www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#123 OCTOBER 2000

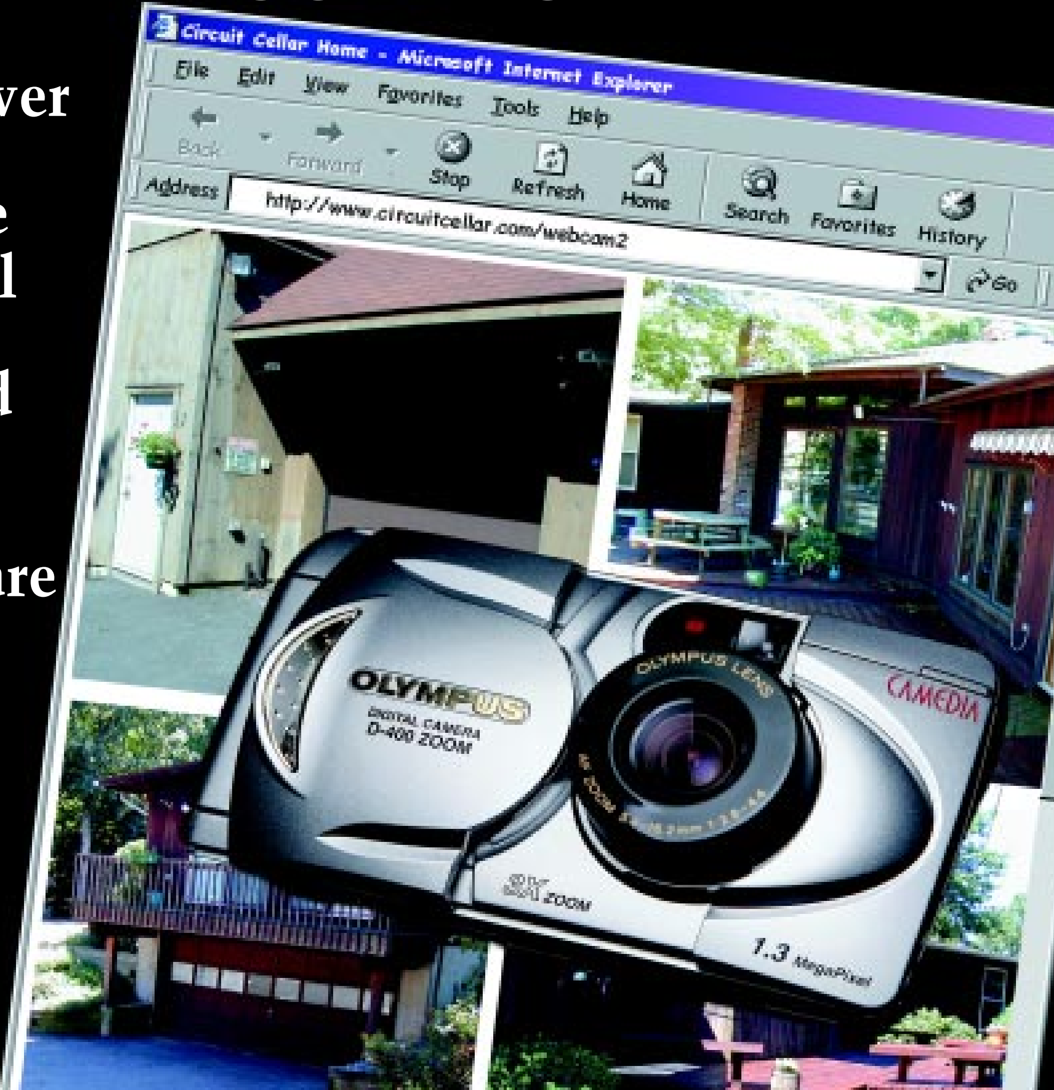
INTERNET AND CONNECTIVITY

A PIC Web Server

An Automotive Diagnostic Tool

Getting Around with GPS

PCI Bus Software


Circuit Cellar Home - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address http://www.circuitcellar.com/webcam2
Go
OLYMPUS
DIGITAL CAMERA D-400 ZOOM
OLYMPUS LENS
CAMEDIA
3X ZOOM
1.3 MegaPixel

CIRCUIT CELLAR ONLINE

THE MAGAZINE FOR COMPUTER APPLICATIONS

Circuit Cellar Online offers articles illustrating creative solutions and unique applications through complete projects, practical tutorials, and useful design techniques.

ASK US

THE ENGINEERS TECH-HELP RESOURCE

Let us help keep your project on track or simplify your design decision. Put your tough technical questions to the ASK US team.

The ASK US research staff of engineers has been assembled to share expertise with others. The forum is a place where engineers can congregate to get some tough questions answered, or just browse through the archived Q&As to broaden their own intelligence base.

★★★★★★★★★★

Test Your EQ
8 Additional Questions

RESOURCE LINKS
- **Content-Addressable Memory**
Bob Paddock
- **GPS Technology Overview**
- **GPS Manufacturers**
Rick Prescott

INSIDE ISSUE 123

Better Vision Through Landscaping

a few weeks ago I finally got sick of squinting to see road signs and went to have my eyes checked. During some small talk, the optometrist asked me how much time each day I spend in front of a computer. Between work and home, I figure that I average eight or nine hours per day. She gasped and then asked me if I had ever considered a career in landscaping. It seems landscapers spend an equal amount of time looking at close up things as they do far away things so their eyes get a healthy amount of exercise.

I consented that she had a good argument, but insisted that if she was a dermatologist, she would be telling me that landscapers spend an unhealthy amount of time in the sun. And, if she was a chiropractor, she would be telling me that shoveling snow and sand for 3–4 months (which is what most landscapers in New England do during the winter) is not good for the lower vertebrae. In the end, we both agreed that there was no easy solution, which is why I'm writing this editorial and not out mowing lawns.

There's no question, the growth of the Internet and technology industries has probably quadrupled the number of people who sit in front of computer screens all day, compared to even five years ago. But I wouldn't consider that to be a bad thing, especially from *Circuit Cellar*'s standpoint. It's been almost a year and a half since we started *Circuit Cellar Online* and in that time we've found that the Internet is a great medium for presenting some material and a poor medium for other material.

Having the online magazine has also allowed *Circuit Cellar* to broaden its perspective without changing the traditional look and feel of the print magazine. Where better to hold an Internet-based design contest than in an online magazine that is primarily read by engineers who are interested and involved in Internet-enhanced designs? By the way, if you still haven't seen the abstracts of the winning projects from the Internet PIC2000 contest, head over to ChipCenter and check out the archived issue for July. Or, for the majority of the winning abstracts, turn to page 12 in this issue and read through Edward Cheung's article about his prize-winning projects (now there's a guy who's into Internet applications!). Articles about some of the other winners will be appearing over the next few months, but the whole idea of having an Internet-related design contest sparked the idea of centering at least one issue of the print magazine on Internet-related material. So, this month we kick things off with the Internet & Connectivity theme.

In case you're one of the 10 people in the world who is not currently trying to Internet-enable a design, don't worry, there's still plenty for you in this issue. You can follow Jeff Stefan as he wanders through some of the details of GPS navigation, or you can go under the hood with Dan Harrison as he demonstrates how to design a diagnostic scan tool that will enable you to keep tabs on today's electronically-enhanced engines.

As you'll see, *Circuit Cellar* hasn't jumped headlong into the lemming race that has become the Internet era, it's more like a tentative first step toward something new. Besides, trying something new can be rewarding. In fact, ever since my visit to the optometrist, I've been printing everything in landscape mode and I think my vision has definitely improved.

Rob

rob.walker@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

MANAGING EDITOR
Rob Walker

TECHNICAL EDITORS
Jennifer Belmonte
Rachel Hill
Jennifer Huber

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Mike Baptiste Ingo Cyliax
Fred Eady George Martin
George Novacek

NEW PRODUCTS EDITORS
Harv Weiner
Rick Prescott

PROJECT EDITORS
Steve Bedford Bob Paddock
James Soussounis
David Tweed

ASSOCIATE PUBLISHER
Joyce Keil

CHIEF FINANCIAL OFFICER
Jeannette Ciarcia

CUSTOMER SERVICE
Elaine Johnston

ART DIRECTOR
KC Zienka

GRAPHIC DESIGNERS
Naomi Hoeger
Mary Turek

STAFF ENGINEERS
Jeff Bachiochi
Anthony Capasso
John Gorsky

QUIZ MASTER
David Tweed

EDITORIAL ADVISORY BOARD
Ingo Cyliax Norman Jackson
David Prutchi

Cover photograph Ron Meadows—Meadows Marketing
PRINTED IN THE UNITED STATES

ADVERTISING

ADVERTISING SALES REPRESENTATIVE

Kevin Dows	Fax: (860) 871-0411
(860) 872-3064	E-mail: kevin.dows@circuitcellar.com

ADVERTISING COORDINATOR

Valerie Luster	Fax: (860) 871-0411
(860) 875-2199	E-mail: val.luster@circuitcellar.com

ADVERTISING CLERK Sally Collins

CONTACTING CIRCUIT CELLAR

SUBSCRIPTIONS:
INFORMATION: www.circuitcellar.com or subscribe@circuitcellar.com
TO SUBSCRIBE: (800) 269-6301, www.circuitcellar.com/subscribe.htm, or subscribe@circuitcellar.com
PROBLEMS: subscribe@circuitcellar.com

GENERAL INFORMATION:
TELEPHONE: (860) 875-2199 FAX: (860) 871-0411
INTERNET: info@circuitcellar.com, editor@circuitcellar.com, or www.circuitcellar.com
EDITORIAL OFFICES: Editor, Circuit Cellar, 4 Park St., Vernon, CT 06066

AUTHOR CONTACT:
E-MAIL: Author addresses (when available) included at the end of each article.

For information on authorized reprints of articles,
contact Jeannette Ciarcia (860) 875-2199 or e-mail jciarcia@circuitcellar.com.

CIRCUIT CELLAR®, THE MAGAZINE FOR COMPUTER APPLICATIONS (ISSN 1528-0608) and Circuit Cellar Online are published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (860) 875-2751. Periodical rates paid at Vernon, CT and additional offices. **One-year (12 issues) subscription rate USA and possessions $21.95, Canada/Mexico $31.95, all other countries $49.95. Two-year (24 issues) subscription rate USA and possessions $39.95, Canada/Mexico $55, all other countries $85.** All subscription orders payable in U.S. funds only via VISA, MasterCard, international postal money order, or check drawn on U.S. bank.

Direct subscription orders and subscription-related questions to Circuit Cellar Subscriptions, P.O. Box 5650, Hanover, NH 03755-5650 or call (800) 269-6301.

Postmaster: Send address changes to Circuit Cellar, Circulation Dept., P.O. Box 5650, Hanover, NH 03755-5650.

NEW PRODUCT NEWS

Edited by Rick Prescott

PHOTOELECTRIC SENSOR

Meet a new mountable photoelectric sensor that measures only 35 mm × 15 mm × 31 mm: the **WORLD-BEAM**. Because its single housing style fits almost all mounting requirements, the WORLD-BEAM can replace hundreds of existing sensors, meet or exceed their performance, and use less space. It is ideal for volume OEM applications in all industries, including material handling, packaging, assembly, printing, and converting.

This sensor is available in all sensing modes. Its opposed-mode sensing range is 20 m, retroreflective mode range is 6 m, and convergent models have a 16- or 43-mm focus. Sensing modes also include regular and wide-angle diffuse (proximity mode) with ranges up to 450 mm. The sensor features versatile output configurations and self-diagnostics. Users can choose NPN (sinking) or PNP (sourcing) outputs, normally one open and one closed, each capable of switching up to a 100-mA load.

The WORLD-BEAM has a rugged ABS sealed housing that stands up to tough applications. The housing is leak-proof and meets the IP67 and NEMA 6 standards for harsh environments. Integral protective circuitry guards the sensors against reverse polarity and transient voltages, short circuits, and false pulse at powerup. The internal pot is isolated from the environment and sealed. LEDs protruding above the top of the sensor provide 360° visibility, keeping users informed of operating status from any angle.

The WORLD-BEAM photoelectric sensor is made to fit all mounting requirements. Models range from 10 to 30 VDC and cost **$61**.

Banner Engineering Corp.
(888) 373-6767
Fax: (763) 544-3213
www.bannerengineering.com

INERTIAL MEASUREMENT UNIT

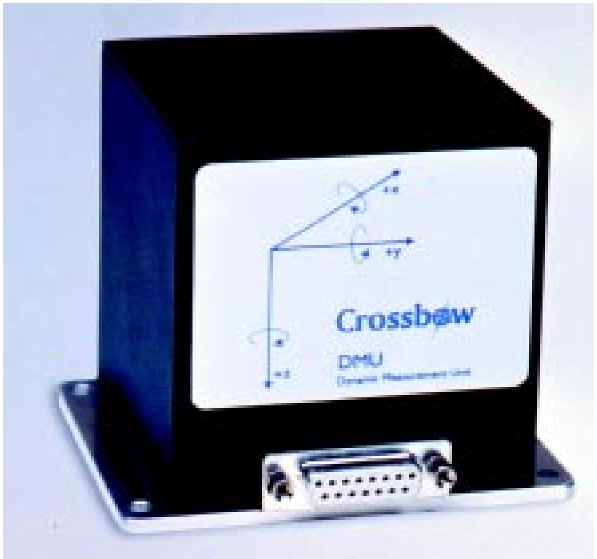
The **DMU-H6X** is a six-axis internal measurement unit (IMU) that provides six degrees of freedom (DOF) motion sensing. The unit is designed for demanding vehicle test environments in automotive, marine, and airborne dynamic environments.

The unit offers bias stability of <±1°/s and <10 mG, respectively over the full –40°C to 85°C range. It will accurately measure X-, Y-, and Z-axis acceleration as well as roll. The key to its performance is the integrated digital signal processor (DSP) coupled with proprietary algorithms that compensate the effects of temperature, non-linearity, and misalignment. The unit uses MEMS sensor technology to create a reliable, inertial solution.

It is designed with both analog and digital (RS-232) output signal formats for easy integration into a variety of data acquisition systems. Transfer of digital data is user configurable for either continuous or commanded (polled) updates.The X-View software accessory is included.

The compact DMU-H6X measures 3.0′ × 3.75″ × 3.25′ and weighs less than 20 oz. Pricing starts at **$3495** in single-unit quantities.

Crossbow Technology, Inc.
(408) 965-3300
Fax: (408) 324-4840
www.xbow.com



T-1 SIMULATOR

The Simulation Platform (TSP) is capable of both T-l and POTS simulation. Designed for maximum flexibility, users can mix and match the hardware modules, which plug into the TSP base chassis. Any size simulation can be built by connecting multiple TSP units to a single PC.

Through the Windows software interface, users can set up and store configurations for multiple test setups, enabling them to quickly change test configurations without reprogramming the unit. The TSP software also logs events and allows definition of pass/fail parameters based on event sequences to help automate testing.

The TSP platform adds T-l capability to Teltone's simulator product family. The TSP is a base platform on which to build new capabilities. Future plans for the TSP include software upgrades for E-l and PRI capabilities.

Depending on the user's mix of modules, list price starts at **$8200**.

Teltone Corp.
(800) 426-3926
Fax: (425) 487-2288
www.teltone.com

NEW PRODUCT NEWS

SUNLIGHT-READABLE FLAT PANEL MONITOR

The **Clarity-18 UHB** is a sunlight-readable 18.1″ flat-panel monitor that accepts analog, digital, and interlaced NTSC input, all on the same video controller board. The Clarity-18 UHB suits outdoor and high-ambient light applications, such as point of information, point of sale, and vehicular and plant automation. Security and medical monitoring systems could benefit from its ability to switch among analog, digital, and NTSC (VCR, camcorder, CCD camera) inputs.

The 18.1″ monitor has SXGA resolution, 16.7 million colors, 900-bit brightness, and a 160° viewing angle. For host computers with digital output, the Clarity-18 UHB's video controller board features the industry-standard PanelLink interface. The monitor is rated for 0 to 50°C and measures 20.4″ × 14.25″ × 3.5″.

The monitor can take lower resolution video and cleanly expand it to fill the 1280 × 1024 screen. The Clarity-18 UHB comes in a rugged metal OEM frame, ready for mounting in an enclosure. A front-mount plate and bezel, which gives NEMA 4 protection to the front surface, and a durable analog-resistive touchscreen are optional.

The Clarity-18 UHB costs **$5500** in quantities of 10.

Computer Dynamics, Inc.
(864) 627-8800
Fax: (864) 675-0106
www.cdynamics.com

READER I/O

MINDING Ps, Qs, AND Ns

The schematic on page 69 of the August 2000 issue identifies transistors Q2-7 as P/N 2N2907, and they are drawn as NPN. The text on page 70 also refers to these transistors as NPNs, however, P/N 2N2907 is a PNP transistor.

Ed Webb

Thanks, Ed! In the schematic, the box next to the six-digit display should read:

Q2–7 = 2N3906
Q8–15 = 2N2907

THE HEART OF THE MATTER

Another fine copy of *Circuit Cellar* arrived and I was scanning the articles to look for something to read when I came upon George Novacek's article "The Joys of Writing Software: Part 1" (*Circuit Cellar* 121).

In your attempt to refute the industry's attempt to define bugs as features, you managed to perpetuate the myth about dangerous cruise controllers in cars! The American propensity to blame someone else for their actions and duck responsibility through litigation almost destroyed the manufacturer of the Audi automobile. In each case of unintended acceleration, the party involved complained of complete brake failure, yet no problem was found with the brakes after the incident. There are some issues with the placement of the pedals in the car and an aggressive idle speed control loop, but the cruise control was never found to be a problem, except in a courtroom full of under-educated jurors.

Today, all new cars have a feature that requires you to place your foot on the brake before shifting the car into gear. (A habit that was taught to me by my driving instructor.) This feature probably costs ~$50 or more per vehicle, and if you multiply that times the more than 10 million vehicles per year, that's a large cost to the American consumer.

Go out to a large parking lot (empty), firmly apply the brake, and give the accelerator a good push. The brakes always win.

Matt Werner
Grand Rapids, Michigan

Thank you for your comment and you are not the first one to mention this particular issue. Not being a part of the automotive industry myself, I am only familiar with the legend. In the context of the article, however, I did not intend to criticize a specific car manufacturer or any industry. I was making a point that, in its infancy, software was viewed by many as a promising, get-rich-quick design approach, which could significantly cut the cost of hardware and its development and simplify its future modifications (listen to an FPGA salesman for a while and you will hear similar claims).

Before we learned that producing good software is serious business and definitely not cheap, we endured and paid for a few embarrassing problems. The cruise control and a lost satellite have been used for many years as the perfect examples of bad software design by instructors in software development training sessions, but I have never seen the actual piece of code containing the bug. However, none of those instructors ever claimed "wrong code" was to blame. As I show in the continuation of the article, the real "wrong code" (programmer-created bugs) accounts for a small part of the overall problem. More often than not, the problems such as the lost satellite or a runaway cruise control have been caused by EMI, an electrical transient, or some other environmental or system influence.

But, as we've seen, the software gets the blame and, in my opinion, rightly so. Making software perform the desired function is a small part of the design. Just about any designer should be able to do it.

On the other hand, well-designed software must be robust, able to properly handle exceptions, and no matter what the cause, revert to a safe, predictable fault condition. Making a system idiot-proof is very difficult, but it must be done. I can freeze just about any program on my PC by hitting a few wrong keys. In this case it is merely an inconvenience that requires me to flip the power switch, no big deal. But it still indicates weak software design—an avionic program would have to be able to handle such condition gracefully. We should expect no less from everyone else.

George Novacek

Editor's Note: on page 35 of the August issue, the bio paragraph for Sandeep Dutta should read:

Sandeep Dutta is a compiler engineer working for WindRiver Systems Inc. He works on the DIAB optimizing C, C++, and Java compiler for 32-bit processors. You can reach him at sandeep.dutta@windriver.com.

