CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

Jim Brady

Build Your Own 8051 Web Server

Building your own web server can be a difficult task, especially if you proceed without proper direction and the right parts for the job. Fortunately, Jim has finished an 8051 server and he's eager to walk you through his project. With this tutorial, you can avoid common difficulties.

This article grew out of an experiment to see how hard it would be to build an 8051 web server and write a minimal TCP/IP stack. It seems like everything is serving web pages these days, so why not an 8051? It was not easy, but it was a fun project. After a few months of studying ARP and TCP, I had something up and running.

In this article, I'll explain how I built an 8051 web server and describe what I learned along the way. I'll also discuss timing and performance. If you want to follow along, download the source code from the *Circuit Cellar* ftp site.

COMPONENTS

I wanted an 8051 with enough RAM to hold a full-sized Ethernet frame of 1.5 KB, and with analog inputs so it could do something useful. The Cygnal parts were my first choice. The C8051F005 is fast, and it has a 12-bit A/D converter and 2.4 KB of RAM. The C8051F005's 32-KB flash memory is large enough for a reasonably sized program plus a few web pages. At first I thought its lack of a conventional bus would be a problem, but it turned out to be no problem at all.

The Cygnal 8051 makes up for being just 8 bits with its speedy 25-MIPS peak performance. The Ethernet controller's RAM adds additional buffering capability for incoming frames, which is key for allowing the CPU time to process a frame while more are received. Browsers running on fast machines can easily fire out two or three Ethernet frames within a millisecond!

For the Ethernet controller, I looked at the Realtek RTL8019AS and the Cirrus Logic CS8900A. The former is inexpensive and NE2000-compatible, but I've used many Cirrus parts over the years, so I went with the CS8900A.

The CS8900A's 4 KB of RAM is enough to hold a number of incoming frames. As with any Ethernet controller, the datasheet for it is long and there are many registers to set up. So, I sat down and read through the datasheet to figure out how to talk to it. By looking at a sample driver, which I downloaded from the Cirrus web site, I was able to create an interface in C, compile it to assembly, and then hand-optimize the assembly code.

So that's it, almost everything in two chips. I added an RS-232 port that runs at 115.2 KB for debugging, even though I found Cygnal's full-speed emulator to be more than adequate.

BENCHMARKING

The first thing I did when I got the Cygnal evaluation board was run the trusty sieve benchmark on it. [1] First,

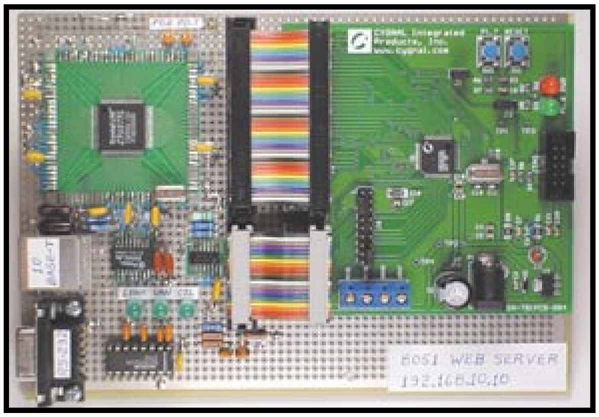


Photo 1—*The Cygnal C8051F005TB sits atop my breadboard at the upper-right. The top ribbon cable connects the 8051 CPU to the CS8900A Ethernet controller while the lower ribbon cable carries analog signals.*

I soldered a 22.1184-MHz crystal onto the board and wrote a function to make the CPU use it instead of the slower on-chip oscillator.

Most of its Cygnal 8051's instructions execute in one or two clocks, as compared to 12 or 24 clocks for standard 8051s. So, I expected good performance, and indeed the C8051F005 runs the sieve benchmark about 19 times faster than a standard 8051. It also ran faster than most 16-bit CPUs I've tested, which is impressive because much of the sieve is 16-bit operations.

