry about finding the exact components. Any 1–10-kΩ potentiometer will do, as will any normal LED. You can even omit sidactor U3 and replace the resettable fuses with 5-Ω resistors. Note that this circuit is not FCC approved and thus should not be connected directly to your local telephone service.

## THE SOFTWARE

I wrote the software in C because the contest deadline was approaching. I intended to rewrite it in assembler to squeeze in some more options. After working with the AN555 serial I/O app note from Microchip, I decided that future implementations would have to have a serial port.

If you are going to experiment with this project, choose a flash-memory–based micro with a serial port and at least 500 bytes of RAM. The code should compile with most C compilers for any common microcontroller, so use one you're familiar with. Once you get it working, scale it down to a smaller processor with OTP memory.

I did all the PPP algorithm development on a laptop computer with Borland's Turbo C. It was great because I could use the internal modem, sprinkle `printf`s everywhere, and trace through the problem areas of code. This method worked so well that I logged onto the Internet after only two weekends of coding and debugging.

The software is set up to have the serial bit-banging routines running in the background and the IP state machine running in the foreground. There are only a couple of subroutines to transmit a serial string, calculate CRC checksums, create a new packet, check for config options, and remove a config option. You can check out the code details via the Circuit Cellar ftp site.

The software initializes the global variables and modem and redials every 30 s until it connects to the ISP modem bank. When `state` is set to 0 for no connection, the state machine takes over and loops forever, checking for received characters, transmitting the next queued character, generating reply packets, and initiating its own packets.

The state machine keeps the value of the current state in none other than the `state` variable. Another important variable is the `in` counter, which increments on every pass through the main loop. This variable is for triggering the appliance-generated packets and is zeroed so the packets will be retransmitted if the stack gets stuck in a state.

The IP state machine is initialized to state 0 after the modem connects with the ISP server. This state waits for an LCP packet from the ISP.

Getting an LCP packet moves it to state 1 (i.e., server detected). Also, getting an LCP request (REQ) packet means that the server is negotiating a PPP connection and it jumps to state 2. If the only ISP connection is requesting option 3, reply with an ACK; otherwise reject (REJ) the other options.

If you stay in state 0 too long, you send an LCP REJ to kickstart the ISP server into sending an LCP REQ. After you've been in state 2 long enough, make your own LCP REQ with no options. If you receive an LCP ACK packet, it means the server accepted your connection options and you enter state 3.

After entering state 3, send a PAP REQ packet with the user ID and password. If the password is rejected, the

state machine locks up in state 4. Otherwise, the ISP server sends a PAP ACK followed by a CCP REQ. If the CCP requests only option 3, it's accepted with a CCP ACK. All other options get rejected with a CCP REJ. The server will retry CCP REQ with reduced options until it gets a CCP ACK reply.

With CCP negotiated, the server attempts to negotiate IPCP with an IPCP REQ. Again, you negotiate the options down to number 3 using IPCP REJ and IPCP ACK. After you accept the server's options, the state moves into position 5. After waiting in state 5, the state machine generates its IPCP REQ to the server.

The server replies to the IPCP REQ with an IPCP NAK because you didn't know your IP address. The stack gets the right IP address from the NAK packet, updates its global address variables, and makes another IPCP REQ with a good IP address. When it gets the IPCP ACK from the server, it jumps to the final state.

In state 6, it sends out a periodic ping or tftp packet, depending on which line is REMed out. A tftp server can capture the packets to log the potentiometer positions. If the appliance gets an IP packet, it treats it as a ping and bounces back a reply.