FEATURE ARTICLE

Edward Cheung

A Winning Combination

PIC Internet Connectivity

Edward's winnings in the Internet PIC2000 design contest turned out to be quite a treat for him, but now we're in for a treat of our own as he explains some of the tricks that went into designing these award-winning Internet projects.

When I entered the *Circuit Cellar* PIC2000 Contest sponsored by Microchip, I saw many possibilities. First, I developed the PIC Web Server, which needed a good application in order to come to life. I developed five Internet applications that could be hosted on the PIC. Four of these projects won—including the top three prizes.

Home automation has been my hobby for a long time. More than 10 years ago, I started with a collection of lamp and appliance control modules from X-10. I gradually added components to the system, including a central desktop computer running custom home control software that communicates with a network of PIC microcontrollers. The network uses the RS-485 standard, which is similar to HCS-II. It interconnects network nodes that do a variety of real-world input and output functions.

I can control the system from any phone, but I wanted to control it from the Internet. Ideally, I would add the Internet/Ethernet interface at a node on the RS-485 network, freeing the desktop computer for other tasks. The thought of a PIC communicating via the local Ethernet network was appealing because it could be used in other applications too.

In July 1999, *Circuit Cellar Online* published an article describing a microcontroller web server. [1] It was based on an Atmel processor interfaced to a standard NE2000-compatible PC Network Interface Card (NIC). The idea was brilliant: take a commonly available Ethernet interface card, use the microprocessor to emulate its bus interface, and you have a microcontroller Ethernet node. That was the proof I needed to create a PIC version.

My finished product is a web server that is roughly the size of two videocassettes. All it needs is power and a 10BaseT connection to provide any user on the local network with real-world interface to his web browser.

The first step in constructing a web server on a microcontroller involves knowing the protocols used on the Ethernet network. I wanted to build a web server from only web resources. It probably stretched the process, but it was more fun to learn my way from the tidbits scattered about the 'Net. In the end, I obtained an understanding of the entire process—from the top level where a user clicks in a URL box of a browser, down to the level of the voltage sent on the physical Ethernet wire.

WEB SERVER HARDWARE

The NIC uses the PC-ISA standard for its bus interface. The PIC emulates this bus so you can communicate and control it. All 16 bits of the data bus

Table 1—*Take a look at the Ethernet packet structure. Nodes on the local network can decode the recipient and sender's hardware address.*

Name	Length	Comment
Preamble	62 bits	Prepended by NIC
Start of frame (SFD)	2 bits	Prepended by NIC
Destination address	6 bytes	Ethernet address
Source address	6 bytes	Ethernet address
Type	2 bytes	Length data for IEEE 802.3
Data	46–1500 bytes	Least significant bit and byte first
Checksum	4 bytes	Calculated and appended by NIC

and 5 bits of the address bus are connected to the PIC. The remaining bits of the address bus are strapped either high or low to presume the base address of the NIC is at 0x300. The number of address lines allows the addressing of 32 registers in the NIC. Depending on the initial setup of the NIC, it may have to be plugged into a computer to disable plug-and-play and to set the base address to the desired value. In addition to these lines, the minimum set of ISA bus control lines is connected to the PIC (see Figure 1).

I chose the P16F877 as the microprocessor because of its flash program memory and the availability of the In-Circuit Debugger (ICD). The 8 KB of program space was just enough to fit the web server and project application code. A Maxim chip provides the voltage level translation needed for RS-232 (for the PIC Web Cam project), and two of the PIC's I/O lines are brought to an RJ-11 connector (for the Chart Recorder project and X-10 interface). There are no additional RAM or ROM chips, which meets my goal of a minimal circuit.

Name	Length	Comment
Type	1 byte	Eight for echo request, zero for echo reply
Code	1 byte	Always zero
Checksum	2 bytes	One's complement type
Identifier	2 bytes	Used for matching messages
Sequence number	2 bytes	Used for matching messages
Data/payload	Variable	Data to be echoed

Table 2—*The TCP packet is sent within an Ethernet packet.*

ETHERNET, FROM THE BOTTOM

At the lowest level, Ethernet is a party line shared by all the computers on the local network. The bits are sent Manchester encoded, which means the logical ones and zeros are not sent as voltage levels (as in RS-485), but as rising or falling edges in the middle of the bit time. In this manner, both data and clocking information is sent to the receiver.

By comparing the sent and received data, the NIC can quickly detect a collision and release the line. The senders then wait a random amount of time. After that, they retry and await confirmation that the data was sent without contention. The packets are separated by a pause cueing the other nodes to try sending their data. The first data sent is the least significant bit of the least significant byte. The bit time is 100 ns for 10BaseT and its variants, such as 10Base2.

The structure of the packets is shown in Table 1. The NIC used is compatible with the common NE2000. It has a 16-KB circular buffer to hold the Ethernet data for the host processor. Of the listed data, the preamble and the SFD are not routed into this circular buffer. This data is sent to bit synchronize other NICs on the network. The destination and source addresses at the start of the packet allow nodes to sort the routing of the packet. This 6-byte address is unique for every NIC made. If the most significant bit of the physical address is one, this message is a broadcast and should be processed by all nodes.

The two bytes in the Type field that follow indicate the kind of packet. All data up to

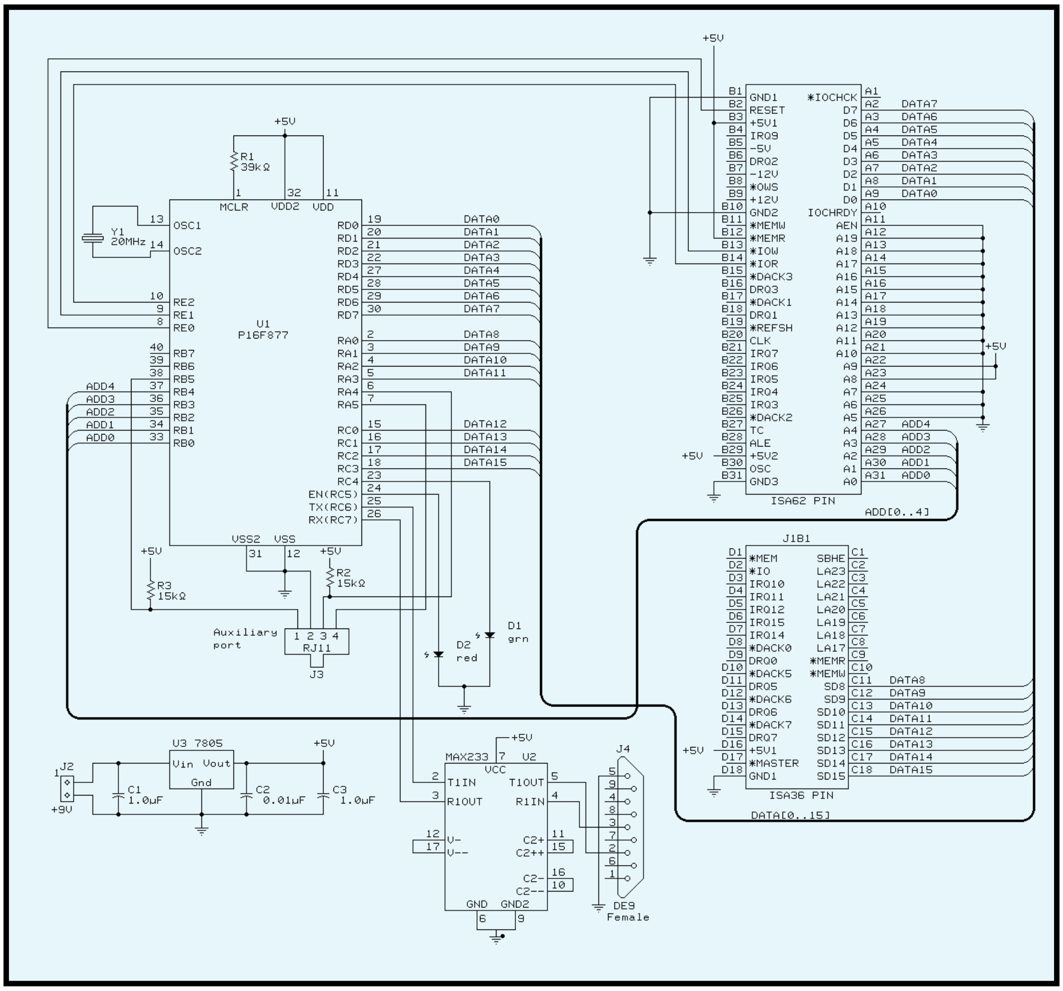


Figure 1—*Here's the PIC Web Server. The PIC controls the Ethernet board by emulating the PC-ISA bus interface. All 16 data, five address, and a minimal set of control lines are implemented. U2 provides the level shifting needed for RS-232.*

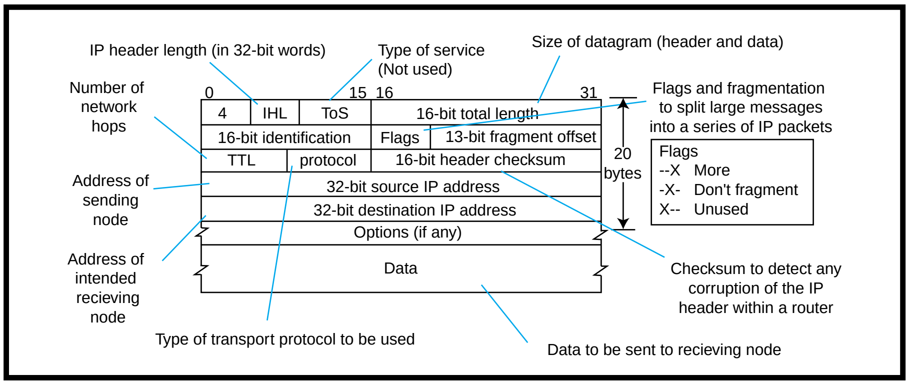

IP header length (in 32-bit words)
Type of service (Not used)
Size of datagram (header and data)
Number of network hops
0
15
16
31
4
IHL
ToS
16-bit total length
16-bit identification
Flags
13-bit fragment offset
TTL
protocol
16-bit header checksum
32-bit source IP address
32-bit destination IP address
Options (if any)
Data
20 bytes
Flags and fragmentation to split large messages into a series of IP packets
Flags
--X More
-X- Don't fragment
X-- Unused
Address of sending node
Address of intended recieving node
Checksum to detect any corruption of the IP header within a router
Type of transport protocol to be used
Data to be sent to recieving node

Figure 2—*Sent within the data field of an Ethernet packet, the IP packet allows the routing of data to an IP stack running on a local network computer.*

this point (not including preamble and SFD) is referred to as the Medium Access Control (MAC) header. The actual data follows this. Packets shorter than 46 bytes need to be padded to fill the difference. Data packets longer than 1500 bytes do not conform to the standard. In practice, the minimum data packet is larger (60 bytes). Finally, the CRC checksum is calculated by the NIC and appended to the packet. National Semiconductor offers a good discussion of the lowest level of Ethernet and the registers in the NE2000 board. [2]

This layer allows you to send packets from one computer to another using various protocols. One common protocol is Internet Protocol (IP), denoted by "0x0800" in the type field of the MAC header (see Table 2). Figure 2 shows the IP packet structure. [3]

Network nodes that receive this packet can decode its total length and the IP addresses of the sender and receiver among other parameters. Suppose a node on the network needs to send an IP packet to another machine. One problem is that you may not have the MAC (hardware) address of the destination machine; you may only have its 4-byte IP address. This is resolved by using the Address Resolution Protocol (ARP). An ARP request on the network essentially asks: "Which computer on the network has the following IP address? Please respond with your physical address." ARP messages are embedded in Ethernet packets (see Figure 3). [4]

The sender of the ARP request uses a broadcast destination address and "0x0806" in the type field of the MAC header. If there is a network node with a matching IP Address, it will respond with an ARP reply filled in with the appropriate information. Often the reply data is cached to keep the number of ARP messages low. To view the ARP cache for your Wintel box, type "`arp -a`" at the DOS prompt.

Now that you can send IP messages from machine to machine, you need something to send. One type of IP message is Internet Control Message Protocol (ICMP), often used for ping applications. Most machines that have a TCP/IP stack running will

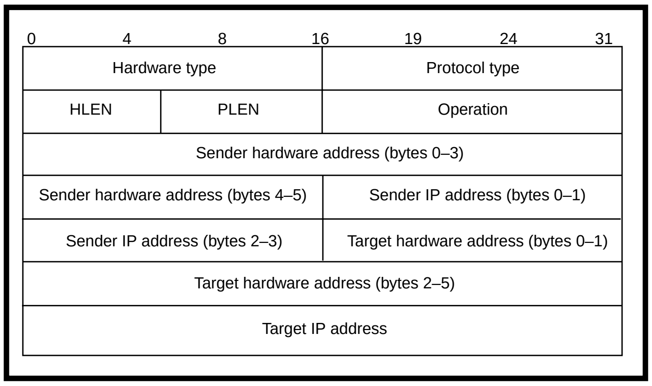

0
4
8
16
19
24
31
Hardware type
Protocol type
HLEN
PLEN
Operation
Sender hardware address (bytes 0–3)
Sender hardware address (bytes 4–5)
Sender IP address (bytes 0–1)
Sender IP address (bytes 2–3)
Target hardware address (bytes 0–1)
Target hardware address (bytes 2–5)
Target IP address

Figure 3—*The ARP packet is sent within an Ethernet packet to exchange hardware address information.*

respond to these requests by echoing the data. Ping often is used to verify connectivity to other computers on the network.

Another type of packet that can be sent inside an IP packet is UDP. UDP is usually used with streaming data, and is unreliable—the sender does not know if all the data is received properly. (However, this is adequate for certain types of data.) Packets such as UDP are "connectionless", meaning a specific connection is not established between the sender and receiver.

Photo 1—*The PIC Web Cam won grand prize in the* Circuit Cellar *PIC2000 design contest sponsored by Microchip.*

Finally, the most commonly known IP protocol is TCP, which is used with web transfers, e-mail, and so on. C programmers will recognize the TCP/IP protocol as being an implementation of the sockets library, where you need to open a connection to another machine before sending data. There also is a handshaking mechanism where the receiver acknowledges every packet sent. Missing packets are then retransmitted. This mechanism ensures that data is sent in a guaranteed fashion.

Note that TCP/IP packets are not only sent to particular hardware and IP addresses, but also to a port address. Commonly known applications have standard port numbers. For example, web servers tend to listen for connections at Port 80. Telnet servers use Port 23. In this manner, packets can be routed to individual applications running on a particular computer. As you see, these protocols are layered on top of the previous ones and placed into the Data field of the lower-level packets. Check the references and resources included to learn more about various fields in the packet structure.

THE NE2000 INTERFACE

I could not find any documents describing the actual NE2000 standard on the 'Net. The most useful resource is National Semiconductor's datasheet on the DP83905 AT/LANTIC Ethernet chip. [2] The NE2000 has registers that allow you to set up its configuration, such as its MAC address for filtering Ethernet messages and 16-KB circular buffer. This important resource is where outgoing data is queued and incoming data is held until the PIC retrieves it.

After the appropriate pointers are set up, data can be read from or written to the circular buffer through one particular register. The pointers are then autoincremented for the next location. Data exchange is 16-bits wide, and all data is simply the contents of the structures described here. In addition to the conventional use of this buffer, the pointers can be configured to leave a portion unused for general storage.

GENERIC SERVER SOFTWARE

The code for the generic web server occupies about half of the 16F877's program space, leaving room for the application code. It is written in C and compiled using CCS's PCM compiler.

The RAM inside the PIC is not used to store Ethernet data. Rather, data that needs to be sent is built up inside the Ethernet card by moving around the write pointer as needed according to the packet structure. If data that takes into account a series of bytes (such as TCP checksums [5]) needs to be prepared, the data is read back one byte at a time while processing. Hence, the NE2000 buffer doubles as PIC memory.

Web clients and servers have a protocol of their own that dictates requests and replies follow an orderly exchange. The PIC Web Server complies with the HTTP 1.0 protocol. [6] This data, as well as HTML files, are sent from server to client in textual form. One programmer-friendly feature incorporated into the web server code is the way text strings are sent to the client. This form of data can be entered into the source code as follows:

```
printf(…,"HTTP/1.0 200\r\n");
```

C programmers will recognize this as a way to output string data. CCS's compiler allows a variant where the name of a subroutine can be substituted for the ellipsis. This causes that function to be called repeatedly for every character in the resultant string. By substituting `ne_print()` (transfers data one byte at a time), whole HTML files can be entered by copying and pasting into the printf statement.

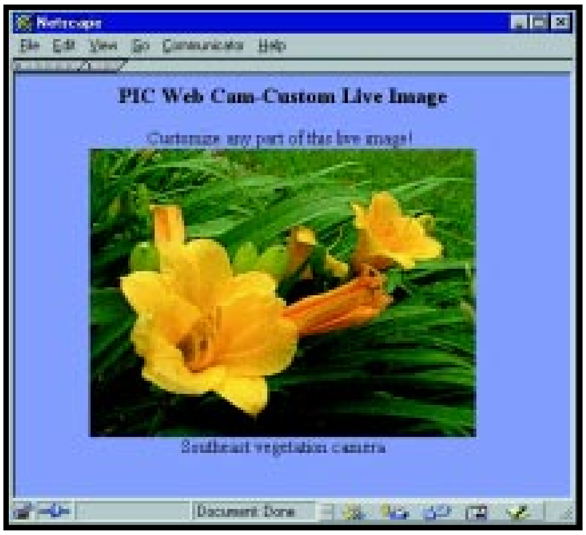
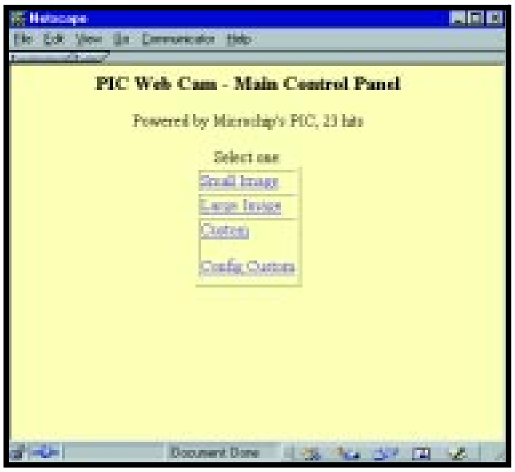

Netscape
File Edit View Go Communicator Help
PIC Web Cam-Custom Live Image
Customize any part of this live image!
Southeast vegetation camera
Document: Done

Photo 2—*This is an example of the image presented by the PIC Web Cam's brower display. In this case, the image is automatically refreshed every 20 s, creating a live image.*



PIC Web Cam - Main Control Panel

Powered by Microchip's PIC, 23 bits

Select one

Small Image
Large Image
Custom
Config Custom

Photo 3—*Here's the PIC Web Cam main control panel as seen on the web browser.*

Building web pages can be easy. Using a text or a WYSIWYG web page editor, format the web page. Then, copy the raw HTML source from the editor into specially prepared printf statements. Finally, compile and run your code. This method doesn't use much program memory space.

Network nodes communicate with the PIC Web Server by first issuing an ARP request with the PIC's assigned IP address. The PIC then responds with its MAC address, allowing future packets to be properly addressed. The TCP/IP stack on the client machine opens a socket to the PIC by issuing a TCP/IP message with the appropriate flags set. Next, the PIC accepts the socket connection and confirms the action. When the socket is open, the web browser submits a GET request containing the full URL as typed by the user in the browser Location box.

This information is then parsed by the PIC Web Server to find out which file is being requested. The PIC returns the file and closes the socket.

With these subroutines, the PIC is now able to serve data to any web browser. Of course, this is of little use without an actual application defining what form of data needs to be handled. So without further delay, here are the applications I submitted to the PIC2000 contest.