To be fair, I should mention that the Cygnal 8051 has the advantage of running entirely out of internal memory. But the results are still representative of what I could expect of the various CPUs in this application. To run the sieve on the C8051F005, I had to scale it down to fit into RAM, and then scale the result to allow comparison to other CPUs. You can see the test results in Table 1.

CPU	Crystal (megahertz)	Sieve—10 loops (seconds)
Cygnal C8051F005	22.1184	0.43
Intel 80C51	11.0952	8.2
Intel 80C196 (16 bit)	18.4320	1.3
Philips XA-S3 (16 bit)	22.1184	1.0

Table 1—*The Cygnal 8051 is much faster than a standard 8051, and even faster than some 16-bit CPUs. The Keil compiler was used for the 8051s, Tasking was used for others, and a large memory model for all.*

BUILDING THE BREADBOARD

Photo 1 shows my breadboard with the Cygnal eval board mounted atop it. The board is part of the Cygnal P/N C8051F005DK package (about $99). It also includes an RS-232-to-JTAG interface to program the 8051's flash memory, IDE, and a full-speed emulator. My breadboard holds the CS8900A, associated Ethernet I/O, thermistor circuit, and RS-232 interface. To hold the 100-lead TQFP CS8900A, I used an RDI/Wainwright solder mount board.

I connected the evaluation board to my breadboard with two ribbon cables, one for analog and the other for digital. In order to keep the ribbon cables short to reduce cross talk, I cut off the prototyping area of the eval board. In addition, I added a 22.1184-MHz crystal and cut the trace that connected the crystal to the I/O connector. I did this to make sure noise could not get into the oscillator.

The Ethernet transformer and associated circuit design in Figure 1 was taken directly from the CS8900A datasheet. The eval board operates from 3.3 V, so I used the 3.3-V version of the CS8900A and the corresponding impedance-matching components for the 10BaseT interface. I kept wire lengths to a minimum in the area between the CS8900A and the RJ45 Ethernet connector.

A thermistor bridge circuit allows the ratiometric measurement of both power supply voltage and temperature. Using a ratio prevents 3.3-V power supply fluctuations from affecting the temperature measurement. After linearizing the thermistor characteristic, the firmware displays temperature on the web page.

Listing 1—*This code reads an Ethernet frame from the CS8900A. The Cygnal 8051 is so fast that NOPs must be inserted to meet the CS8900A worst-case access time.*

```
*****************************************************************
This fragment reads the incoming frame from the CS8900A.
Call from C as read_frame(UINT xdata * buf, UINT len).
R6 and R7 point to buf, and the length is in R4 and R5.
*****************************************************************
RSEG ?PR?_READ_FRAME?CS8900
_READ_FRAME:
MOV  DPL, R7               //Set up data pointer
MOV  DPH, R6
LOOP:
MOV  P1, #018H             //Take CS8900A CS low, set address
CLR  P1.3                  //Take CS8900A RD strobe low
NOP                        //Allow for CS8900A access time
NOP                        //Each NOP is 45 ns at 22.1184 MHz
NOP
MOV  A, P2                 //Read low byte into port 2
MOVX @DPTR, A
INC  DPTR
MOV  A, P3                 //Read high byte into port 3
MOVX @DPTR, A
INC  DPTR                  //Advance data pointer
MOV  P1, #038H             //Take RD strobe, chip select high
DJNZ R5, LOOP              //Decrement, loop again if needed
DJNZ R4, LOOP
RET
```

ETHERNET I/O

Listing 1 shows the assembly code that reads Ethernet frames from the CS8900A. Less you think this bit-banging approach is inefficient, consider that the Cygnal I/O speed is 40 ns while the maximum access time of the CS8900A is 135 ns. The CS8900A access time imposes the limit, not the 8051. I suspect this method of data transfer is at least as fast as conventional 8-bit bus I/O.

Figure 1 shows the interface between the CPU and Ethernet controller. 8051 port lines P1.0 to P1.2 select the CS8900 address. Only three lines are required because most of the CS8900's registers are indirectly addressed. Pulsing port pins P1.3 and P1.4 generates read and write strobes. 16-bit data is transferred in and out of ports 2 and 3.