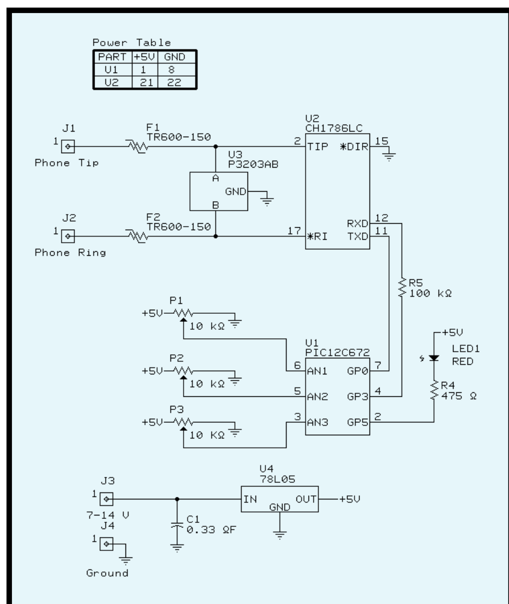



**Figure 4**—*This schematic shows the details of the prototype in Photo 1. These processor ports were chosen to maximize the versatility of the remote accessible pins. Pins AN1, AN2, AN3, and GP5 can be configured as digital inputs or outputs, or pins AN1, AN2, or AN3 can be 8-bit analog inputs. R5 limits the current to the modem during in-circuit programming.*

`MakePacket` creates an outgoing packet in the transmit buffer. Every loop of the state machine checks if the transmitter is ready for another character. If the transmit buffer has characters, it transmits the next character and resets the buffer on the final character in the packet, 0x7e.

Every loop of the state machine also checks the modem for received characters. Bit-stuffed characters are immediately converted from their two-byte form to the original character.

Some ISP servers compress the PPP control and protocol from 0x038021 to 0x21, others from 0x038021 to 0x0322. The state machine decompresses the control and protocol so the IP header always starts at the same buffer offset.

The CRC checksum is immediately calculated as the bytes come in so that there is no pause for a big calculation at the end of a packet. If the packets are longer than the receive buffer, the CRC is still calculated but the extra bytes are lost. Instead of calculating the CRC checksum over the final 0x7e, it stops a character early and should be 0xf0b8 instead of 0x0000.

`OptionTest` is true if the configuration option is in the string and no other options are present. `RemoveOption` removes the specified configuration option from the option list.

The rest of the code is pretty basic, but you'll want a fair understanding of the protocols before making modifications. A lot can be improved on, especially making it robust enough to connect to any PPP server. You may even be able to squeeze TCP into a tiny micro for the world's smallest web server.

More important are the end uses for this kind of technology. I chose to use the technology for low-cost remote data collection.

Because it uses a normal modem for dial out only, you should be able to string a hundred of them in parallel along a 5-mi. twisted pair. It would also be a simple way for utility companies to read residential meters. Maybe it'll even find use in everyday appliances.

I started off thinking the whole TCP/IP stack could be squeezed in and ended up with little more than the PPP negotiation, ping, and tftp. I was excited at how easily it worked with one ISP server but was frustrated by differences between other servers.

The books and RFCs have a lot of information, but some is hard to digest and some is hard to find. I hope this article provides the extra information that inspires new ideas or bridges a gap in your implementation of the IP and PPP protocols. ■

*Myron Loewen is a design engineer at Norscan Instruments, a leader in fiber-optic cable management systems. He enjoys simplifying complex problems to find the optimal solution. You may reach him at myron@norscan.com.*

## SOFTWARE

Source code and information regarding embedded Internet solutions and tftp resources is available via the Circuit Cellar web site.

## REFERENCES

[1] W.R. Stevens, *TCP/IP Illustrated*, Vol. 1, Addison Wesley, Reading, MA, 1994.

## RESOURCES

Microchip PIC12C672 datasheet, www.microchip.com/download/lit/picmicro/12c67x/30561a.pdf

Cermatek 1786LC datasheet, www.cermetek.com/pdf/ch1786%20rev%20o.pdf

## SOURCE

**PIC12C672**
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

**Modem**
Cermetek
(408) 752-5000
Fax: (408) 752-5004
www.cermetek.com

**Potentiometers**
Tocos
(847) 884-6664
Fax: (847) 884-6665
www.tocos.com