THE PIC WEB CAM

With the web server in place, it is easy to present web pages that contain textual data. Without images, this data is boring. When I was considering applications for the server, I asked myself "What better way to show the power of the PIC than an image being served up? Not only that, what if the image was live?" This led to the PIC Web Cam, which received grand prize in the PIC2000 contest (see Photo 1).

Several web servers on the Internet offer live images. They typically involve video cameras feeding image capture boards that are contained in large desktop machines with Ethernet boards. The PIC Web Cam accomplishes this with the web server. The images are obtained from an Olympus D-220L digital camera via its RS-232 interface. The image seen on the browser is refreshed automatically, creating a live image (see Photo 2).

The front of the sealed case is clear, allowing the unit to be used in harsh environments. I custom-built the yellow board above the camera that holds the PIC. The Ethernet board is located behind the camera.

This no-computer web camera is similar to the project published in *Circuit Cellar* 121, by Steve Freyder, David Helland, and Bruce Lightner. They connected an inexpensive digital camera to their Picoweb server. The difference is their use of a Java applet to move the raw image from the server.

Because of the inherent image compression of the JPEG format, the PIC Web Server can deliver a higher resolution image in less time. In addition, you can use the PIC Web Cam with older web browsers that do not support Java.

After the PIC Web Cam is connected to the local Ethernet network, you can communicate with it by using ping or by requesting the default home page with the URL http://IP_ADDRESS, where *IP_ADDRESS* is the PIC's preselected IP address. The resultant display on the web browser is shown in Photo 3.

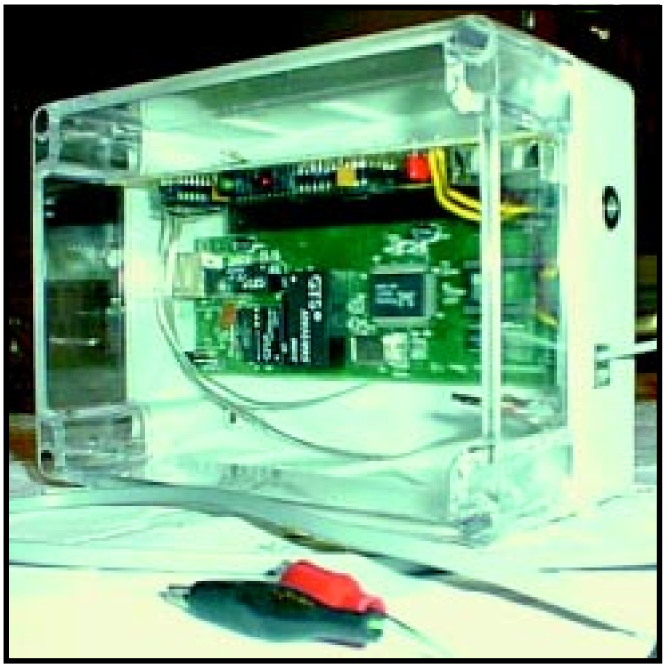


Photo 4—*I configured the PIC Web Server as a Web Chart Recorder. The RJ-11 jack on the right of the box allows connection of the two alligator leads in the foreground. You would connect this pair to the data to be sampled.*

Clicking Small Image decreases the image to 160 × 120, which is refreshed every 10 s. Large Image automatically reloads every 30 s. Lastly, Custom brings up the page that you can customize via the Config Custom link (a form pops up with the existing HTML code). You can edit and store the code in nonvolatile memory.

On the custom page, parameters such as background color, image size, and refresh interval can be adjusted. Because the file is stored in the PIC's EEPROM, a maximum of 256 bytes can be stored.

Camera data is sent to the PIC in 2-KB chunks at 57,600 bps. The data is split because this is larger than one TCP/IP packet. Multiple groups of 2-KB data are transferred from the camera to the browser until the entire image is complete. It takes about 3 s to transfer a small image and about 6 s for a large image. The PIC web server has been tested with three simultaneous web clients.

This example illustrates the amazing capability of a tiny PIC processor. It can not only do the work of a desktop computer, but also present digital images from a still camera. The PIC processor is a valuable addition to any system requiring an inexpensive, effective way to transmit live images over the Internet.

PIC WEB CHART RECORDER

Another valuable project is the PIC Web Chart Recorder, which was awarded first prize in the Internet Applications category of the contest.

The graphical capabilities of the Internet provide a good way to present continuously varying data in the form of a chart. Sites including stock brokerages and weather stations benefit from this technology. Instead of using a large desktop computer, the PIC Web Chart Recorder (see Photo 4) provides this capability with a PIC.

Analog data is gathered by the PIC's A/D converter and presented by the generic web server. Use the control panel to control the sampling speed—stopped (no sampling), 10 s, 1 min., 15 min., or 1 h per division.

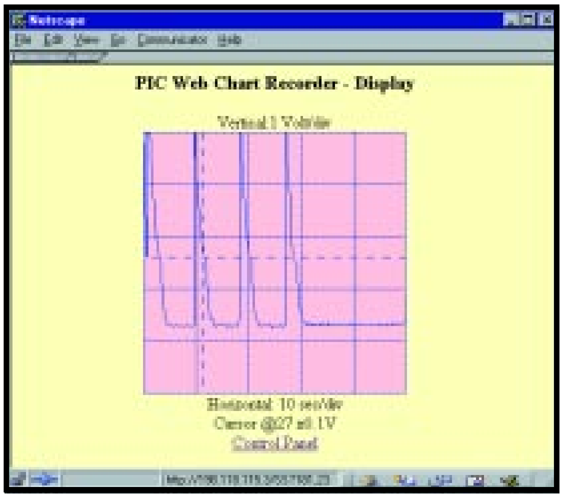


Photo 5—*The chart is presented in the form of a GIF that is generated on the fly in the PIC Web Server. Note the following: the data is the solid line; division markers are dotted lines. Moving over the GIF updates the mouse coordinates at the bottom of the web page (here they are 181, 23). Clicking on the image puts a dashed crosshair showing the value of the chart at the x-coordinate (its voltage is at the bottom (here it's 2.7 V)).*

The graphical data is packaged as a GIF image that is generated on the fly by the PIC. To simplify the generic web server code, it is split into five 255 × 52 GIFs that are assembled at the browser. Because they are all the same width, they stack seamlessly. The display is automatically refreshed once per minute. During each refresh the graph is shifted from right to left to make room for new data.

The core C code that generates the GIF file is a modified version of the one published in *Circuit Cellar* 107, by Paul Breed. Modifications include changing it to a RAM-limited 8-bit environment and making it stream output into the Ethernet card.

For better readability of the graph, horizontal and vertical dotted lines are added during creation of the GIF. In addition, where the data jumps by greater than one count from one sample to the next, vertical lines are added to connect the graph in a continuous manner. Each column of pixels in the image represents one sample of the PIC A/D converter, and the vertical row represents the data value. As a result, the GIF is 256 pixels high (representing the one byte data range), and 255 pixels wide (representing the 255 samples that are stored).

The chart is defined as an HTML image map, causing the coordinates of the mouse cursor to be continuously displayed at the bottom of the web page. The browser updates the information as you move the cursor over the chart. This is useful when reading data values.

If you click on the GIF, a new image will be fetched from the PIC Chart Recorder, showing a dashed crosshair at the x-coordinate of the mouse click and the y-coordinate of the corresponding data value. In addition, the display underneath the GIF is updated with the voltage value of the sample. In Photo 5's example, the data value is 2.7 V.

The PIC Web Chart Recorder is a useful addition to any system where analog data needs to be gathered in a graphical form. The unit can not only gather data, but also present it in a web-friendly format. Real-time generation of GIF images, normally accomplished by desktop computers, is done here by a PIC.

Although the PIC Web Cam delivers data that is more interesting, the Chart Recorder represented a bigger challenge concerning software development. I did not know until the end if a complete web server and GIF file generator would fit in 8 KB of program space. Fortunately, the results exceeded my expectations.

In addition to the complex chart discussed here, the GIF can include objects that are easily described in mathematical form, such as circles, lines, and rectangles. So, the PIC Web Server can generate dynamic graphics such as thermometer bars, simple animation, and other indicators that change appearance based on real-time data acquired by the PIC.

PIC X-10 WEB SERVER

For the next web server project, I went back to the original application that I had intended for the PIC Web

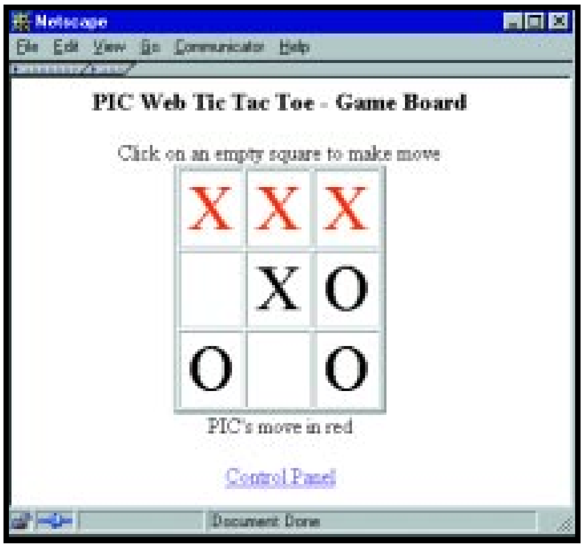

Netscape
File Edit View Go Communicator Help
PIC Web Tic Tac Toe - Game Board
Click on an empty square to make move
X
X
X
X
O
O
O
PIC's move in red
Control Panel
Document: Done

Photo 6—*Many people can play tic-tac-toe against the computer simultaneously. The PIC won this time.*

Server, home automation. The home automation industry is dominated by a standard known as X-10. X-10 units communicate over the existing power line wiring to control all sorts of loads. A simple power line modem enables a computer to send and receive these commands.

The PIC X-10 Web Server combines a server and an X-10 controller. With this project, you can control and monitor the status of your home or office over the Internet. First prize in the Internet Connectivity category was awarded to this project.

The X-10 Web Server control panel page automatically reloads itself every 5 s, allowing you to view the status of the units in a dynamic color-coded fashion. In addition, you can click on the hyperlinks to send commands to control the loads. If any other controller sends a command to the load, its status is also automatically updated in the home control panel.

The control panel can be reconfigured if you click on the Config button at the bottom of the main control panel. To add a unit, simply enter the name and X-10 address and click Add. You are then returned to the main autoloading home control panel with the new unit added (its status will be displayed as a blue question mark until a command is sent or received). Conversely, you can eliminate a unit from the control panel with Delete. The names and statuses of up to 16 units are preserved in the PIC's nonvolatile EEPROM.

The TW523 Two-Way X-10 power line modem uses a low-level, timing-critical interface. Thanks to the buffer in the NE2000 card, no particular attention needs to be paid to the server's Ethernet timing. An interrupt that fires with every power line transition ensures that bits get sent and received properly over the power line.

This powerful and compact web server will benefit any environment requiring the capability to send and receive X-10 commands without installing unusual hardware or software. Any computer on the local network can perform as the control panel.

THE PIC SLIP SERVER

In addition to the previous two projects, a third design was successful in the Internet Connectivity category of the contest. The PIC SLIP (Serial Line Internet Protocol) Server earned third prize.

Internet-connected PIC processors have thus far used serial ports for physical connection to the outside world. Unfortunately, the advantages of the PIC's small size are tainted by the need for a desktop computer as the Internet gateway. The SLIP Server addresses this deficiency.

The PIC processor's serial port provides the SLIP line. [7] The server allows almost any microprocessor to access the local IP resources (web and FTP servers, etc.) using only its serial port.

The versatile SLIP Server has many applications, among them is providing a wireless LAN connection. Using off-the-shelf wireless RS-232 links, you can put together a wireless network connection. A second application is establishing connectivity for computers with a serial port and no network card, such as PDAs, palmtops, and embedded computers. By adding a standard modem, the SLIP Server can create a remote access server. And it can provide Internet access for various other PIC projects, allowing them to send and receive data between FTP and web servers using only its serial port.

Using dial-up networking software, a Windows95 PC served as the SLIP client while I developed this project. The PC serial port and the SLIP Server were connected, and standard Internet clients (like Netscape and ping) were used to test the SLIP Server.

The main difference between IP data sent on the serial line and Ethernet is the Ethernet header (a.k.a., MAC). The latter contains the 6-byte hardware address of the source and destination nodes. Before any Ethernet data can be sent, the hardware address of the recipient must be determined by the Address Resolution Protocol (ARP). ARP requests are sent in broadcast mode, and the node that matches the addressing information responds with an ARP reply. Data obtained via the ARP protocol is stored in nonvolatile EEPROM memory. The data can be retained between power cycles or cleared during powerup.

Serial characters from the SLIP client are examined for the modem "AT....<CR>" character sequence. If one is detected, an "OK" string is echoed. This fools the PC into thinking that the modem has dialed and connected to the remote computer. The PIC continuously searches for the SLIP flag character, which demarks the start and stop of all packets. If it is at the beginning of a packet, the NE2000's registers are readied for loading data. If the flag character marks the end of a packet, the packet is examined to determine if the destination IP address is in the PIC's ARP table. If the entry is not found, an ARP request is sent to obtain the physical address. If the entry is found, the correct physical address is prepended to the IP packet and sent out on the Ethernet network.

On the Ethernet receiving end, new packets are checked for their type. A correct ARP reply is stored in the PIC's ARP table. If a remote IP node is querying the PIC for its hardware address, an ARP reply is sent back. Incoming IP packets are forwarded to the serial line with the correct SLIP flag and escape sequences.

In operation, the SLIP Server allows the client PC to access any IP service on the network. It also allows remote nodes to access any server on the client PC, showing that the SLIP Server has bridged the connection to the Ethernet. The client computer can now interact with the network resources as if it has its own Ethernet connection.

PIC WEB TIC-TAC-TOE

Although this last web application hosted by the PIC did not win a prize, it is worthy of mention. When I was a young boy, I visited a computer display at a science center. A refrigerator-sized computer was located in the middle of the room attached to six large displays with keyboards. People could get a sense of the "intelligence" of the computer by playing tic-tac-toe against it. I was impressed that a computer could play against humans, and that it was difficult to beat.

This capability has been shrunk into a PIC processor running the web server. Using any web browser on the local network, you can match wits with the PIC by playing against it over the Internet. As many as several dozen people can play simultaneously.

You can choose who makes the first move. An example game board during play is shown in Photo 6. During each turn, you move by selecting a hyperlink on an open square. This causes the web browser to retrieve a web page from the PIC. The returned HTML file contains a new game board updated with your move and the PIC's countermove. When either party wins, the winning row is highlighted in red.

The state of the game is kept in the hyperlinks. Thus, an unlimited number of people can play against the PIC because no additional storage is needed per player. The only limitation is the maximum number of moves that the PIC can handle per second. Assuming a conservative value of 20 moves per second and a 2-s pause between moves, the PIC can handle 40 simultaneous players! As a side benefit, because the entire state of the game is encoded in the hyperlink, you can bookmark Resume Game.

The strategy that the PIC uses to countermove is simple:

- if a win is imminent, go for the win
- if the opponent is about to win, go for the block
- if the center square is open, take it
- if a corner square is open, take it
- play any open square

This sufficiently simulates intelligent play for most matches.

The tic-tac-toe project illustrates how far silicon has come in the past few decades. Instead of a refrigerator-sized machine, the computer is the size of a fingernail. Teamed with a standard Ethernet card and attached to the local network, many players can match wits against the PIC using only their browser software.

THE END RESULT

What happens when you combine a powerful little microprocessor with the ubiquity of the 'Net? You get the PIC Web Server. With the HTML language, a PIC can now communicate in style with any user on the network. Complex and fancy user interfaces with control buttons, live images, dynamic graphics, charts, and up-to-date data are now possible. A single web page can contain data from an almost unlimited number of PIC Web Servers, making the display come alive with real-world data.

Edward Cheung works at NASA Goddard Space Flight Center in Maryland where he builds instruments for the Hubble Space Telescope. Astronauts install this hardware in space every two to three years. His interests include home automation and gardening. You can reach him at edward.b.cheung.1@gsfc.nasa.gov or visit cheung.place.cc.

SOFTWARE

You can download the source code from www.mindspring.com/~dr_ed/awards/pic2k/pic2k.htm.

REFERENCES

[1] Steve Freyder, David Helland, Bruce Lightner, "A $25 Web Server," *Circuit Cellar* Online, July 1999, www.chipcenter.com/circuitcellar/july99/c79b11.htm.
[2] National Semiconductor, AT/LANTIC Ethernet chip, www.national.com/ds/DP/DP83905.pdf.
[3] University of Aberdeen, IP packet structure, www.erg.abdn.ac.uk/users/gorry/eg3561/inet-pages/ip-packet.html.
[4] ARP packet structure, melb.alexia.net.au/~www/vendor/internetinfo/arp.html.
[5] Rensselaer Polytechnic Institute, TCP checksum, www.ecse.rpi.edu/Homepages/shivkuma/teaching/sp99/i07-udp/sld015.htm.
[6] University of Massachusetts, HTTP protocol, gaia.cs.umass.edu/kurose/apps/http.htm.
[7] The Ohio State University, SLIP protocol, www.cis.ohio-state.edu/htbin/rfc/rfc1055.html.

RESOURCES

Camera protocol
www.average.org.digicam/protocol.html

NBC weather charts
www.aws.com/nbc/wrc

X-10 home control module
X10, Inc.
www.x10.com

SOURCES

C Compiler
Custom Computer Services, Inc.
(262) 797-0455
Fax: (262) 797-0459
www.ccsinfo.com

MAX233
Maxim Integrated Products
(408) 737-7600
Fax: (408) 737-7194
www.maxim-ic.com

PIC 16F877
Microchip Technology Inc.
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

FEATURE ARTICLE

Jeff Stefan

Navigating with GPS

If you're going to be heading out with a minivan full of little ghosts and goblins this Halloween, you may want to brush up on some GPS navigation background info. Luckily, Jeff has all the details that you'll need to find your way.

Global Positioning System (GPS) receivers are abundant and cheap, paving the way for anyone to write a simple yet powerful navigation program. All you need is a C compiler, a laptop or small computer, and a couple navigation formulae. Seems simple? It is if you know what data to use, what conversions to make, and which formulae to use. This article reveals the twists and turns required to put your GPS receiver to work and get you navigating quickly. The application program and functions I'll present allow you to calculate the distance and heading from your current position to any other position on earth. The distance and bearing functions provide the heart of a dynamic and useful navigation system.

GPS FUNDAMENTALS

GPS became available in 1978 with the successful launch of NAVSTAR 1. NAVSTAR 1 was the first of four NAVSTAR satellites launched that year, creating an operational satellite navigation system for the military. Then in 1982, Russia launched a system called GLONASS.

GPS satellites are incredible instruments. Each satellite contains four atomic clocks that operate on a level of one second of error in three million years. This degree of precision time keeping is required so each satellite can operate autonomously yet remain synchronized. GPS satellites transmit ranging codes based on a signal's time of arrival, not position and motion.

These satellites, which are at known locations at all times, transmit on two L-band carrier signals. The satellite's receiver marks the difference between the time the signal was sent and received, and multiplies the difference by the signal speed (close to the speed of light). Using ranging code from four satellites, a GPS receiver can calculate its own position in three-dimensional space, including the receiver's velocity.

The NAVSTAR system breaks down navigation into two domains, Standard Positioning Service (SPS) and Precise Positioning Service (PPS). PPS accuracy is published at 21-m horizontally and 29-m vertically. The early NAVSTAR SPS was so accurate that it was considered a threat, so the gap between SPS and PPS was intentionally widened. The accuracy level of the SPS was decreased to 100 m in the horizontal plane and 160 m in the vertical plane. The decrease, called selective availability (SA), introduced error into the satellite orbital data and time transmissions.