My first design used the interrupt output of the CS8900A to interrupt the 8051 when an Ethernet frame arrived. The problem with this is that Cirrus Logic recommends reading everything out of the chip in the interrupt service routine. Because the CS8900A has more RAM than the 8051, I went with polling and only read the most recent frame. This way the CS8900A can queue a number of frames while the 8051 pulls them out and processes them one at a time. I configured the CS8900A to capture only the frames directed to my MAC address, plus broadcast frames. If the 8051 had to deal with every Ethernet frame on a busy network, it would be in big trouble.

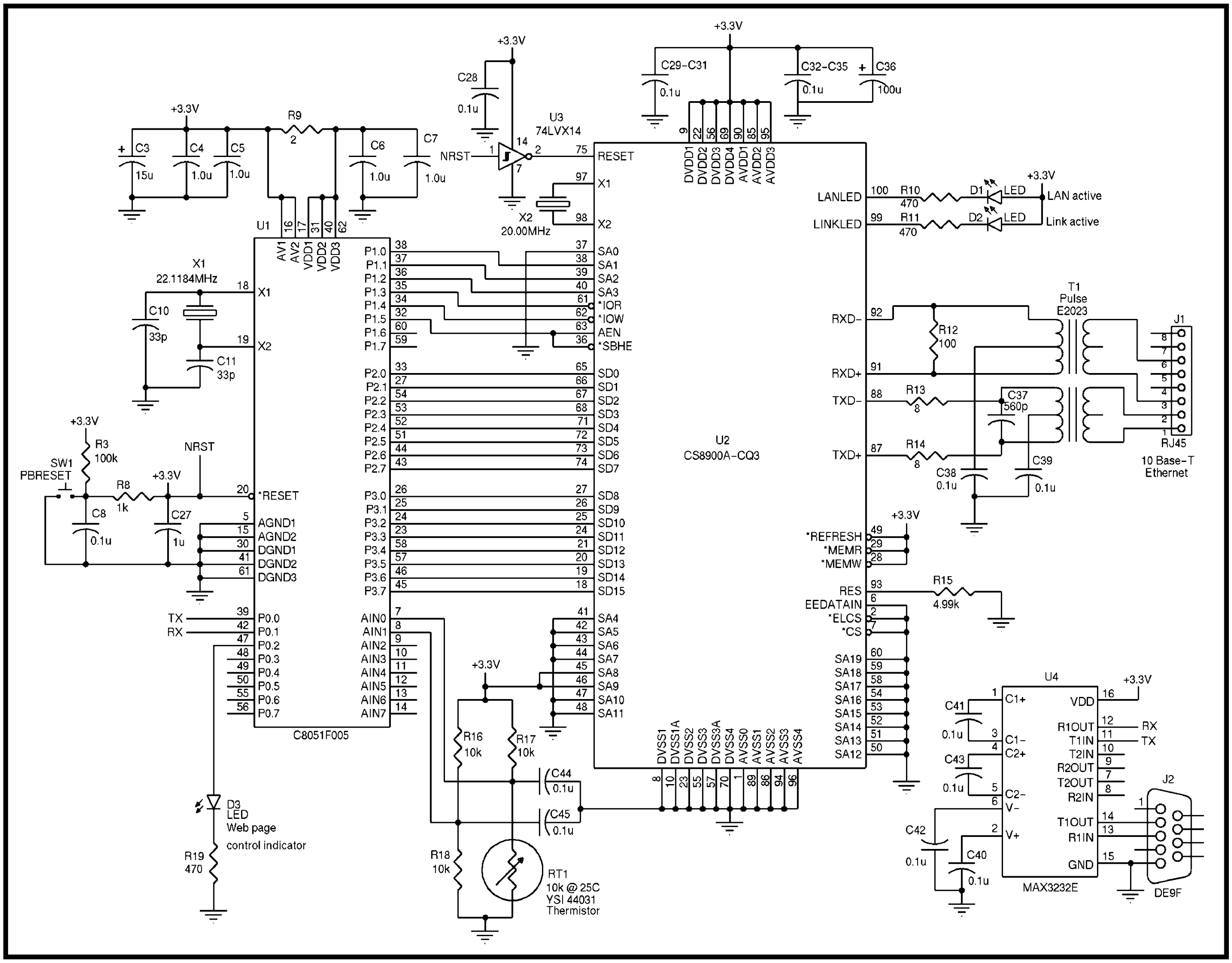
CAN IT BE DONE?