SA made life more difficult for commercial GPS-based navigation systems. One hundred meters (roughly 300′) of accuracy isn't bad, but if you're trying to develop a precise hand-held or automotive navigation system, more accuracy is needed. To the delight of the navigation community, the U.S. government turned off SA on May 1, 2000. Instead of

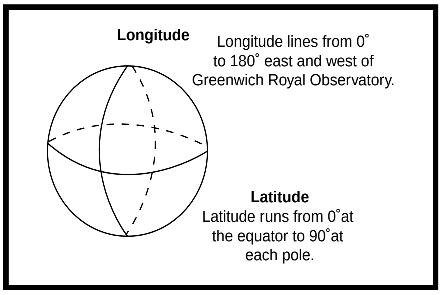

Longitude
Longitude lines from 0˚ to 180˚ east and west of Greenwich Royal Observatory.
Latitude
Latitude runs from 0˚at the equator to 90˚at each pole.

Figure 1—*Longitude lines run east and west from pole to pole. Latitude lines run north and south, parallel to the equator.*

100 m, accuracy now is within 10 to 30 m in the horizontal plane and slightly more in the vertical plane.

Now, the floodgate is open for new and highly accurate GPS applications based on latitude, longitude, and time. GPS receivers turn up in everything from wristwatches to locomotives.

Latitude and longitude are fundamentals of navigation. Sometimes it's difficult to remember which is which. I use the mnemonic "it's a long way from the North Pole to the South Pole." Longitude lines run from the North Pole to the South Pole and are measured in half circles from the Royal Greenwich Observatory in Greenwich, UK. Longitude lines run from 0° to 180° east and 0° to 180° west (see Figure 1).

Latitude lines run in parallel from the equator to the North and South Poles. Latitude lines run from 0° at the equator to 90° at the North and South Poles. As the lines of latitude get closer to the poles, they become smaller. This presents a problem when trying to use a two-dimensional distance formula, as I'll explain later.

GETTING THE DATA

Most GPS receivers output data in NMEA-0183 format. NMEA stands for the National Marine Electronics Association. The data is sent via RS-232 at 4800 bps, with most GPS receivers providing a serial port that outputs NMEA GPS messages at 1-s intervals. NMEA messages are sent by talkers (identified by a two-character ID string with a "GP" prefix) and received by listeners. The messages are one-way: from talker to listener. Thus, an application program's control of the receiver's output is limited. Data enters your application program at 4800 bps at 1-s intervals, period. The data rate and burst time can't be changed. This is OK for most land- and sea-based applications.

An NMEA sentence contains an address field, data field, and checksum. The address field is composed of a sentence formatter and talker identifier. The latter indicates where the data comes from. For GPS, the talker identifier is GP. Two useful GPS talker identifiers are RMC and GGA. The sentence formatter indicates the content of the data field.

NMEA messages are easy for a program to parse because they are consistent and well defined. The general NMEA message format is:

$<Address>,<Data>*<Checksum><CR><LF>

The address field, <Address>, is broken down as <talker><sentence_formatter>. All fields are comma delimited except <Checksum>, which is delimited by a star (see Figure 2).

Table 1 lists eight examples of NMEA GPS messages. The most useful one is the RMC message, which contains all of the basic information required to build a navigation system. RMC is listed as recommended minimum specific GPS/transit data (see Figure 3). Although I don't know what the "C" stands for in RMC, I know I like the utilitarian nature of this message. It contains time, status, position, speed, course, and date.

Looking at the RMC message, the first chunk of data encountered is $GPRMC. As the NMEA sentence describes, this is the talker and sentence formatter. Universal Time Coordinated UTC data follows the sentence formatter; the time is given in hours, minutes, seconds, and decimal seconds. Next is the GPS status indicator (A), which indicates whether or not the incoming GPS data is valid. The V in this field seems to indicate that the data is valid, however, it means the opposite. An A in this field means that the data is indeed valid.

There are many reasons why a GPS receiver would output invalid data. For example, the receiver might not have acquired enough satellites for a position fix yet, foliage or buildings might block the GPS signals, or the GPS almanac or ephemeris data could be out of date. Invalid data output from a receiver is almost always temporary, and a V usually will become an A within seconds or minutes.

The next two fields cover latitude and determine whether the latitude is in the Northern or Southern Hemi-

Table 1—*These are useful GPS NMEA messages used for navigation applications.*

RMC	Contains recommended minimum specific GPS/transit data
ALM	Provides GPS almanac information
GLL	Provides latitude, longitude, and UTC (Universal Time Coodinated) data
ZDA	Contains UTC along with day, month, year, and local time
GGA	Contains UTC, fix, and position data
GSA	Provides GPS DOP and active satellite information
VTG	Provides "track made good" and ground speed
ZDA	Provides the current time and data

Listing 1—*Here's the structure that holds parsed RMC messages.*

```
#define STRING10 10
#define STRING7  7

struct rmc
{
   char UTC[STRING10];
   char Status;
   double Lat;
   char NS;
   double Lon;
   char EW;
   double Speed;
   int Course;
   char Date[STRING7];
   char Var[STRING10];
};
```

Ddmm.mmmm to dd.dddd	Separate and save dd from the incoming latitude and longitude Divide mm.mmmm by 60, yielding 0.dddd Add the saved dd to 0.dddd, yielding dd.dddd
dd.dddd to radians	Radians = dd.dddd/57.2957795
Radians to dd.dddd (degrees)	Degrees = radians × 57.2957795
Radians to nautical miles (NM)	NM = radians × 3437.7387
NM to statute (land) miles (MI)	MI = NM × 1.150779
MI to feet (FT)	FT = MI × 5280

Table 2—*Here are common navigation units and conversion factors.*

sphere. Following the latitude fields are the corresponding longitude and east/west indicators. The two fields after that, speed (knots) over ground and course (degrees) over ground, are handy. Next is the date, and then the magnetic variation (east or west).

The RMC message is available on almost all receivers that output NMEA messages. As stated, GPS receivers supporting NMEA messages output data at 1-s intervals at 4800 bps, so processing data at 1200- to 1800-ms intervals ensures enough time to fill up a receiver buffer and transfer the data to a holding buffer. The data in the holding buffer can be parsed and processed while new information enters the receive buffer.

Listing 1 shows a C structure into which you can deposit the parsed RMC message data. The [`#defines`] identify character array lengths and are optional. The structure contains the proper data types to contain the RMC data fields. You can create similar structures for additional NMEA messages that an application needs.

After the GPS receiver deposits data in a buffer named `InputQueue`[], the data is transferred to another buffer called `InputMsgBuff`[]. The latter is used to extract the RMC or other NMEA messages of interest. To extract the data, create a pointer and have it point to `InputQueue`[]. For transferring data, you need to dereference the pointer to `InputMsgBuff`[]. This is illustrated in the C code fragment in Listing 2.

When a message is in a buffer, the talker and sentence formatter can be identified and processed. This bit of code collects a sequence of messages so multiple messages can be processed. This allows you to create custom structures, spanning the data from multiple messages. For example, a structure can be created that holds speed, course, latitude, longitude, the number of satellites in view, and dilution of precision values.

WAYPOINT NAVIGATION

Waypoint navigation is based on great circle navigation. Great circle navigation is general and good for planes, boats, and cars. Waypoint navigation systems navigate via latitude and longitude pairs. The navigation computer accesses a list of latitude/longitude pairs and calculates the distance and bearing from one point to another. Information presented is usually the current distance and bearing from your present position to the next waypoint. Often, a dynamic directional pointer is displayed, which you follow to the next waypoint.

Before navigating, the data from the GPS receiver must be converted to a form acceptable to the great circle navigation algorithms (i.e., the distance and bearing formulae). First and foremost, all of the data must be in radians. This seems straightforward, but there's a complication. The latitude and longitude data emitted by most receivers is in a form that cannot be directly converted to radians. So, an intermediate latitude and longitude conversion sequence must take place.

All NMEA data is emitted as ASCII data. Latitude and longitude data received from a GPS receiver in NMEA-0183 format is in units ddmm.mmmm, where dd equals degrees, *mm* equals minutes, and *.mmmm* is decimal minutes. These units are not appropriate for the distance and course calculations; they must be converted to degrees and decimal degrees, then to radians.

The first step is converting the latitude and longitude data from the form ddmm.mmmm to dd.dddd. This is a straightforward algorithm, but it still takes a substantial amount of code. To determine the algorithm, first separate and save dd from the incoming latitude and longitude string. Then, divide mm.mmmm by 60, resulting in an exponent of zero and a new mantissa, 0.dddd. Third, add the saved dd to the result, yielding dd.dddd.

After you finish converting both latitude and longitude, radian conversion is possible. The formula to convert from dd.mmmm to radians is:

$$\text{radians} = \frac{\text{dd.dddd}}{57.3}$$

After performing the distance and bearing calculations, the data needs to be converted back to dd.mm.mmmm. This is done by following the formula degrees = radians × 57.2957795. To convert back to the form ddmm.mmmm, save dd, multiply .dddd by 60, and add the exponent to the result, yielding mm.mmmm. Then concatenate the saved dd, resulting in ddmm.mmmm.

That takes care of the latitude and longitude conversions. Now, you can tackle the knots and nautical miles (NM) conversions. All speed and distance data contained in NMEA messages is in terms of knots and nautical miles. One NM corresponds to the traversal of 1 s of arc. One knot is 1 NMph. So, if you're traveling at five knots (5 NMph), you'll traverse 5 s of arc in 1 h.

Now, convert knots to miles per hour and nautical miles to statute (land) miles (see Table 2). Remember that the output of the navigation calculations is in radians. The conversion to nautical miles is NM = radians × 3437.7387. Next, you can convert nautical miles to land miles (MI) using MI = NM × 1.150779. Converting from land miles to feet (FT), the formula is FT = MI × 5280.

NAVIGATION FORMULAS

Now that the units are all in line, the latitude and longitude data points can be run through the great circle algorithms, yielding correct results. The distance calculation is performed first because the distance is a factor in the bearing calculation. To compute the great circle distance between two pairs of latitudes and longitudes, use:

$$d = \text{acos}(\sin(\text{Lat1}) \times \sin(\text{Lat2}) + \cos(\text{Lat1}) \times \cos(\text{Lat2}) \times \cos(\text{Lon1} - \text{Lon2}))$$

This formula accurately yields the distance between two points on the globe. Remember that the units are in radians, so to convert from radians to nautical miles, use the formula NM = radians × 3437.7387. Then you can convert to land miles or kilometers. Some languages and programming environments, such as Visual Basic, do not support a direct `acos( )` function. Instead, you can use an `atan( )` function coupled with the relation $\text{acos}(x) = \text{atan}(\text{sqrt}(1 - x^2) / x)$. For calculating distance, I use the sequence of temporary variables as follows:

$$t1 = \sin(\text{Lat1}) \times \sin(\text{Lat2});$$
$$t2 = \cos(\text{Lat1}) \times \cos(\text{Lat2});$$
$$t3 = \cos(\text{Lon1} - \text{Lon2});$$
$$t4 = t2 \times t3;$$
$$t5 = t1 + t4;$$
$$\text{rad\_dist} = \text{atan}\left(\frac{-t5}{\sqrt{(-t5 \times t + 1)}}\right) + 2 \times \text{atan}(1)$$

This sequence works well. While taking a few more steps than one monolithic formula, intermediate variables are exposed, allowing you to debug the distance algorithm as it progresses. And, this sequence works with all programming languages. To prove it, t1 through t5 can be consolidated, but sometimes it's good to see

Figure 2— *This is the NMEA sentence format.*

```
$<Address>,<Data>*<Checksum><CR><LF>
<Address> = <Talker><Sentence_formatter>
<Talker> = GP
<Sentence_formatter> = one of RMC GGA GSV…ZDA
```

RMC message

$GPRMC,nnmmss.ss,A,IIII.II,a,yyyy.yy,a,x.x,x.x,xxxxxx,x.x,a*hh<CR><LF>

$GPRMC	Address field
hhmmss.ss	UTC of position fix (hours, minutes, seconds, decimal seconds)
A	GPS status: A means data is valid, V means data is invalid
LIII.II	Latitude
a	North/south
yyyy.yy	Longitude
a	East/west
x.x	Speed over ground in knots
x.x	Course over ground, degrees true
xxxxxx	Date: ddmmyy (day, month, year)
x.x	Magnetic variation
a	East/west
*hh<CR><LF>	Checksum

Figure 3—*Here's the NMEA RMC message format, followed by definitions.*

what's going on in a mathematical algorithm at different steps.

Why not use the Pythagorean Theorem (remember, $d\sqrt{x^2 + y^2}$) to compute the distance between two points? For navigation, *x* would be the absolute value of the difference of the latitudes, and *y* would be the absolute value of the difference of the longitudes. This is true for the proximity of 300″ but rapidly deteriorates beyond that.

Hence, the Pythagorean Theorem is useful only in two-dimensional space. You're navigating in three-dimensional space, so for short distances, the theorem appears to work, but it fails for long distances. So, although it's an easier formula touse, you can't use it for any significant distances.

Now that distance is calculated, the next thing to do is calculate the bearing from one point to another. Bearing tells you which way to go. It is defined as the angle measured horizontally from north to the current direction of travel. North can be true north or magnetic north. Again, the great circle distance (d) between two points must be previously calculated. The classic bearing formula is:

$$c = \operatorname{acos}\left(\frac{\sin(\text{Lat2}) - \sin(\text{Lat1}) \times \cos(d)}{\cos(\text{Lat1}) \times \sin(d)}\right)$$

where *d* equals the great circle distance. The result *(c)* must be qualified by testing whether or not sin(Lon2 – Lon1) is negative. If negative, the true course is determined by 360° – c. You will end up at the destination, but

Listing 2—*This code transfers an NMEA message from a raw input queue to a message processing buffer.*

```
unsigned char *locptr;       // local buffer pointer
int i = 0;                  // array index
/****************************************
* Transfer InputQueue data to InputMsgBuff[]
***************************************/
locptr = InputQueue;
/****************************************
* Check for NMEA message start character '$'.
* If '$' is found, transfer message to InputMsgBuf.
****************************************/
if (*locptr == '$')
{
   while (*locptr != '\0')
    {
        InputMsgBuff[i++] = *locptr;
     }
  InputMsgBuff[i] = '\0';
}
```

you'll be taking the long way around the globe. Again, I like to break down the algorithm into discrete steps using temporary variables (see Listing 3).

To create a direction pointer, subtract the current GPS heading of the RMC message from the calculated bearing. Add 360° if the result is negative, creating an angle value that points from one waypoint to another.

Many different sources are available to determine waypoints. Inexpensive PC-based mapping programs provide methods of converting map points to latitude and longitude. Converting from an address to a latitude and longitude value is called geocoding. Converting from latitude and longitude values to an address is called reverse geocoding. Using the algorithms provided here and a GPS receiver, you can create your own waypoint-capturing program. Simply provide some code that will save the incoming RMC message when you pass over a location. The saved message contains the latitude and longitude of the point passed over. You can use a collection of these values to create accurate maneuver lists for roads, trails, rivers, and lakes.

Figure 4 illustrates the main components of a simple navigation system. Data is input from a GPS receiver serially to an input buffer at 4800 bps, 8 data bits, 1 stop bit, and no parity. The data is input periodically at 1-s intervals. During the time between the input data bursts (typically 200 to 800 ms), the input buffer data is transferred to a parser buffer. The data in the parser buffer is used as input to the NMEA parser that separates the data into different components—latitude, speed, and so forth.

Data from the NMEA parser is made available to the display and navigation engines. The navigation engine computes the distance, bearing, and direction pointer, then gives the information to the display engine for rendering and display.

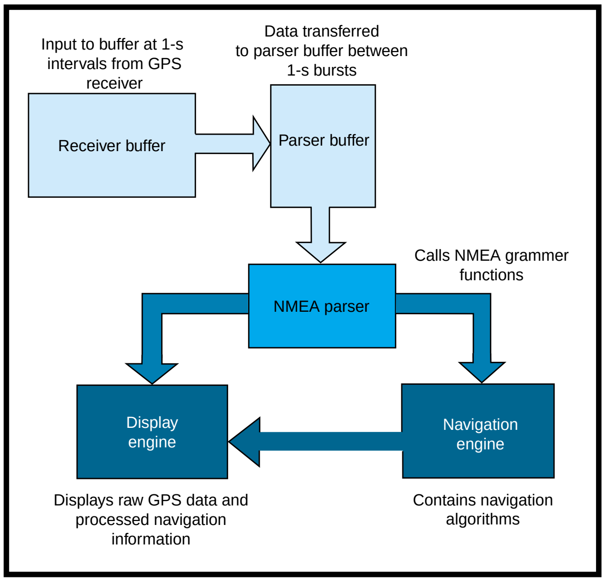

Input to buffer at 1-s intervals from GPS receiver
Data transferred to parser buffer between 1-s bursts
Receiver buffer
Parser buffer
Calls NMEA grammer functions
NMEA parser
Display engine
Navigation engine
Displays raw GPS data and processed navigation information
Contains navigation algorithms

Figure 4—*The simple navigation system architecture is pictured here.*

Listing 3—*There's no doubt this can be easily optimized, but the algorithm is broken up to be more illustrative and instructional than optimal.*

```
if (sin (Lon2 - Lon1) < 0.0)
{
  t1 = sin(Lat2) - sin(Lat1) * cos(rad_dist);
  t2 = cos(Lat1) * sin(rad_dist);
  t3 = t1 / t2;
  t4 = atan(-t3 / sqrt(-t3 * t3 + 1)) + 2 * atan(1);
    rad_bearing = t4;
}
else
{
  t1 = sin(Lat2) - sin(Lat1) * cos(rad_dist);
  t2 = cos(Lat1) * sin(rad_dist);
  t3 = t1 / t2;
  t4 = -t3 * t3 + 1;
  t5 = 2 * 3.14 - (atan(-t3 / sqrt(-t3 * t3 + 1)) + 2 * atan(1);
  rad_bearing = t5;
}
```

The companion program to this article, navcalc.c, takes latitude and longitude pairs from the command line and computes the great circle distance from the first latitude/longitude pair to the last pair. The source and destination latitude and longitude values supplied in the program text are for southeastern Michigan. To create a dynamic navigation program based on this code, use the latitude and longitude received and parsed from a GPS receiver as the source coordinates, and continuously calculate the distance and bearing to the destination coordinates.

Try different latitude/longitude source and destination pairs in your city and compare how well the output values match reality.

PARTING COMMENTS

The C code provided supplies the basic building blocks for a small, low-cost yet significant navigation application program. GPS receivers are available on the 'Net for bargain basement prices. Mapping programs that provide latitude and longitude data are widely available. The C code supplied in the example program is portable, so it runs on most processors that support floating-point operations and trigonometry functions.

Now you're on your way to creating your own navigation program. Being lost will be a thing of the past! ■

Jeff Stefan is an engineer at OnStar. He holds a B.S. in Computer Science and has worked in embedded systems software design for many years. Jeff is the author of more than a dozen technical articles and currently is working on his first book. You may reach him at jmstefan@mindspring.com.

SOURCE

NMEA Specification
National Marine Electronics Association
(919) 638-2626
Fax: (919) 638-4885
www.nmea.org

FEATURE ARTICLE

Alan Singer

Internet Connectivity

Do-it-Yourself or Off-the-Shelf?

Implementing Internet connectivity into products can be a gamble, but as Alan explains, you've got to know when to roll your own.... Understanding the different factors that need to be considered can take the chance out of playing the I-card.

More and more we're reading about the deployment of Internet-enabled appliances such as cell phones supporting WAP (wireless access protocol), set top boxes that connect your TV to an ISP, and interactive games that let you play video games with someone on the other side of the world. Most of these devices use powerful 32-bit microprocessors that run sophisticated applications, such as voice compression or render high-resolution graphics.

MPUs can perform calculations of 100 or more MIPS (millions of instructions per second). Generally, MPUs don't use all this processing power, so extra MIPS are available for other uses. Design engineers correctly argue that MPUs have available headroom to run the Internet protocols as well.

In order to connect a device to the Internet, it must support the relevant Internet protocols required for specified connectivity. The basic TCP/IP stack may use between 35 and 60 KB of memory, depending on whether or not the vendor supports the complete stack from the PPP protocol on the Physical Layer through the TCP and UDP protocols on the Transport Layer of the Open System Interconnection (OSI) seven-layer networking model. On top of the TCP/IP stack, there may be e-mail protocols to support, including SMTP (Simple Mail Transfer Protocol) for sending to a mail server and POP3 (Post Office Protocol 3) for receiving from a mail server.

These protocols can add an extra 5 to 15 KB of memory, depending on whether or not they are both supported. If the device supports HTTP, the web client deals with an extra 10 to 20 KB and the server faces an extra 20 to 40 KB. Thus, adding Internet connectivity can increase 70 to 135 KB to the application, not counting a real-time operating system and additional management, which can add another 55 KB (see Figure 1).

The question is whether to do the work yourself or buy off the shelf. In this article, I'll explore the risks and benefits of each approach.

Few design engineers don't want to do it themselves. After all, they are paid to develop solutions and like the challenge of a new project. Manufacturers could buy the protocols off the shelf from vendors such as Allegro, Agranat, Be, and Wind River. But, then they integrate the protocols into their application, which means they need Internet programmers onboard or subcontractors.

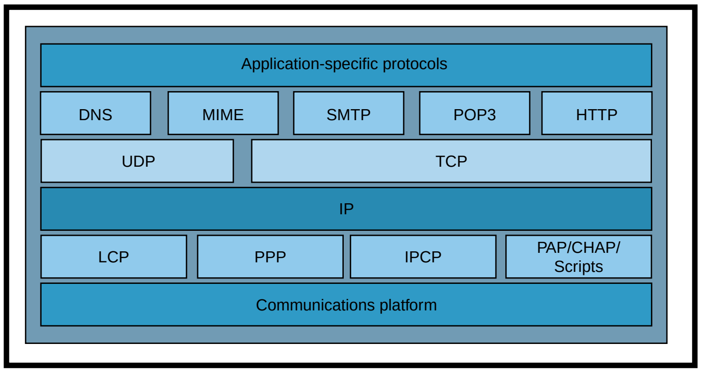


Figure 1—*Connect One customizes the iChip protocol stack according to the dial-up application.*

THE BASICS

There are manufacturers ruled by the NIH (not invented here) mentality. In such places, the engineer writes the complete protocol stack from top (Application Layer protocol) to bottom (Physical Layer). But the job doesn't stop there; the protocols must be maintained.

Many cell phones, set top boxes, and interactive video games already have lots of onboard memory. Therefore, there is no need to add memory to support the Internet protocols, design engineers say. Once again, the engineers are correct from a mathematical and physical point of view. They conclude that Internet connectivity can be added to the design without changing the hardware.

If the device already has a high-powered 16- or 32-bit processor and enough available memory, they may be right. But the Internet is a dynamic medium and is implemented differently from server to server. It may be necessary to modify the protocols to match those supported by a particular ISP. Furthermore, dial-up parameters to an ISP may change and need to be updated. If a device does not have nonvolatile memory, these updates cannot be stored in the device. Thus, the device may not be able to connect to the Internet if it does not have EEPROM or flash memory.

Next, the device must be programmed to dial up the ISP, log on, authenticate the user account and password, send and receive an e-mail or a web page, or open and close a socket interface, then log off. These commands must be written into the application and stored with the relevant Internet protocols in the device's memory. If the device does not have enough memory to store the Internet protocols and configuration parameters, additional memory must be designed in. Flash memory that can be updated in the field is desirable.

If the device does not have a processor with enough available MIPS to run the Internet protocols, a new processor must be added. A new development environment may be necessary if the manufacturer has selected a new processor. New development tools and kits may be required for programming the processor during the development stage. In addition, a new RTOS may also be required to go with the processor.

Finally, all these elements must be integrated, debugged, tested, deployed in alpha and beta sites, revised, released in the production version, and maintained. The entire development process can take one year or more for a new product. In addition to expending engineers' time, such a long development process can cost success in the marketplace. A long development cycle will delay the time to market for products where early introduction and technological leadership often determine market leadership.

RISKY BUSINESS?

Doing it yourself has serious financial, marketing, and technological risks, especially when dealing with an

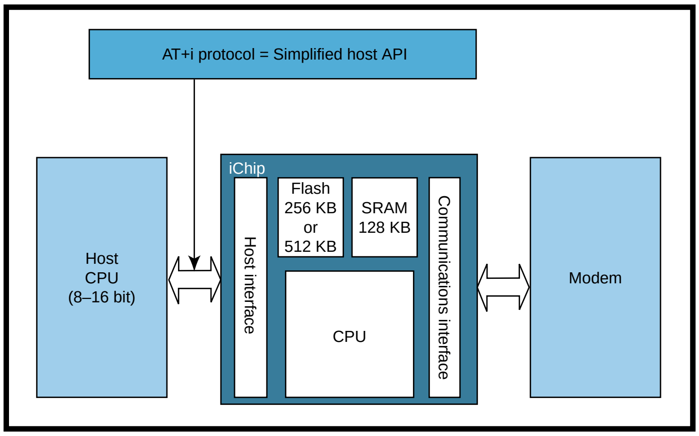


Figure 2—*This Internet controller is a peripheral chip that works in tandem with a device's host processor and mediates the connection to the Internet. Connect One's AT+i protocol simplifies host programming.*

	Current design cost	Internet design at 5k units	Internet design at 10k units
Hardware			
CPU	4	8	8
Flash memory (512 KB)	0	6	6
RAM (128 KB)	2	4	4
Modem	6	10	10
Subtotal	12	28	28
Software			
License amortized per unit			
RTOS at $20k	0	4	2
PPP, TCP/IP, SMTP, POP3, HTTP server at $55k	0	11	5.50
Subtotal	0	15	7.5
Total hard costs	12	43	35.50
Soft costs			
Development time			
Man months at $x/month amortized per unit	0	48	24
Ongoing SW maintenance			
Hourly rate/month/year amortized per unit	0	10	5
New CPU devopment			
Cost/seat at $7.5k x 4 environment amortized per unit	0	6	3
Total soft cost	0	64	32
Grand total	12	107	67.50

Table 1—*Here's an analysis of the costs of doing it yourself at 5 and 10k units. All units are in U.S. dollars.*

unfamiliar technology such as Internet connectivity. Risks include overruns and delays, being late to market, incorrectly implementing the Internet protocols, and not being able to update the protocols.

Should a manufacturer take these risks by stepping outside of its expertise and spending resources in developing Internet connectivity? Or should it invest in perfecting the functionality of its product?

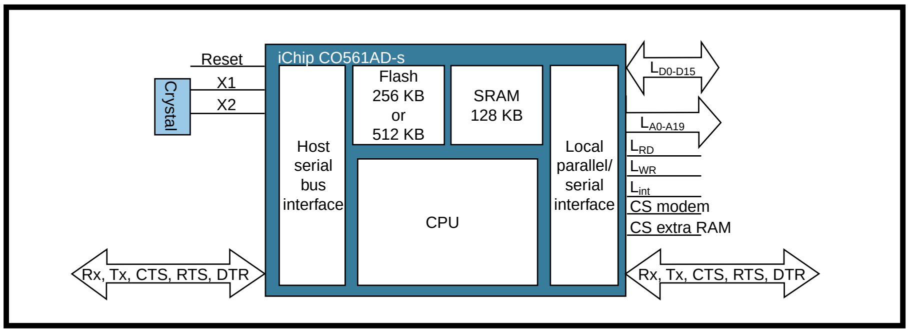


Figure 3—*iChip includes a processor, up to 512 KB of flash memory for storing the Internet protocols, commands, and nonvolatile parameters. It also includes 128 KB to buffer messages.*

COST ANALYSIS

Let's analyze the hard and soft costs of doing it yourself. For small production quantities (less than 10k units), adding dial-up Internet connectivity to an existing design can double the cost of the basic processor, memory, and modem. More memory is required to store the Internet protocols and a higher power processor is needed to run the protocols and modem (one that won't time out on the Internet). Buying or developing a new RTOS and Internet protocol software can increase the cost 25 to 50%. The soft costs of a new development environment, development time, and software maintenance can add another 100 to 150% on top of the hardware and software.

See Table 1 for a case study of upgrading an existing design to include Internet capability. For Table 1, assume the following:

- Current design uses an 8- or low-end 16-bit processor with built-in code memory, no external flash memory, limited RAM, and low-speed modem
- For DIY, upgrading to an Internet-enabled design requires a stronger processor, faster modem, and more memory
- Software licenses are non-transferable, one-time, royalty-free, single-use licenses
- A new CPU environment is required only if the manufacturer upgrades the processor

What about the cost for larger production quantities? Software adds little to the cost, but the soft costs of development time, software maintenance, and a new development environment raise the cost of the hardware plus software 13 to 23%. The main costs are the delay in time to market and the risks.

	Do-it-yourself Internet design at 5k units	Do-it-yourself Internet Design at 10k units	Internet design with iChip at 5k units	Internet design with iChip at 10k units
Hard costs				
Hardware				
CPU	8	8	4	4
Flash (512 KB)	6	6	0	0
RAM (128KB)	4	4	2	2
Modem	10	10	6	6
iChip	0	0	25	20
Subtotal	28	28	37	32
SW license amortized per unit				
RTOS @ $20k	4	2	0	0
PPP, TCP/IP, SMTP, POP3, HTTP server at $55k	11	5.50	0	0
Subtotal	15	7.50	0	0
Total hard costs	43	35.50	37	32
Soft costs				
Development time				
Man months at $x/month	48	24	2	1
Ongoing SW maintenance amortized per unit				
Hourly rate/month/year	10	5	0	0
New CPU development environment				
Cost/seat at $7.5k x 4	6	3	0	0
Total soft costs	64	32	2	1
Grand totals	107	67.50	39	33

Table 2—*Check out the do-it-yourself versus the off-the-shelf version at 5 and 10k units. iChip consists of a processor, the necessary firmware, RAM, and flash memory. You have to add 512 KB of flash memory and 128 KB of SRAM. All units are in U.S. dollars.*

	Do-it-yourself Internet design		Internet design with iChip	
	at 50k units	at 100k units	at 50k units	at 100k units
Hard costs				
Hardware				
CPU	7	6	3.50	3
Flash memory(512 KB)	5	4	4	3.25
RAM (128 KB)	3.50	3	3	2.50
Modem	9	8	5	4.50
iChip	0	0	13	11
Subtotal	24.50	21	28.50	24.25
Software license amortized per unit				
RTOS at $20k	0.40	0.20	0	0
PPP, TCP/IP, SMTP, POP3, HTTP server at $55k	1.10	.55	0	0
Subtotal	1.50	0.75	0	0
Total hard costs	26	21.75	28.50	24.25
Soft costs				
Development time				
Man months at $x/month	4.80	2.40	0.20	0.10
Ongoing SW maintenance amortized per unit				
Hourly rate/month/year	1	0.50	0	0
New CPU development environment				
Cost/seat at $7.5k × 4	0.60	0.30	0	0
Total soft costs	6.40	3.20	0.20	0.10
Grand total	32.40	24.95	28.70	24.35

Table 3—*This table represents off-the-shelf versus do-it-yourself options at 50 and 100k units. For the latter, upgrading to an Internet-enabled design requires a stronger processor, faster modem, and more memory. All units shown are in U.S. dollars.*

Now, let's compare this to the cost of an off-the-shelf solution. Connect One's iChip COS561AD-S Internet Controller is a peripheral chip that works in tandem with a device's host processor and mediates the connection between the host CPU and the Internet (see Figure 2).

iChip is independent of the host processor and operating system. It contains two UARTs for serial interfaces. One is used to communicate with a host CPU, and the other connects to a communication peripheral. The iChip's firmware supports a dial-up modem operating in the range from 2400 bps to 56 kbps. It also supports connectivity to Ethernet LANs and cellular modems. Because iChip includes onboard memory, no other hardware is necessary in order to implement Internet connectivity. There's no need to redesign the basic product with a higher-powered processor to run the Internet protocols, nor to add extra memory to store the protocols, nor to change the operating system. iChip enables a manufacturer to utilize its existing design, thus shortening the time to market for new Internet-enabled products.

The iChip Internet Controller includes onboard flash memory, enabling it to be as dynamic as the Internet. As an Internet controller, iChip keeps the Internet protocols and configuration parameters separate from the application. It includes up to 512 KB of flash memory and 128 KB of SRAM. New protocols and configuration parameters can be downloaded to the iChip (see Figure 3).

Connect One provides a high-level API known as the AT+i protocol, between the host processor and the iChip Internet Controller. This protocol eliminates the need for the manufacturer to subcontract, hire, or use in-house Internet programmers to implement the Internet protocols and commands. With the AT+i command set, a few lines of code transmit plain ASCII text commands from the host to invoke an AT+i command on iChip.

After receiving the AT+i command, iChip switches from command to Internet mode and takes over the communication line. A set of intuitive commands, parameters, and values are defined for sending and receiving e-mail or web pages, and for opening and closing up to five TCP/UDP sockets for direct packet transfers. Commands then activate the appropriate protocol or handshake sequence in the iChip.

The AT+i protocol supports the Hayes AT command set. If iChip receives a regular AT command, it enters transparent mode, enabling the command to pass through directly to a modem. Successful AT commands are returned with an AT result code directly from the modem. Setup parameters for permanent use need to be entered only once on the host using the AT+i command set and the equal sign, which stores them as default settings. Internet commands will become part of the host application, issued as ordered by the host processor. Commands can be entered on a single-session basis when needed and will revert to the default settings after the session is completed.

DECISION TIME

With iChip, there is no need for new hardware or an operating system. All the Internet protocols are also included so no software licenses are required. The combined hardware and software cost for iChip is less than the cost to do it yourself. Development time with iChip is approximately one month versus the typical 12 months for a do-it-yourself solution.

The bottom line is that the iChip off-the-shelf solution costs almost one-third of the cost of the do-it-yourself solution at the 5k level, and half the cost at the 10k level (see Table 2).

At the 50- to 100k-unit level, Connect One provides the iChip processor plus firmware. iChip costs $15 for 50k units and $11 for 100k units. You provide your own flash memory and SRAM. Although the hard cost of using the iChip is greater, the soft cost of doing it yourself is greater because of the long development time (see Table 3). Even at quantities over 500k units, time to market and no risk remain the key selling points.

The Internet controller concept proves its worth financially, commercially, and technologically. The solution saves manufacturers the time, cost, and aggravation to learn that developing and deploying Internet devices is not simple. ■

Alan Singer is vice president of Connect One Ltd. He received his B.S. from Cornell University and MBA from Columbia University. You can reach him at alan@connectone.com.

SOURCE

iChip COS561AD-S Internet Controller
Connect One Semiconductors, Inc.
(408) 986-9602
Fax: (408) 986-9604
www.connectone.com

FEATURE ARTICLE

George Novacek

The Joys of Writing Software

Part 3: Design Tips

In this final segment of George's series, he'll delve into the third aspect of creating excellent software. He tackled planning and testing in Parts 1 and 2, and now George addresses the ultimate test of software quality: fault-tolerance.

In the two previous parts of this series, I talked about planning and testing software with the ultimate goal of eliminating bugs. Hardware designers have long recognized that all components eventually fail, so they design products that do so in a safe and predictable manner. Many software designers make the mistake of thinking that because software doesn't wear out, as long as it's functionally correct and no bugs can be found, their job is done. But programs can skip rail as a result of electrical transients, memory bits can get altered by alpha particles, and impossible input combinations can happen. It is how the software responds to such exceptional conditions that makes it mediocre or great.

Like hardware, designing software to perform the intended function is a small fraction of the task. Making it fault-tolerant so it may recover from the unexpected (or at least fail predictably) is the most difficult part of design. This is what separates the good design from the bad. The customer expects bug-free software and we, as professionals, should strive for nothing less. So, how can we do it?

If you read my previous articles carefully, you already know the answer. There's no magic trick or silver bullet. Bug-free software is the result of discipline and keeping things simple (i.e., testable). The robust software development process forces you to make it a habit. Along the way, it posts checkpoints in the form of design documents and audits. Passing these checkpoints and quality assurance audits forces you to review what you've have already done and helps to uncover problems.

Critics say successful audits do not guarantee working software. You can pass them with flying colors and still produce software that doesn't work. Strictly speaking, the critics are right because the purpose of the audit is to review adherence to the process, not to check the technical correctness or the engineers' work and logic. Theoretically, as an engineer, you could start with a flawed specification, carry through the entire development process while adhering to every detail, and fool the non-engineer auditors along the way. Of course, under normal circumstances, successfully passing the checkpoint also validates the results of the previous one. To end up with flawed software, any organization would have to be either completely incompetent or purposely want to waste time and money to prove that the system can be fooled.

CHECKPOINTS

My checkpoints consist of a number of design documents bound together by the traceability matrix. I've already developed the plans described in Part 1. But before we get down to the design documents, in an organization where several programmers work in a design team, I need to set some design rules, or standards. Not only do I want to make sure the designers' styles are compatible, more importantly, I want to ensure that the code they produce satisfies the overall goal in testability, modularity (i.e., one input, one output rule), safety, and so forth.

I set these rules in the Software Requirements Standards document, which defines the methods, rules, and tools used to develop high-level requirements. Generally, this document defines structured methods for developing the software requirements, nota-

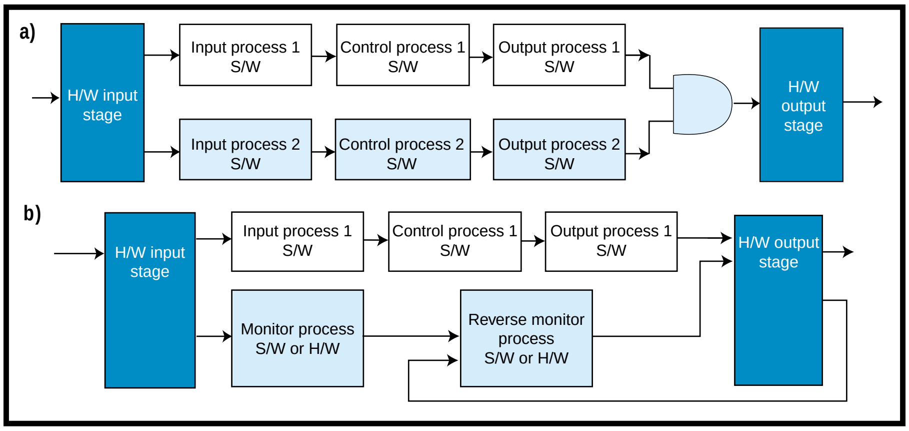


Figure 1a—*Processes 1 and 2 perform the same function through two different paths in the software. The two routines are coded differently and may not be performed by two different processors. They originate from the same hardware stage and end in the same output stage. Their outputs must agree for the hardware output stage to react.* **b**—*The control path performs all the computational functions and drives the output. The monitor reads the output, reverses it into the input, compares it with the input monitor and, if the results agree, enables the output stage.*

tion, formal specification languages, development tools, and constraints. On rare occasions, this document varies for different jobs, so it is like a process manual for the design engineers.

The Software Design Standards document also defines the methods, rules, and tools for the development of architecture and low-level requirements. It addresses design description methods, naming conventions, constraints on design tools, and more. But, the most important part is the definition of design and complexity restrictions, the actual methods that will guarantee testability later. As an example, I will need to limit the number of nested loops and calls, maintain one function, one entry, and one output per module, and exclude recursion and data aliases.