Can a CPU with only 2 KB of RAM really handle large Ethernet messages and the complexities of protocols such as TCP and ARP? Surprisingly, it turns out that even a few hundred bytes would suffice. Small RAM footprints work with TCP because you can tell the other end to limit the message size. TCP can, for example, advertise a maximum segment size of only 100 bytes.

TCP checksums are computed over the entire TCP segment and placed near the beginning of it. This makes it desirable to hold the entire segment in RAM to compute its sum. But here again, if the segment is small, this is not a problem. You can also handle IP processing in a small memory space, as long as you do not try to reassemble fragmented incoming messages. And you do not need to, because the other machine's TCP layer will limit the size of the message it sends. Only UDP will send you large, possibly fragmented messages, and I'm not trying to support that here. To serve a web page, the only other protocol you need to worry about is ARP.

A web server must handle ARP requests because you will receive them when the other end wants to find out your hardware address. You also need to originate your own ARP requests in order to find out the hardware address of the device you are sending to. Of course, you could simply reply back to the same hardware address that sent you the frame, but this would only work for a server (not a client) on a local network. In addition, this would get complicated with multiple simultaneous connections.

ARP is a simple protocol that can cause big problems for a small server. An ARP request must be sent, and a reply received, while a regular message is waiting to be transmitted. One send buffer no longer suffices. You need more space (i.e., enough RAM to hang on to the message-in-waiting), and you must buffer the outgoing and incoming ARP messages. It's a good thing that ARP messages are only 64 bytes long. Figure 2 shows how ARP fits into the flow of things, as well as the flow of an Ethernet frame through the various protocol layers.



+3.3V
R9 2
C3 15u
C4 1.0u
C5 1.0u
C6 1.0u
C7 1.0u
C28 0.1u
U3 74LVX14
NRST
X2 20.00MHz
U1 C8051F005
AV1 AV2 VDD1 VDD2 VDD3
X1 22.1184MHz
C10 33p
C11 33p
X1
X2
R3 100k
SW1 PBRESET
R8 1k
C8 0.1u
C27 1u
*RESET
AGND1 AGND2 DGND1 DGND2 DGND3
TX
RX
P0.0 P0.1 P0.2 P0.3 P0.4 P0.5 P0.6 P0.7
P1.0 P1.1 P1.2 P1.3 P1.4 P1.5 P1.6 P1.7
P2.0 P2.1 P2.2 P2.3 P2.4 P2.5 P2.6 P2.7
P3.0 P3.1 P3.2 P3.3 P3.4 P3.5 P3.6 P3.7
AIN0 AIN1 AIN2 AIN3 AIN4 AIN5 AIN6 AIN7
D3 LED Web page control indicator
R19 470
R16 10k
R17 10k
R18 10k
C44 0.1u
C45 0.1u
RT1 10k @ 25C YSI 44031 Thermistor
U2 CS8900A-CQ3
RESET X1 X2
SA0 SA1 SA2 SA3 *IOR *IOW AEN *SBHE
SD0 SD1 SD2 SD3 SD4 SD5 SD6 SD7 SD8 SD9 SD10 SD11 SD12 SD13 SD14 SD15
SA4 SA5 SA6 SA7 SA8 SA9 SA10 SA11
DVDD1 DVDD2 DVDD3 DVDD4 AVDD1 AVDD2 AVDD3
DVSS1 DVSS1A DVSS2 DVSS3 DVSS3A DVSS4 AVSS0 AVSS1 AVSS2 AVSS3 AVSS4
C29–C31 0.1u
C32–C35 0.1u
C36 100u
LANLED
LINKLED
R10 470
R11 470
D1 LED LAN active
D2 LED Link active
RXD– RXD+ TXD– TXD+
R12 100
R13 8
R14 8
C37 560p
C38 0.1u
C39 0.1u
T1 Pulse E2023
J1 RJ45 10 Base-T Ethernet
*REFRESH *MEMR *MEMW
RES EEDATAIN *ELCS *CS
R15 4.99k
SA19 SA18 SA17 SA16 SA15 SA14 SA13 SA12
U4 MAX3232E
C1+ C1– C2+ C2– V– V+ VDD GND
R1OUT T1IN T2IN R2OUT T2OUT R2IN T1OUT R1IN
C41 0.1u
C43 0.1u
C42 0.1u
C40 0.1u
J2 DE9F