When you deal with safety-critical software, your design standards require dual processing paths, as shown in Figures 1a and 1b. For those of you involved in the design of embedded software, the major task is to avoid unpredictable, safety-critical activity as a result of erroneous behavior like endless loops (or deadlocks), microprocessor erroneous performance, unconditional jumps to undesired code section, erroneous writing and reading to and from reserved memories, and so forth. I'll come back to this when I discuss the design. Clearly, I must address how to perform fault trapping, exception handling, and what the minimum diagnostics requirements are.

The Software Code Standards document specifically defines programming languages, methods, rules, and tools used in coding. It also addresses the format, such as indentation, line length restrictions, blank line usage, headers, and naming conventions.

Customarily, the three subjects are combined into a single document called Software Design Standards, which is truly an engineering manual (not specific to a project) and is used across the entire software development spectrum. Quality assurance auditors are interested in knowing that the engineers follow the well-founded rules (they are spelled out). In good software design standards, there is no room for (misplaced) artistic ambitions of programmers.

SOFTWARE REQUIREMENTS

The Software Requirements document is the most important document, a specification for the software you are about to develop. If you don't get this one right, you'll end up developing something the customer didn't ask for. The basis for creating this specification is the System Requirements document, which is not part of the software documentation suite for the simple reason that it is prepared by the systems engineers. It addresses overall system aspects, such as electromechanical and hydromechanical components and interfaces, power actuators, environmental effects, and so on. It makes no references to software, other than the Hazard Analysis, which will affect the software criticality level. Software designers must study and distil this document and install the software requirements needed for the design. Not surprisingly, engineers with this responsibility must have a good knowledge of and experience with both the software development process, as well as the system design.

My understanding of the system (which will be validated by audit)

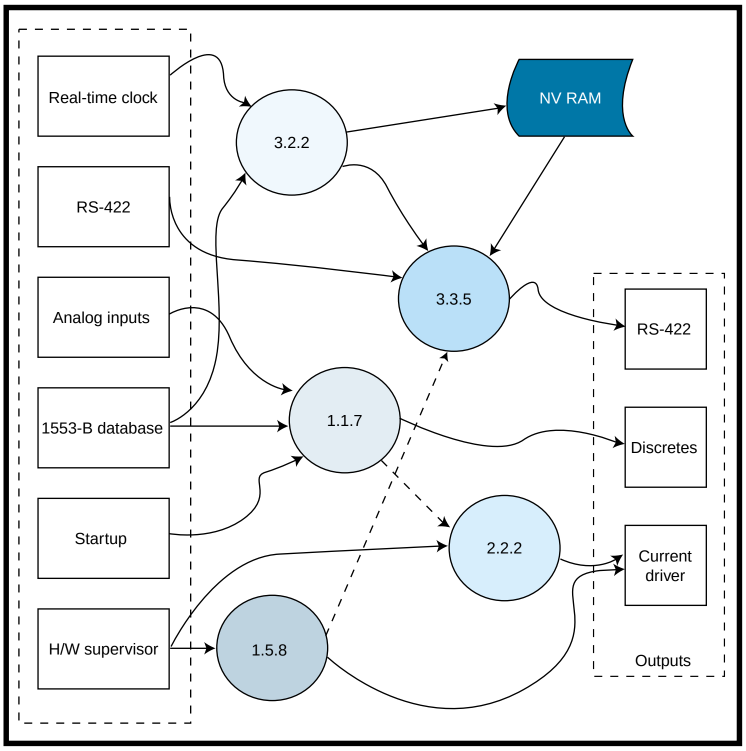


Figure 2—*Command and data flow diagrams define the required processing and assign functions numbers for identification. The numbering system is defined in the Configuration Control Plan and the design standards.*

finds its way, in a concentrated form, into the Plans for Software Aspects of Certification described in Part 1. Although major aspects of the system may be totally unrelated to software, I want to make sure everybody understands (albeit in a simplified form) what the system is about. Software requirements, in conjunction with the hardware definition, must allocate functions between hardware and software and extract from the systems requirements the modes of operation, definition of performance and safety requirements, and how it proposes to handle failure conditions.

Remembering that this is the binding document for the software designers, I must specify performance, precision, accuracy, and all other criteria in detail under all modes of operation. Full definition of memory size, mapping, and timing constraints is crucial. Hardware and software interfaces, communications protocols, frequency, amplitude and bandwidth of signals, and processing algorithms are all vital to the programmers and must be defined. Last but not least, modularity, partitioning, performance monitoring, failure detection, and exception handling are precisely nailed down.

Figure 2 is an example of a high-level mode of operation as defined in the Software Requirements. It defines inputs, outputs, and control or data flow and their relation to individual software functions. Each function is identified by a number that refers to a section in the document containing the specific requirements.

Figure 3 shows how functions are defined. You must identify the purpose, timing constraints, algorithm, equations, inputs and outputs, and whatever is pertinent for the designer to do the job.

And, finally, your Software Requirements document will contain the traceability matrix, which I'll discuss towards the end.

INCREMENTAL DEVELOPMENT

It would be nice if I could start working with the fully defined System Requirements document and work step by step, moving on only when the previous step is done. Unfortunately, life's not that simple. It is the nature of development that we are treading on unbroken ground. More likely, the System Requirements documents will not be complete by the time you receive them, with many TBDs in place where firm data is needed. And even the firm data may prove to be wrong by the time the prototype is built and integration testing starts.

Software development, like any other development, is an iterative process. Time after time, you will have to go back one or two design levels during development, or even back to square one. The secret to maintaining configuration and the design discipline is to stick to the process. You should document everything and, when you need to retrace, do not progress to the next level unless all the currently known requirements have been fully satisfied and the QA audit has given you a clean bill of health.

1.1.7 Propeller Balance

Purpose—upon detection of vibration exceedance, determine .Correction_Weight and .Correction_Weight_Position, which would correct imbalance.

Requirement—propeller balance calculation shall be performed:
- after a Prop_Vib_Maint_Exceedance is detected
- once per flight during a stable flight condition

Calculations:
(Trbn_Vibr.Ampl_Prop_Spd) x (K_TRBN_PROP_SENSITIVITY)=
Prop_Bala_Corr
(Trbn_Vibr.Phase_Angle)+(K_TRBN_PROP_PHASELAG)=
Propeller_Balance_Trbn.Correction_Weight
…
…

Inputs	Description
Prop_Vib_Maint_Exceedance	Indication that prop vibration…
…	…

Output	Description
.Correction Weight	Weight in units
…	…

Figure 3—*This is an excerpt from Software Requirements, which illustrates how functions can be defined using example function Propeller Balance. Calculations, inputs, and outputs are defined.*

DESIGN DESCRIPTION

At this point, can you finally start coding? Absolutely not! Now is the time to design. Take one high-level function (as shown in Figure 2) at a time and decompose it into modules, appropriately numbered, so the relationship with the high-level function is immediately understood. For instance, function 1.1.7 in Figure 3 for balancing propellers could be decomposed to five modules, 1.1.7.1 to 1.1.7.5.

The Software Design Standards document should provide a guideline for decomposition. There is no rule stating that a function must be broken up into modules, but remember why you're doing all this work. You want to deliver bug-free software, and one crucial aspect for accomplishing that is being able to test the modules. And, the simpler the structure of the module, the easier it will be to test (and the more complete the test coverage will be).

Some developers arbitrarily decide that a module will not be allowed to have more than 100 lines of code. If this is the source code, it could still result in some highly complicated modules. On the other hand, if lines of executable code are counted, with today's high-level languages, you may not be able to generate some modules. My preference is to determine the module as the smallest practical function that makes sense to stand on its own and can be effectively tested. Look again at function 1.1.7 in Figure 3. I would create a separate module for every calculation; when the data typing has been included, each will be just a few lines of source code.

Various methods can be used for module design. Some (not many) still like to use flow charts because of the graphics. This works well when I have a presentation where I need to explain a structure to people who are not software developers. The most prevalent method today is the use of pseudo-code, which exists in many, often customized versions. Computer Aided Software Engineering (CASE) tools help you generate pseudo-code, as well as control and data flow diagrams.

In addition to the module decomposition and architecture, your design description must not leave out definitions of inputs and outputs, data dictionary and data flow, address design constraints (resource limitations), and how you're going to test the module. In fact, testing must always be on your mind while you're defining the modules. Discovering that the module is not testable, even though it appears to be working fine, means the software cannot be certified.

One of the most important design aspects is the module's ability to handle incorrect data. Division by zero, zero or negative operand on logarithms, and negative operand to square root can all play havoc on your routines. The design must be able to tolerate such situations even if they are not expected to arise. It is easy to forget details, so never skimp on data typing whenever the module could be affected by incorrect data. Later, your test routine must verify that the data typing is effective. See Part 2 of this series for possible strategies.

Watchdog timers are helpful, provided their limitations are understood and they are not viewed as a panacea. A useful implementation has every subroutine exercised during the main program loop, upon which a flag is set. Then, at each completion of the main program loop, a roll call is performed on the flags, but they only get cleared and the watchdog is reset for another round when they have all been set in proper order.

The structure that can surely kill a module (making it difficult, if not completely untestable) is a nested loop. Figure 4 shows three fundamental types. A concatenated loop consists of two single loops (Figure 4a). These are simple, effective, and should be adequate for most applications. A single loop can be tested with eight test cases— 0, min – 1, min, min + 1, max, max – 1, max + 1, and one (or a few) typical middle values. A concatenated loop can be treated like several single ones. It works fine with the dual concatenated loop, requiring 16 tests for reasonable assurance. The nested loop (see Figure 4b) makes life difficult. If you need eight tests for a single loop, the double-nested loop will need 64 tests, a triple-nested loop will need 512 tests, which is just about the maximum reasonable limit. I have always considered the double-nested loop the practical maximum. Anything more than that is hard to test, and if the function cannot be done in a different, more acceptable manner, perhaps a review of the overall architecture should be considered.

Figure 4c shows an example of one monstrosity you would be well advised to stay away from. It is extremely hard (if not impossible) to test and the potential gain arising from its design simplicity will be quickly spent by the potential trouble in testing.

At the end of the design effort is the Design Description document, from which any competent programmer can take and write code. Please note the fundamental distinction between the software designer and the programmer. The programmer is like an interpreter who needs to know the language to efficiently convert the designer's creation into code. Often, the designer and the programmer are one person, but they don't have to be. The bottom line is that after the designer and QA have signed off on the design description, everything is laid out for the person wearing the programmer's hat (even though he or she may be the person who prepared the document). With the programmer's hat on, this person is not expected to check the logic, develop algorithms, worry about timing, or alter the design as laid out in the document. All of this should have been done during the design, but this is not to suggest that the programmer can put on blinders or ignore the discovery of any possible errors.

SOURCE AND EXECUTABLE CODE

Finally comes what many consider the alpha and omega of programming, the stuff the proverbial Twinkie-munching, Coke-guzzling programmers are made of. In reality, the actual coding is an insignificant part of the software design process. In the DO-178B standard, it is given a measly two paragraphs of seven lines total.

Of course coding is important, especially good coding, but it is quickly becoming a mechanical discipline. Compilers do a good job of syntax checking. Also, there are tools such as CASE, fuzzy logic, Visual Basic, or C development systems that accept design in a graphical or pseudo-code form, then spit out source or executable code. I don't dare say how reliable such mechanically generated code is, especially for critical embedded applications. My experiments with fuzzy logic produced decent C source. And because software development cost reduction is on everybody's mind, there is a lot of activity to produce better and better tools.

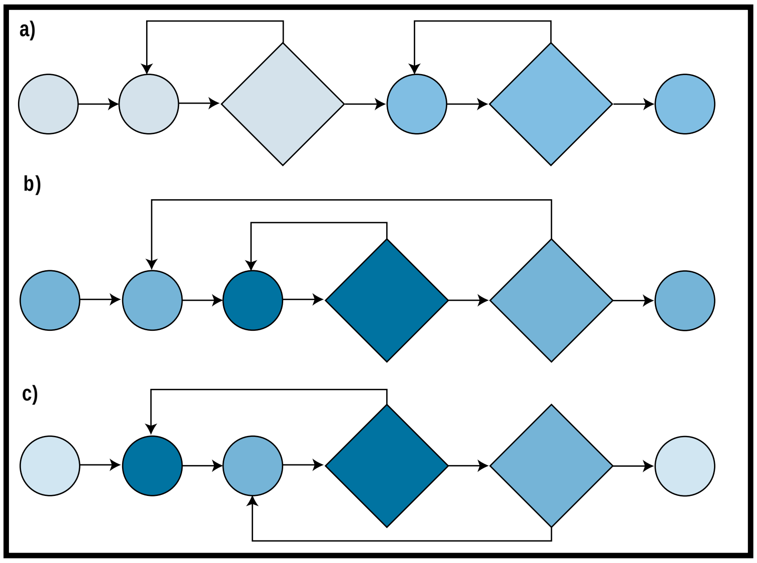


Figure 4—*Knowing what structures to avoid can save you a lot of time,* **(a)** *shows two concatenated single loops;* **(b)** *is a double-nested; and* **(c)** *demonstrates a loop that is difficult to test.*

VERIFICATION TEST CASES AND PROCEDURES

If I followed the procedures and designed tests while designing modules, the Software Verification Test Cases and Procedures document would be completed at the same time as the Design Description.

This article cannot delve into the details of software testing. The References listed at the end should give you a good start for further study. I discussed the subject of testing at length in Part 2 and will reiterate (at the risk of becoming repetitive) how important it is to consider integration and testing during the module design.

Usually, the first extremely useful test is the code walkthrough. This cannot be automated and, in essence, is nothing more than reading the code to make sure it reflects the design description. It is also followed by a review of the traceability matrix.

The rest of the tests should be automated as much as possible. At the very minimum, C1 and C2 testing should be done, which includes 100% statement coverage and 100% branch coverage. The sky's the limit after that. Automated software tools are now available to establish test coverage and to assist in design and performance of the tests. Among the most commonly used are path, transaction, data flow, domain logic, and state and transition testing. Syntax testing is at the top of the list. This may be obvious, but because it is done automatically by every compiler, dwelling on it is superfluous.

In terms of test performance, it can be done on a target system or a host computer. The market offers many simulators, emulators, and entire development systems depending on the type of processor, language, and your solvency. These tools are available for less than $100 all the way up to $100,000. When complete, the test results are issued in a separate document called Test Results.

SOFTWARE ACCOMPLISHMENT

With the tests successfully behind me, before I can certify the software, the last document I need to issue is the Accomplishment Summary. It is a brief summary of the documents generated during the development process and the configuration index. It addresses subjects such as the system and software overview, certification considerations, change history, configuration data, and the compliance statement, which tells the customer that the job has been done in full compliance with the specification.

The major tool to prove the compliance is the traceability matrix. The traceability matrix is a database that is generated automatically by development tools. In the past, when it was created and maintained manually, it represented a major effort and cost. In the first column of the matrix, customer requirements raised in the System Requirements document are listed, usually referred by their paragraph number. In the second column, corresponding paragraphs in the Software Requirements document are quoted. The implementation in the design description comes next, then the source code, and finally the test cases and test results. Armed with the traceability matrix, you can take any customer requirement from the specification and show how it has been addressed through the design process, which part of the code satisfies it, and how it was tested and proven correct. Conversely, you can take any line of source code and trace it back to the original requirement.

The purpose of the traceability matrix is manifold. First, it helps to make sure that all the customers' requirements have been satisfied and documented. Second, it allows you to verify test coverage and show that all the features have been tested. It also quickly uncovers dead code or undocumented features and hooks, which are an absolute no-no. And finally, should a bug slip through in spite of your best effort, understanding the symptoms and having the traceability map on hand is an invaluable tool for zeroing in and fixing it.

CLOSING WORDS

This concludes my three-part series on software development. As I began to write, I had to come to terms with the inescapable reality of the scope of these articles, which allowed me only to scratch the surface. The most difficult part was selecting just the highlights. I agonized over much of the material that I had to leave out.

Through years of engineering experience, I was involved in the inevitable investigations of product failures. The hardest concept for software engineers to swallow is their culpability for software behavior in response to abnormal conditions. Today, software is the brain of all the sophisticated machinery surrounding us. If this machinery, as a result of an external stimulus, goes out of control because the program got derailed, the software design is unacceptable. Period!

It makes no difference that the subsequent tests found no formal error, thus declaring it bug-free. In my book, such software contains the biggest bug of all: it is not fault-tolerant and allowed itself to go out of control unchecked. An applications program that freezes when you hit the most unlikely combination on the keyboard is not fault-tolerant and flawed. I am serious about this and every software engineer worth his salt should be too.

This series is merely an overview of the main issues facing the software engineer. Those of you interested in deeper knowledge will find excellent sources in the References section.

George Novacek has 30 years of experience in circuit design and embedded controllers. He currently is the general manager of Messier-Dowty Electronics, a division of Messier-Dowty International, the world's largest manufacturer of landing-gear systems. You may reach him at gnovacek@nexicom.net.

REFERENCES

Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc., Washington, D.C., 1992.

M.R. Lyu, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

Beizer, *Software Testing Techniques*, Van Nostrand Reinhold Company, NY, 1990.

Ralston, *Encyclopedia of Computer Science and Engineering*, Second Edition, Van Nostrand Reinhold Company, New York, 1983.

George Novacek, "Defects For Sale Revisited," *Circuit Cellar Online*, February, 2000.

SOURCE

LDRA Ltd.
+ 44 (0) 151 708 8505
Fax:+ 44 (0) 151 709 2027
www.ldra.com

NOUVEAU*PC*

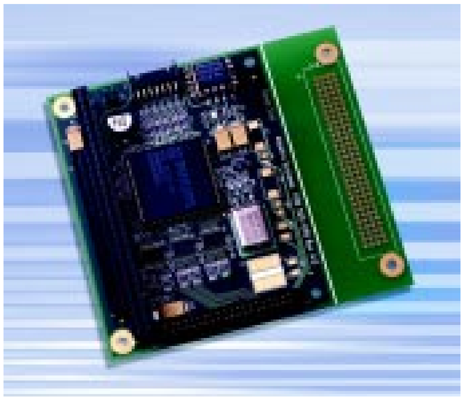
Edited by Rick Prescott

DIGITAL VIDEO MODULE

The **DV-104** is a digital video module for embedded applications. The DV-104 digitizes live video (NTSC, PAL, or SECAM) and outputs the digitized stream on a zoomed video bus. When combined with a zoomed video-capable single-board computer (SBC), the result is a system that can display live video in a window on a SVGA monitor (CRT or flat panel) and capture video at rates of 20 to 30 frames per second.

The DV-104 comes in three forms: PC/104, PC/104-*plus*, and standalone. The standalone form is a compact 3.6″ × 2.8″. Its features include video digitization up to 640 × 480 pixels (NTSC), six composite or three S-Video inputs (or a combination), and 3.3-V digital video (zoomed video) output.

The DV-104/Venus combination is supported by a software package that defines an API that allows applications to easily display, manipulate, and capture live video in both GUI and non-GUI environments. Versions of the software are available for Windows, NX, and Linux. Sample applications demonstrate "video windowing" (various CPU signals shown on a single display) for QNX running Photon microGUI and Linux running X-Windows. The Windows version supports DirectX and an OCX control.



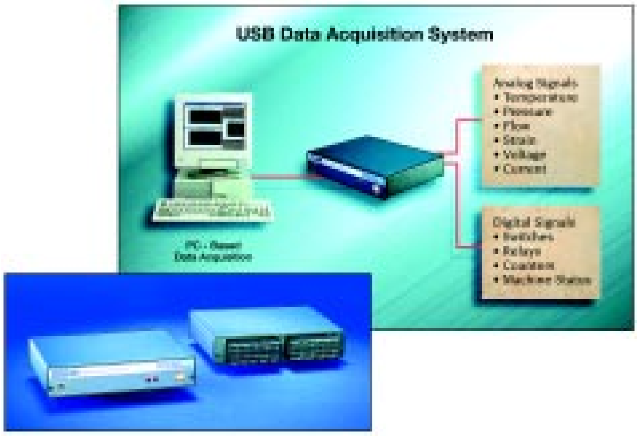
The DV-104-*plus* Venus combination is ideal for information kiosks, video-on-demand systems, game applications, security systems, and more.

Versions are available for **$120** in quantities of 100.

Adastra Systems
(510) 732-6900
Fax: (510) 732-7655
www.adastra.com

USB DATA ACQUISITION SYSTEMS

The portable **UDASTM USB Data Acquisition Systems** is designed with ease of use and portability in mind. UDAS systems provide an out-of-the-box alternative to plug-in PC data acquisition boards. The PC auto detects the addition or removal of the USB data acquisition system. Some typical applications for USB I/O systems are automated test and measurement, data logging, temperature measurement, laboratory automation, portable data acquisition, production test, electronics test, and research and development.



Several USB Data Acquisition models are available, including some systems featuring a mix of analog and digital I/O channels. They feature a bus-powered design, allowing the system to be powered from the PC through the USB port. The systems can be configured for 16SE/8DIF 12-bit analog-input channels with programmable gains, two 12-bit analog-output channels, eight digital-input channels, eight digital-output channels, and one 16-bit high-speed counter channel. Two termination options allow users to select the signal conditioning design.

Other models interface with a variety of external termination panels for interfacing with sensors and transducers. They also feature built-in termination. These systems accommodate direct thermocouple measurement and are ideal for use with portable laptop computers. Board-only versions are offered. A variety of support software is available.

Prices start at **$700** for OEM versions in single quantities, and the plug-and-play models range from **$895** to **$1060**.

Intelligent Instrumentation, Inc.
(800) 685-9911
Fax: (520) 573-0522
www.instrument.com

NOUVEAU *PC*

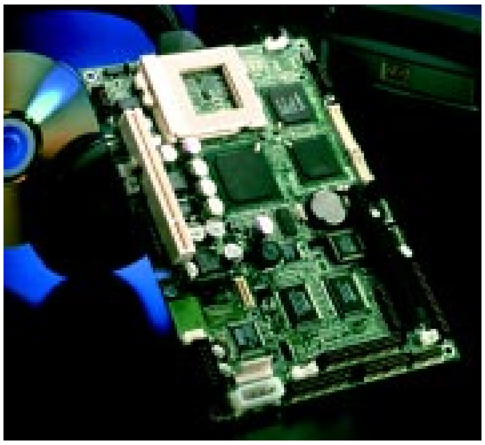
AUDIO ADDED TO PENTIUM III SINGLE-BOARD COMPUTER

Audio capabilities have been added to a Pentium III single-board computer (SBC) to equip it for operation in music preview kiosks, multilingual public information systems, arcade games, and other applications where graphics, processing speed, and sound are important.

The new **PCM-9574** has a Socket 370 for a Celeron or Pentium III processor, up to 256 MB of small-outline SDRAM, up to 4 MB of frame buffer SDRAM for graphics, a hardware-assisted MPEG II accelerator, and a CompactFlashTM socket. And it has SGVA display support (with an optional PanelLink transmitter chipset for longer-distance connections), a 2X accelerated graphics port, IDE channel, PC/104 connector, Ethernet, software modem, and one extra PC1 slot. All other pieces and ports required are included.

Audio components include a Multistream Direct Sound controller with direct sound 3-D acceleration, microphone in, line in, line out, and CD in. All of these are packaged in the form factor of a 5.25 CD-ROM.

The PCM-9574 is priced at **$462** in quantities of 100 without processor or SDRAM.



Advantech Technologies, Inc.
(800) 866-6008
Fax: (949) 789-7179
www.advantech.com/epc

MICRO WEBTARGET HARDWARE/SOFTWARE DESIGN KIT

The **Micro WebTarget** is a complete hardware/software Java technology solution designed for the development of resource-constrained embedded Internet devices. Based on the Connected Limited Device Configuration (CLDC) component of the Java 2 Platform, the Micro WebTarget is aimed at developers of embedded designs like cell phones, pagers, and other mobile devices.

The form factor of the Micro WebTarget prototyping board, measuring 3 cm × 6 cm, demonstrates the potential benefits of Java-capable mobile applications, in which size is a major issue. A single processor core from Hyperstone Electronics combines the functionality of a 32-bit RISC processor with a 16-or 32-bit DSP unit. The processor core provides 200 MOPS of processing power at 50 MHz, giving the Java Virtual Machine (JVM) the execution speed required to operate even in demanding real-time applications.

The Micro WebTarget includes a built-in Ethernet controller and interface transformer support for a direct network connection. Onboard DRAM, flash l-MB DRAM, and 1-MB flash memory are available. HyNetOSTM, a real-time, multitasking network operating system optimized to run on the Hyperstone RISC/DSP architecture, is included. HyNetOS provides the device management for network connectivity and peripheral I/O. The WebTarget also includes a TCP/IP stack and protocol manager.

Application development can be in Java, C, or a combination of both on any PC or workstation. A C compiler is provided with the development kit. The onboard flash memory allows upgrades through the network interface. The Micro WebTarget can be mounted on an optional shuttle board.

The kit is available for **$2350**. This includes the MWSl/X Micro WebTarget starter board, MWS shuttle expansion board, HyNetOS base version with an Evaluation License, Hyperstone Development Tools on CD-ROM, plus one year of technical support and updates.

Smart Network Devices (SND)
+49-(0)2131-223267
Fax: +49-(0)2131-223269
www.smartnd.com

Ingo Cyliax

Catching the PCI Bus

Part 2: Configuration

Getting up to speed on the PCI bus takes more than just an understanding of the protocol. This month, Ingo digs into PCI configuration to show us how the puzzle comes together. Listen up, because this is Ingo's last stop on the PCI bus, for now.

Last month I described the basic peripheral component interconnect (PCI) bus protocol, which is fairly simple—after all, it has to be implemented in hardware. If you'll recall, PCI is a multiplexed address data bus. The device that needs to access another device will broadcast the address and type of transfer it needs to perform and then transfer data to or from the device. The initiator or target can abort the transfers when they are done or if there is an error condition.

But, protocol isn't all you need to know. This month, I'm going to describe PCI configuration, an important part of PCI-based systems.

CONFIGURATION

Older buses like VMEBus and ISAbus, which are primarily buses that are being replaced with PCI, typically use configuration jumpers or switches to set the bus address and interrupt level of peripheral cards. This can lead to several problems.

The software that runs on these systems cannot determine the address of cards. It has to use prior knowledge about where the card's base address might be. IBM PC serial ports are at well-known locations, and systems usually need large technical manuals to describe what address cards need to be configured and what the address selection jumpers and switches actually mean. Clearly, it's a large effort to keep the technical configuration manual in sync with the operating system.

Another problem is the chance for address collisions in the system. This happens when some cards are not configured well and their address ranges or interrupt request levels overlap (e.g., two ISAbus serial cards that claim the same I/O port range). The software will find the card at the expected address, but things will get confused when it tries to use the serial cards. I'm sure many of you have run into this. It's annoying for PC users, and unless you've worked with large VMEbus systems, you haven't seen anything yet.

PC manufacturers tried to address the problem by introducing plug-and-play (PNP) for ISAbus peripherals, more popularly known as plug-and-pray for obvious reasons. Although it is easy for the OS to query the cards and find out what's there, PNP cards can still be mixed with old cards to create

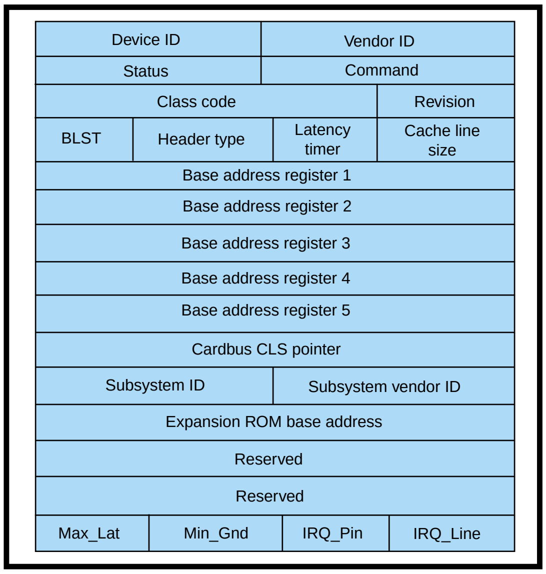


Figure 1—*Here is the PCI configuration register space layout. This register space is implemented in the PCI chipset and controls the PCI interface of the device or chip.*

conflicts. Also, many legacy device drivers still assume that I/O addresses can only be at certain well-known addresses.

However, PNP is a step in the right direction and the configuration support draws heavily on PNP. PCI starts from scratch, so a minimal set of configuration functions has to be implemented to begin working in this environment.

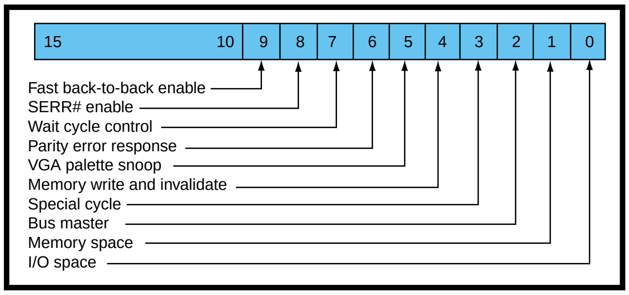


Figure 2—*The PCI command register layout is used to enable various aspects of the PCI interface for the device.*

SO, HOW DOES IT WORK?

Each PCI agent has to support configuration read commands. Configuration reads are bus cycles just like regular I/O or memory reads, except that they access a special configuration register space on the agent. There are several versions of this register layout. The layout for a Type 0 configuration register is shown in Figure 1.

Each agent starts with a device and vendor ID. The vendor ID is a unique 16-bit number assigned by the PCI special interest group (SIG). The device ID is a 16-bit number that is assigned by the vendor. The idea of the vendor/device ID pair is to uniquely identify a peripheral so that a device driver can be found.

The configuration register space also implements command and status registers. These registers are for the PCI interface of the card or chipset and have nothing to do with any possible command or status register that is needed to control the device.

The PCI command register allows the PCI subsystem to control how the card behaves in the system. The command register has enables for I/O and memory-mapping registers that control whether or not the card's address decode will work for the I/O and memory address space.

A master enable will work if the card can act as a bus master. There are enables that control cache snooping and video functions, as well as tracking whether the card responds to parity errors or generates system error conditions. If the card does not implement some of the functions and features, it will simply implement the corresponding bit as read-only set to zero. The system will detect that the feature is not implemented when it tries to enable the feature and the command register does not respond. Figure 2 shows the command register layout.

The status register tells the system if it has features such as 66-MHz bus speed support, fast back-to-back support, and how fast the card can decode device accesses. The status register also reports various errors that could have occurred during aborts. Figure 3 shows the status register layout.

The revision code register is an 8-bit number that is assigned by the vendor to indicate a particular version of a card. This can be used by the driver software to enable certain features or avoid bugs.

Using only the vendor ID and device ID to identify the card and find a matching device driver can be cumbersome. Also, for some critical system functions like a disk controller or keyboard controller, you need to be able to use the device before you have access to a device driver. The class code register identifies devices and places them in several device types. The class code is made up of three 8-bit fields—the class code, subclass code, and programming interface (Prog I/F) field. Table 1 shows the class, subclass, and program function codes.

If the class code is 0x03 (display controllers) and the subclass or Prog I/F is 0x00/0x00, then you're dealing with a basic VGA graphics controller that a BIOS or low-level OS routine can use as a basic console. After the OS is loaded, along with extra functionality, a specific device driver can be used that extends the device's functionality, if possible.

Finally, the header type identifies the format of the configuration header and whether or not the device implements a multifunction device. Multifunction devices are cards that can implement more than one device function in one card. A modem plus Ethernet card is a multifunction device. These devices and other header types are beyond the scope of this article. Check out the Resources section at the end for more information.

REGISTERS

The address and address space (I/O and memory) are programmed using the configuration space base address registers (BARs). There can be up to six address ranges for memory and I/O, one for an expansion ROM and another for a CardBus CIS pointer. Expansion ROMs are used by the BIOS (on PCs) to extend boot capabilities. For example, they implement the boot code needed to boot the system from a remote server over a network. Not all address registers need to be implemented, and most devices don't implement more than one or two different address ranges, perhaps only an I/O port range and one memory range.

The address registers can also be read-only, in which case the device might have a hard-coded address for decoding. The keyboard controller is

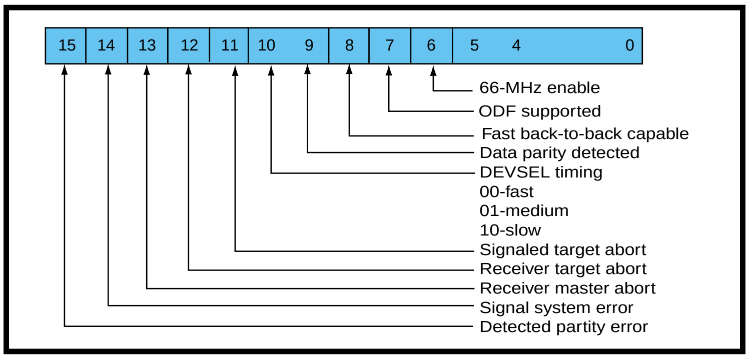


Figure 3—*From the PCI status register layout you can find out about various errors that may have occurred, as well as the capabilities of this device.*

one example of this. Address registers can also be preprogrammed to select at a specific address, yet are changeable if the device needs to be re-mapped elsewhere.

The address registers come in two basic flavors—memory and I/O base address register format. Figure 4 shows the layout. They are identified by the least significant bit—a one to identify I/O, and a zero for memory. I/O base address registers are easier to cover, so I'll start the discussion there.

The base address field has 30 bits in the register. Low-order bits in this field are set to zero to indicate the block size of registers that the device needs to use. Typically, to size the I/O register space, a value of 0xffffffff is written to the BAR. The first bit from the least significant end of that set indicates the size of the register (e.g., if 0xffffff00 is

Listing 1—*Here is a list of the PCI table that the BIOS built while booting the system. Under Linux this list can be seen with the file* `/proc/pci`.

```
PCI devices found:
  Bus  0, device   0, function  0:
    Host bridge: Intel 440BX - 82443BX Host (no AGP) (rev 2).
        Medium devsel.  Master Capable.  Latency=64.
        Prefetchable 32 bit memory at 0x0 [0x8].
  Bus  0, device   2, function  0:
      CardBus bridge: Texas Instruments PCI1250 (rev 2).
        Medium devsel.  Master Capable.  Latency=168.
              Min Gnt=192.Max Lat=7.
  Non-prefetchable 32 bit memory at 0x50000000 [0x50000000].
  Bus  0, device   2, function  1:
      CardBus bridge: Texas Instruments PCI1250 (rev 2).
        Medium devsel.  Master Capable.  Latency=168.
              Min Gnt=192.Max Lat=7.
  Non-prefetchable 32 bit memory at 0x51000000 [0x51000000].
  Bus  0, device   3, function  0:
     VGA compatible controller: Neomagic MagicGraph NM2160
       (rev 1).
        Medium devsel.  Fast back-to-back capable.  IRQ 11.
Master Capable.  No bursts.  Min Gnt=16.Max Lat=255.
Prefetchable 32 bit memory at 0x48000000 [0x48000008].
Non-prefetchable 32 bit memory at 0x49000000 [0x49000000].
Non-prefetchable 32 bit memory at 0x49400000 [0x49400000].
  Bus  0, device   6, function  0:
     Bridge: Intel 82371AB PIIX4 ISA (rev 2).
        Medium devsel.  Fast back-to-back capable.
              Master Capable.  No bursts.
  Bus  0, device   6, function  1:
     IDE interface: Intel 82371AB PIIX4 IDE (rev 1).
        Medium devsel.  Fast back-to-back capable.
              Master Capable.  Latency=32.
        I/O at 0xfcf0 [0xfcf1].
  Bus  0, device   6, function  2:
     USB Controller: Intel 82371AB PIIX4 USB (rev 1).
        Medium devsel.  Fast back-to-back capable.  IRQ 11.
Master Capable.  Latency=48.
        I/O at 0x8000 [0x8001].
  Bus  0, device   6, function  3:
     Bridge: Intel 82371AB PIIX4 ACPI (rev 2).
        Medium devsel.  Fast back-to-back capable.
```

read back, you know that this BAR decodes a 16-byte register block). After it's sized, the system sets the BAR to where the device should be decoded in the I/O address space. A serial port might be set to 0x000003f8. Note that the BAR is 32 bits, but PCs normally use only 16 bits.

Card designers are encouraged to allow mapping of their I/O devices in memory space because it is larger (4 GB) than the I/O address space (64 KB) on PCs. Also, several processor architectures like the Motorola 680000 and PowerPC series of processors might not be able to access devices that are only I/O-mappable.

The memory BAR works similarly, however, there is also a 2-bit field in the BAR that indicates where this device would like to be mapped in the memory space. This is important for devices that can only deal with 1-MB or 32-bit address spaces. If 64-bit addresses are allowed, then the BAR takes two words of configuration register space instead of one. Also, there is a pre-fetch bit that indicates whether or not the instruction may be pre-fetched from this memory. The sizing and address selection works the same as the I/O base register select except that the minimum memory block size is 16 bytes and the minimum I/O block size is four bytes (one 32-bit word).

PUZZLE PIECES

At this point, you might be asking yourself how all this fits together. In order to get a better picture, let's look at what happens to the PCI bus and devices when a PC tries to boot.

When the PC boots, the BIOS initializes the host-to-PCI bridge, which is chipset-dependent, and the BIOS initialization routines have to match the bridge controller chipset used on the motherboard. After the host-to-PCI bridge is initialized, the processor can generate configuration read and write requests on the attached PCI bus.

Configuration accesses to PCI devices are decoded using physical device selects for each slot. This host-to-PCI bridge has the device selects used by the processor to test each slot and read the configuration space in each attached slot. Remember that even though PCI chips and bridges can be on the motherboard, they still take up a virtual slot. If the processor finds another PCI or PCI bus bridge and initializes it, it continues to probe the slots beyond the bridge. The BIOS records each device it finds and its physical location (bus and device) in a table.

Now that all of the devices have been enumerated, the BIOS will find appropriate system resources to boot the system. This includes the programmable interrupt controller, the DMA controller, RTC, keyboard controller, and a VGA or compatible display adapter. For each, the BIOS will use a configuration write cycle to program the BAR of each device and map it into the proper location. For most of the system resources, this will be in a standard location so that legacy software will be able to find and treat them as standard PC peripherals.