Figure 1—*The left side shows part of the Cygnal C8051F005TB target board. On the right is my CS8900A Ethernet interface, temperature-sensing bridge, and RS-232 port.*

The firmware is written so that the functions in each protocol layer are concerned only with that layer.

WHAT DO YOU NEED?

My web page includes a JPEG image, which the browser loads after the HTML portion. Most of the browsers I tested establish a single connection to load both parts of the page. Netscape 4.7, however, opens two separate connections to the server, and the loading of the HTML page and image overlap to some extent.

Each connection comes from a different port on the browser's machine. To handle this situation, I put all connection-specific information, such as client IP address, client port number, sequence number, ack number, and TCP state into a structure that's indexed by the connection number. Each element of this structure can be thought of as a connection.

When a TCP segment arrives, I check to see if it matches an existing connection and use the state information for that connection. That allows the web server to handle simultaneous connections from the same PC or from multiple PCs.

Beyond ARP, IP, and a TCP state machine to handle simultaneous connections, what do you need, and what can you safely leave out? TCP has a long list of goodies, such as algorithms to estimate the best time-out, avoid congestion, assemble out-of-order segments, and so on. I made a list of about 20 such items. The truth is that I am serving web pages to various browsers and multiple PCs simultaneously, but I've implemented only a few of these capabilities. So, I am content for now, and I have a 20-item to-do list for a rainy day.

EMBEDDED WEB PAGES

My program uses 22 KB out of the 32-KB on-chip flash memory, leaving enough room for my 7-KB web page. Access to on-chip flash memory is fast. For more storage, the obvious choice would be an SPI serial EEPROM. 64-KB devices are inexpensive, and they can be clocked in the 2- to 5-MHz range, depending on the part. Using the built-in SPI port on the C8051F005, set to provide a 2-MHz SPI clock, my 7-KB web page could be transferred into the 8051 in about 30 ms. This would add about 50% to the time needed to serve the page.

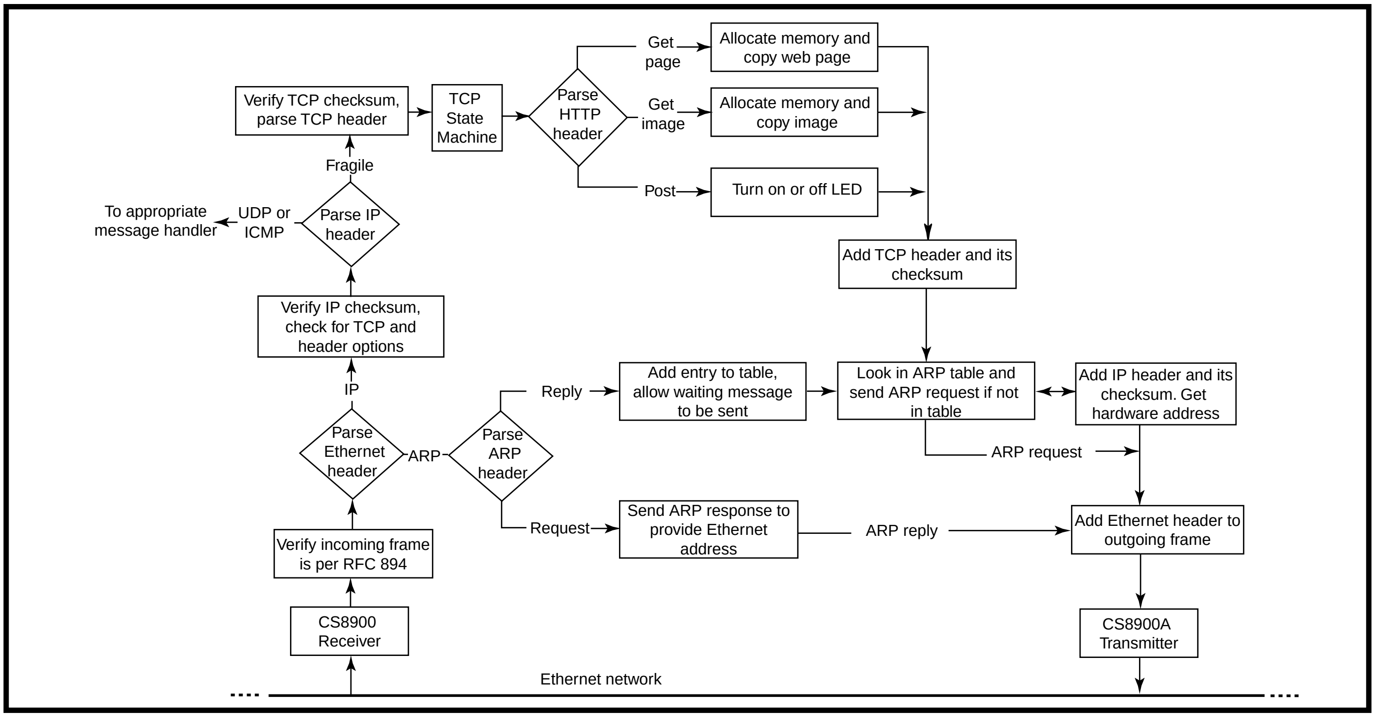
Web pages for embedded systems need capabilities that simple static web pages do not, such as dynamic data and two-way capability. Dynamic data is handled by inserting a tag in HTML. A tag number is included to tell the server what variable to insert. Two-way communications is provided using a form, which is a standard HTML construct that allows the browser to send selections back to the web server.

One purpose of a small web server like this is to make a product easy to use. A little eye candy is nice, and web pages for embedded systems would do well to have one or more images to make them attractive.

Photo 2 shows my web page, as presented on an MSIE browser. This page is bidirectional in that it both displays data and has radio buttons to turn an LED on the breadboard on and off. The state of the buttons is sent to the web server in a post message and can be easily extracted. The 0.8-KB HTML portion of the page and 6.2-KB JPEG graphic combine for a total of 7 KB. This page is used as the basis for comparisons later on in the article.

RUN-TIME PROFILING

Web pages are for human consumption, and 100-ms response times appear



Get page
Allocate memory and copy web page
Verify TCP checksum, parse TCP header
TCP State Machine
Parse HTTP header
Get image
Allocate memory and copy image
Fragile
Post
Turn on or off LED
To appropriate message handler
UDP or ICMP
Parse IP header
Add TCP header and its checksum
Verify IP checksum, check for TCP and header options
IP
Reply
Add entry to table, allow waiting message to be sent
Look in ARP table and send ARP request if not in table
Add IP header and its checksum. Get hardware address
Parse Ethernet header
ARP
Parse ARP header
ARP request
Request
Send ARP response to provide Ethernet address
ARP reply
Add Ethernet header to outgoing frame
Verify incoming frame is per RFC 894
CS8900 Receiver
CS8900A Transmitter
Ethernet network

Figure 2—*As a message goes up the left side of the flow chart, two checksums are verified before reaching the TCP state machine. Web page requests generate an HTML page or image, and then two checksums are added going down the right side. ARP message flow is shown in the central portion of the chart.*

Task	Times run	Accumulated time (milliseconds)
Search and replace tags	6	23.8
Copy buffers	8	9.1
Write to CS8900A	11	4.6
Parse incoming HTTP headers	2	4.5
Compute checksums	38	4.4
Read from CS8900A	8	0.6
TCP state machine	8	0.4
Total time		47.4

Table 2—*As you can see, searching for and replacing tags consumes the most time. To serve a single web page, 19 frames are sent and received.*

snappy. Using the Finisar (formerly Shomiti; go to www.finisar.com for more information) Surveyor Lite, I measured the time required for my 8051 web server to serve the 7-KB web page to a 500-MHz Pentium machine running MSIE 5.5. [2] It took 60 ms. In contrast, it takes my 100-MHz Pentium server running Apache about 32 ms to serve the same page. This demonstrates that the response of the 8051 is respectable. For these measurements, I had to clear the browser's page cache each time to make sure my browser was actually transferring the web page rather than just displaying a cached version.

To figure out how much time my web server spent doing various tasks, I added debug code that set a port pin when the CPU began the task and cleared the pin when it completed the task. Because many of the tasks I was interested in run a number of times while serving the web page, I had to add up all the pulse on times. This was hard to do with a oscilloscope, so I used an 82C54 timer chip. The 80C51 port pin output drives the 82C54 gate input. When in the high state, the 82C54 counts transitions from a 1-MHz oscillator. This provides accumulated pulse-width times with 1-µs resolution. I set up another 82C54 counter to count the number of times an event ran. Run times are summarized in Table 2.

The total of 47.4 ms falls short of the 60 ms it takes to transfer the page. When I added up the intervals between the 8051 sending an Ethernet frame and the browser's 500-MHz Pentium responding, I came up with 8.5 ms. This accounts for most of the difference. It's mind boggling, but true, that the 8051 is waiting for a Pentium.

Searching for and replacing tags is the most time-consuming task. My web page uses tags as placeholders for dynamic values, such as temperature. When it serves the page, it searches for these tags and then replaces them with the appropriate value.

It turns out that the `strstr()` function is the time hog. After some investigation, I found this to be true in general of `strstr()`. This makes sense because it has to parse through a lot of text, comparing each letter of the text to the corresponding letter of the search string. It may have many partial matches before it finally finds a complete match. One way to speed up the process would be to tightly limit the search range of `strstr()`. Another approach would be to keep an index of offsets to the tags, but the index would need to be changed each time a page was added or modified.

The second most time-consuming task is copying the web page from flash memory to RAM, using `memcpy()`. Why not just skip this step and copy directly from flash memory to the CS8900A? Again, the tags are the problem; they need to be replaced with actual values, and you can't replace them while in flash memory. Perhaps a faster approach would be to copy directly from flash memory to the CS8900A, looking for tags as you copy. But then you would have a thorny problem with the TCP checksum. It's computed over the entire segment, but must be inserted at the beginning of the segment.

Description	Code space
TCP/IP	9.5 KB
Web page including image	7.0 KB
HTTP server	3.8 KB
ARP	2.5 KB
C Library	2.9 KB
UDP	1.4 KB
CS8900A I/O	1.0 KB
RS-232	0.5 KB
Analog	0.3 KB
Priority task switcher	0.3 KB
Total	29.2 KB

Table 3—*Here you can see the footprints of various parts of the code.*

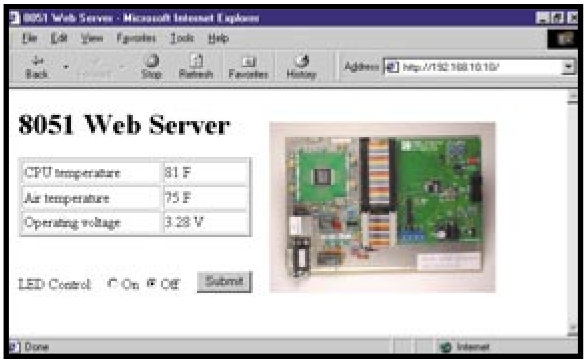
It's interesting to note that the checksum is computed a whopping 38 times to transfer a single web page. This transfer is made up of 19 Ethernet frames, 11 from my 8051 server and eight from the browser. It takes three frames to establish the connection, two frames to transfer the HTML page, eight frames to transfer the image, and six frames that are just acks. For both incoming as well as outgoing frames, two checksums are computed: one for the IP header and the other for the TCP segment, which makes 38 checksums. I was glad I used assembler for the checksum code!

I can't help but wonder how much a 16-bit CPU would speed things up, just by virtue of its being 16 bits. The checksum would certainly run faster because the sum is done over 16-bit chunks. Also, CS8900A I/O is 16 bits. Other tasks, such as memcpy() and strstr(), may need custom library code, because many 16-bit compilers default to doing these operations 1 byte at a time.

MEMORY USAGE

Of the 2.4 KB of RAM on the 8051, the lower 256 bytes are used for frequently accessed variables. The 2048-byte area of additional on-chip memory is addressed as XDATA memory. Incoming and outgoing message buffers are dynamically allocated from this space. Dynamic allocation is unusual for an 8-bit CPU, but it makes sense here because sometimes the incoming frame is large and the outgoing frame is small (occasionally, it can also be the other way around). At other times, the outgoing frame must be held in memory while an ARP message is sent and received. The firmware uses dynamic allocation using the library functions `malloc()` and `free()`, provided with the Keil C compiler. With this approach, no more RAM is tied up handing Ethernet frames than there needs to be at any given moment.

When it was all said and done, I had consumed 29 KB of the 32-KB flash memory. This reminds me of some-



8051 Web Server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Stop Refresh Favorites History
Address http://192.168.10.10/
Done
Internet

8051 Web Server

CPU temperature	81 F
Air temperature	75 F
Operating voltage	3.28 V

LED Control: ○ On ● Off Submit

Photo 2—*Seeing is believing! The web page and image served by the 8051 can be seen in the browser window. The on/off buttons control the state of an LED on the breadboard.*

thing I heard once about projects expanding to fill available space. The details are shown in Table 3. The TCP/IP portion is small as stacks go, but remember that I have implemented only a subset of TCP/IP here.

FUTURE DIRECTIONS

All things considered, I'm glad that I picked a part with 2.4 KB of RAM. A CPU with a few hundred bytes could do the job, but I wouldn't go through the trouble. It would be interesting to port the code to a 16-bit DSP chip and compare performance to this fast 8051. Most DSPs have enough RAM. It would also be nice to have a modular-size TCP/IP stack and create an API like the big boys have. This will surely require more than 32 KB, but lo and behold, 64-KB flash memory 8051s are already here. ■

Jim Brady is an embedded systems engineer living in southern Oregon. He has 20 years of experience designing with microcontrollers and device networks. You may reach him at jimbrady@aol.com

SOFTWARE

To download the source code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2002/146/.

REFERENCES

[1] J. Gilbreath and G. Gilbreath, "Eratosthenes Revisited: Once More Through the Sieve," *BYTE*, January 1983.

[2] E. A. Hall, *Internet Core Protocols*, O'Reilly & Associates, Inc., Sebastopol, CA, February 2000.

SOURCES

CS8900A Ethernet controller
Cirrus Logic, Inc.
www.cirrus.com

C8051F005 Mixed signal MCU
Cygnal Integrated Products, Inc.
www.cygnal.com

C compiler
Keil Software, Inc.
www.keil.com

RDI/Wainwright solder mount board
RDI/Wainwright
www.rdi-wainwright.com

RTL8019AS
Realtek Semiconductor Corp.
www.realtek.com.tw