The BIOS then goes about booting a system in the standard way of looking for expansion ROMs and appropriate boot devices. When the operating system is loaded, the OS will either enumerate all of the PCI devices again or use PCI BIOS calls to extract the list of devices the BIOS has already enumerated during the boot process. The OS then will try to load appropriate device drivers. For some devices like

Listing 2—*A sample device driver snippet shows how to find a PCI device or card and how to set the I/O base address using the base address register in the configuration register space.*

```
mydevice_probe()
  ....
  for(dev = pci_devices; dev ; dev = dev->next){
    /*printk("%04x %04x\n",dev->vendor,dev->device); /**/
    if(dev->vendor == MY_VEND_DEV_ID){
      ... initialize ...
           printk("--> ");
        pci_read_config_dword(dev,PCI_COMMAND,&val);
           printk("cmd: %x ",val);
           printk("\n");
           printk("--> ");
          pci_read_config_dword(dev,PCI_BASE_ADDRESS_0,&val);
           printk("bar: %x ",val);
           val = 0xffffffff;
          pci_write_config_dword(dev,PCI_BASE_ADDRESS_0,val);
          pci_read_config_dword(dev,PCI_BASE_ADDRESS_0,&val);
           printk("bar: %x ",val);
           printk("size: %d bytes",(~val | 0x3)+1); /**/
           printk("\n");
           val = IO_BASE;
          pci_write_config_dword(dev,PCI_BASE_ADDRESS_0,val);
          pci_read_config_dword(dev,PCI_BASE_ADDRESS_0,&val);
           printk("bar: %x ",val);
      val = 0x0005;   /* IO and Master enable */
        pci_write_config_dword(dev,PCI_COMMAND,val);
        pci_read_config_dword(dev,PCI_COMMAND,&val);
           printk("cmd: %x ",val);
           printk("\n");
                   break;
             }
  }
  ...
}
```

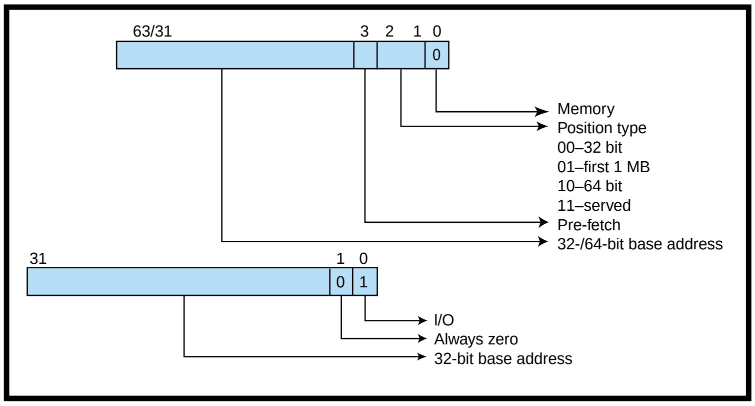


Figure 4—*The base address registers can be in one of two formats, depending on whether or not the low-order bit is zero. Memory base configuration registers also come in 32- or 64-bit address width varieties.*

VGA cards, the OS might only use generic drivers. However, if better drivers are available for a specific card that enables features, it may try to match a driver based on the vendor and device ID found while enumerating.

After an operating system is up and running (Linux on my laptop, for example), I can look at the list of PCI peripherals (see Listing 1).

Linux has a convenient interface for looking at the PCI BIOS table at the command interpreter level. However, a device driver will access a kernel level list that has been built using PCI BIOS calls. Listing 2 shows how a card might look through the list of PCI devices and find which one it needs to use.

So, this routine will search the list of PCI devices. In this case, I'm looking for a card that matches `MY_VEND_DEV_ID`. Then I can use the physical location stored in the device structure pointed to by the device pointer to access the configuration space of the card. I included the command register for illustration purposes (see Figure 2).

The PCI bus will map all of the devices to free memory and I/O regions if possible. It's OK to use the regions assigned by the BIOS. But in this case, I want to map the card at a specific device address (`IO_BASE`) in I/O space.

To do this, I test the BAR by writing a 0xffffffff to it and decoding the expected register size. Most device drivers would already know the register size, but this illustrates how it's done. I then set the address to `IO_BASE` and turned on I/O mapping to enable master mode in the command register. This card uses master transfers for some functions, so I turned the master enable bit on.

The initialization routine is run in a kernel-level module and is dynamically loaded. The card shows up in the I/O port space after it's mapped and then a user-level application can actually use it. Although this example is Linux-centric, the same basic scheme applies to other operating systems. The basic steps are to find the card by vendor and device ID, use the physical bus and device location to map and turn on the card, and finally, use your regular device driver scheme for talking to the card.

Class	Subclass	Description
0x01	0x00	SCSI controller
0x01	0x01	IDE Controller
0x01	0x02	Floppy disk controller
0x01	0x03	IPI controller
0x01	0x04	RAID controller
0x02	0x00	Ethernet controller
0x02	0x01	Token ring controller
0x03	0x00	VGA-compatible controller
0x06	0x00	Host/PCI bridge
0x06	0x01	PCI/ISA bridge
0x06	0x02	PCI/EISA bridge
0x06	0x03	PCI/Micro channel bridge
0x06	0x04	PCI/PCI bridge
0x06	0x05	PCI/PCMCIA bridge
0x06	0x06	PCI/NuBus bridge
0x06	0x07	PCI/CardBus bridge
0x07	0x00	8250-compatible serial controller
0x07	0x00	16450-compatible serial controller
0x07	0x00	16550-compatible serial controller
0x07	0x01	Parallel port
0x07	0x01	Bi-Directional parallel port
0x07	0x01	ECP 1.x-compatible parallel port
0x0c	0x00	Firewire (IEEE-1394)
0x0c	0x03	USB (Universal Serial Bus)

Table 1—*This is a list of device class codes and subclass codes. These codes are used by the system to find devices for specific system functions.*

PCI LAYERS

This concludes my two-part series about PCI in a nutshell. Of course, there are several more layers than offered in this short series. The Resources section should point you in the right direction from here. However, for most device driver work, all you need to know about PCI is how to find your card and map it to the address you want to use. After that, it pretty much behaves like the device implemented on the card and all of the PCI business is transparent. If you found this series informative, let me know if you want to read more about PCI or perhaps find out how to build your own card. ■

Ingo Cyliax is the Sr. Hardware Engineer at Derivation Systems Inc. (DSI) where he designs and builds embedded systems and hardware components. DSI is the leader in formally synthesized FPGA cores and specializes in embedded Java technology. Ingo has been writing on various topics ranging from real-time operating systems to nuts-and-bolts hardware issues for several years.

SOFTWARE

The list of PCI device class codes and solo-class codes is available on the *Circuit Cellar* web site.

RESOURCES

T. Shanley and D. Anderson, *PCI System Architecture*, Third Edition, Mindshare, Inc., Addison Wesley, Reading, MA, 1995.

PCI Special Interest Group, *PCI Local Bus Specification*, Revision 2.1, Portland, OR, June 1, 1995.

PCI Special Interest Group, *PCI Compliance Checklist*, Revision 2.1, Portland, OR, 1995.

B. Dipert, *The PCI Handbook*, Annabooks, San Diego, CA, 1995.

M. Predko, *PC Interfacing Pocket Reference*, McGraw-Hill, New York, NY, 1999.

EPC
Applied PCs

Fred Eady

Rabbit Season

Part 2: Jackrabbit Development Board

Last month we met the Jackrabbit dev board, now it's time to dig a little deeper. After a little C, then it's on to some IP, and before this series is done, you'll have all the info you need to get your Ethernet interface hopping along.

The Rabbit 2000 may have been as still as a cold rock in the snow since the last installment, but the Rabbit's innards (that's Southern for internal organs) have been hard at work. In fact, a little Rabbit indigestion has taken place since last time. (Must have been those spicy carrots.)

I mentioned that the Rabbit is I/O rich to the point of not needing the address and data bus to affect the Ethernet interface. Well, I'm going to have to call in some favors on that statement. The Rabbit can use its built-in data and address bus to interface to the CS8900A Ethernet Controller. But, I'm not building the Rabbit from scratch. Because I'm depending on the architecture of the commercially available Jackrabbit board from Z-World (see Photo 1) and I'm not designing my own printed circuit board based on the Rabbit IC, the privilege of laying down a desired bus structure on a printed circuit board does not exist. As Photo 2 shows, the data and address bus lines aren't brought out to the Rabbit's monkey board (i.e., a board you monkey around with) either. The Rabbit folks have a more accurate name for the monkey board. They call it the Jackrabbit Development Board.

So, I'll change the initial design before it's even a design. Rabbit Port A will now become the official Ethernet bidirectional data bus. Ports D and E will take up the slack (address lines, I/O strobes, and distinct CS8900A control lines) wherever needed. I hash all this out in Figure 1.

I've been studying the Rabbit documentation in detail since the last time we "spoke," and I found something that I would normally classify as insignificant, but it was interesting enough to follow through. Someone asked the question, "Why did you call your new IC a Rabbit?" The answer is, "Because it is as quick and agile as a real rabbit." Well said. Now that I've proven that with the impromptu engineering change, let's get on with the Ethernet interface.

DEFINITIONS YOU CAN "C"

Consider this. On our little rock in the cosmos, if something has no physical or logical definition, then it simply cannot exist. Well, at least not to us and not to the C programming

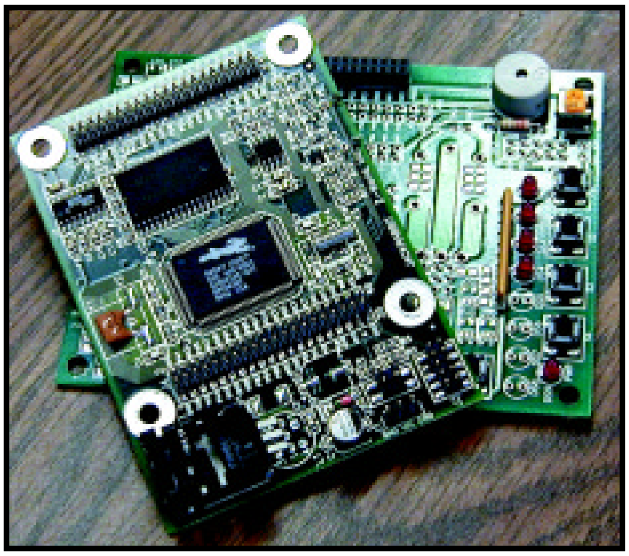


Photo 1—*All of the I/O lines are brought out to headers and the data and address bus lines are dedicated to the Jackrabbit SRAM and flash memory.*

language either. My mere mention of C this early in the game has probably made many of you think that I've caved in to the weight of Dynamic C for this project. Although this entire Ethernet interface could be written in Z80 assembler, with the Rabbit Development Kit, it would be foolhardy to do so. Dynamic C combined with the Rabbit BIOS is the better way to go. There's no reason to reinvent the wheel using assembler. Dynamic C has all of the I/O code resources up front, tested, and ready to go.

I did spend some time attempting to emulate the Dynamic C routines in Z80 assembler and found that trying to beat Dynamic C with assembler was more interesting than productive. But don't count assembler out just yet. I may find a perfect place for it somewhere in this project.

If you're going to talk to other Ethernet-capable devices with the CS8900A-CQ Ethernet Controller IC, you better know who or what is on the other end. Unfortunately, I (and all of you reading this) was born too early. Instead of being a 3-D virtual reality columnist or writing for a magazine that will someday read itself to my great grandchildren, for now I am simply using words on paper. That is, I can't reach out and personally give you the code you see in the listings or provide helpful casual conversational insight as to why I did something one way instead of another.

With that thought, the Ethernet code I present in this offering and those to come will not be short and sweet. So, I won't be able to show it all to you in one issue or a single article. Thus, in the upcoming articles, I'll list only the code that is relevant to the text describing it and refer you to my parallel series of articles that starts this month in *Circuit Cellar Online* for other code as we go along. Also, I am making arrangements to make the printed circuit board and hardware available to those of you who wish to raise these particular types of Ethernet Rabbits at your house.

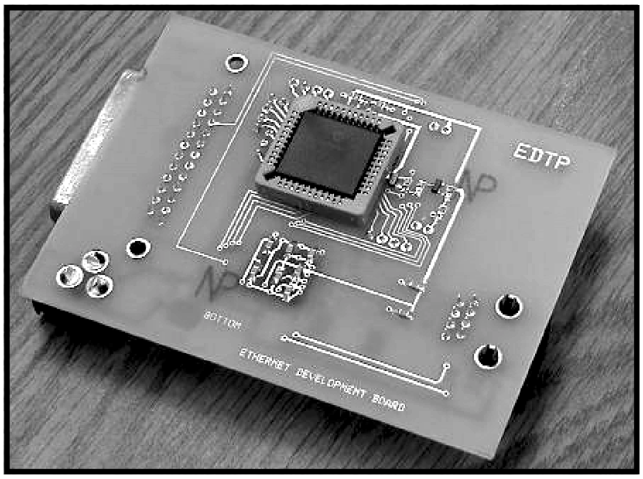


Photo 2—*Everything else is here, including surface-mount pads for single-gate logic parts. But if you want to gain access to the data and address bus, you're going to have to roll your own bunny board.*

As you go further into Alice's Ethernet Wonderland with bunny in hand, you're going to realize that the hardware is not the hard part. The CS8900A-CQ, some common capacitors, resistors, connectors, voltage regulators, crystals, and magnetics do most of the work. As programmers and users of these parts, you must be able to convey a protocol that can be applied to and used by the basic structure of the Ethernet-based electronics you are manipulating. Fortunately, the CS8900A-CQ's internal Ethernet engine and the supporting electronics are aware of all the protocol types you will send down the pike. All you have to do is define and assign the protocol elements so that the Rabbit and CS8900A-CQ are in sync. So, your precise definitions as to protocols ensure that the CS8900A-CQ hardware will respond accordingly and transmit and receive your precious Ethernet packets.

IP 101

All of us have waded through numerous discussions about what IP (Internet Protocol) is and how it works. The sad thing is that we've all been exposed to the "high-level" description of IP. It's like you're not supposed to know why IP does what it does, you're only supposed to know how to make IP do something using an upper-level interface or someone else's code. For instance, unless you're the network administrator, you're not supposed to know your station's IP address or its gateway address. Every instructional text I've seen tells you that the network guys provide that stuff. Sorry, but in this series, I guarantee you will know what every line of Dynamic C is doing and why. When it comes to IP addressing, you'll be able to tell the network administrator what's what. In fact, many of you will probably write to me with suggestions for improvement or send along your own version of "How to IP with Dynamic C." (Not very poetic.)

In the first part of this series, I wrote that most folks can talk the talk but cannot walk the walk when it comes to Ethernet. Well, let's put that to the test. Do pure Ethernet communications care about the IP address? If your answer is "no," turn the page and keep driving or, if you're not in your car, go get a sandwich while I talk to those of you who answered "yes."

As you have ascertained from the previous question and answer session, Ethernet in its raw form does not care about IP addresses. Your next logical question should be, "Then how does Ethernet know where to deliver IP addressed data or any other data it may be carrying?" Well, it's more or less all in the definitions. Check out Listing 1, which shows that I selected an arbitrary IP address for my Rabbit. This particular IP address is really not arbitrary. First of all, it is reserved for local networks and cannot (and should not) be routed over the global Internet you and I have grown to love. The dotted IP address is what you see and use when addressing other IP-based workstations or devices. Actually, the machine and application programming use the dotted IP address. You may tend to equate this "dot.dot.dot.dot" address to something wordier because it is easier for most of us to speak than compute.

Next, check out the 6-octet MAC address to follow. (Mozart's ghost must be in the Florida room today and if so, he's probably here to jam with the Stones' Mick and Keith.) Normally, this MAC address is partially assigned by the IEEE. Each MAC address for every piece of commercial Ethernet physical interface electronics is unique. I used poetic license and assigned my own unique MAC address of "Rabbit." The common commercial MAC address is most often a combined IEEE-supplied ID merged with the Ethernet device's manufacturing serial number. You can normally find this Organizationally Unique Identifier (OUI) as a silkscreen image or a printed tag attached to the Ethernet hardware. According to IEEE, it costs $1250 (U.S.) to register for an OUI. [1] I often use the SMC 9432TX Ethernet adapters, and I can pick out the common SMC OUI code (00 E0 29 XX XX XX) in each of the MAC addresses on differing cards. Intel must have the Rabbit idea because I was involved with tracing an Ethernet

Listing 1—*Don't fall in love with the type definitions within the structures. Using arrays is the best way to describe the lengths of the fields, but as we progress, you may come across a better way to define the packet data.*

```
// Define the rabbit's IP address
// All current IP addresses are 4 octets (bytes)
#define      IP1   192   ;first octet of IP address
#define      IP2   168   ;second octet of IP address
#define      IP3   1      ;third octet of IP address
#define      IP4   200   ;fourth octet of IP address

// Define the rabbit's 48 bit OUI (organizationally unique identi-
fier)
// This is the MAC address or hardware address
#define           MAC1   0x52        // R
#define           MAC2   0x41        // A
#define           MAC3   0x42        // B
#define           MAC4   0x42        // B
#define           MAC5   0x49        // I
#define           MAC6   0x54        // T

// ARP PACKET
typedef struct {
char hardware_type[2];  //hardware type
char protocol_type[2];  //protocol type
char hardware_len;      //hardware address length
char protocol_len;      //protocol address length
char operation[2];      //ARP operation (1=request, 2=reply)
char sender_hwaddr[6];  //senders hardware address
char sender_ipaddr[4];  //senders IP address
char target_hwaddr[6];  //target hardware address
char target_ipaddr[4];  //target IP address
}arp_packet;
```

network one evening and saw the word Intel appear in the workstation MAC field on the network sniffer. The Ethernet card's electronics know what their unique MAC address is and use that and Ethernet protocol to identify and communicate with other Ethernet cards, or here, interfaces. This information is normally kept in a small EEPROM like the Microchip 24LC*xxx* series. I will hard-code the address information.

Right now, I bet you're thinking, "I've never entered any MAC address to establish IP communications. Fred's gone to play with Mozart and the Stones, and forgot his guitar." If you think I'm playing air guitar, here's the scoop: people use names that represent IP addresses to leverage communications between two or more computing devices. Ethernet-based machines use hardware or MAC addresses for the same purpose. The magic that brings this all together, Address Resolution Protocol (ARP), is under the cover of IP communications.

THE RABBIT MUST ARP

If the Rabbit is to speak with any other Ethernet station on the network, its Ethernet interface must be able to recognize and respond to an ARP message. Basically, ARPs come in two flavors, request and response. On Ethernet-based devices that use full TCP/IP stacks and intelligent IP applications, the ARP is transparent. The IP user-written applications don't pass or produce destination MAC addresses. The source and destination machines are usually identified by the application using the dotted addressing scheme. Knowing that the Ethernet physical hardware wants a MAC address, the sending machine first checks its internal ARP cache to see if it can resolve the destination-dotted address to a corresponding destination MAC address. If a match is not found, the sender issues an ARP request to all stations in its LAN segment, asking who belongs to the dotted address in the ARP request packet. In the case of Rabbit Ethernet electronics, you won't be implementing a complete TCP/IP stack mechanism and thus, won't be keeping an ARP cache. For devices to communicate with, your Rabbit Ethernet device will depend on address information from incoming packets.

ARP packets are small and self-contained. From Listing 1, you can see how the ARP packet is laid out. Because the ARP is basically a broadcast, your Rabbit's companion Ethernet IC must pull in the Ethernet header information and process the Ethernet packet to find out if it belongs to the IP address in question. As you might have guessed by now, the ARP request has every field filled in with the exception of the unknown destination MAC address. So, to identify with the ARP request, your Rabbit ARP code must take the incoming Ethernet packet apart field by field, making a decision to respond or trash the incoming ARP request.

When the ARP request frame is read into Rabbit memory, the packet type field in the Ethernet header is verified for 0x0806, which tells you that the data encapsulated within the

Ethernet frame is an ARP packet. Each field in the ARP packet is interrogated and verified to match up with an ARP request. When you're sure that it is a valid ARP request, the target IP address is checked to see if it matches the Rabbit Ethernet Adapter IP address. If a valid request field is verified and the target IP address belongs to your Rabbit and Crystal LAN Ethernet Controller combination, an ARP reply is assembled and transmitted.

The incoming address information, both IP and MAC, is used to route the ARP reply packet back to the sender. This time, the Rabbit's Ethernet MAC address is sent along with the corresponding Rabbit IP address.

BABY'S FIRST HOPS

I'm running low on space for this time around, so I'll come back to ARP and the code to implement it later. For now, let's turn that schematic I referenced earlier into some real Ethernet hardware like the kind in Photo 3. The Ethernet controller hardware is compact and straightforward, so let's start from the left and go right to describe what you will find along the way.

The business end of our little Ethernet Controller is the RJ-45 female receptacle. Although the AMP RJ-45 jack sports eight physical connections, the Ethernet standard only requires that four of them be used. This set of wires comprise the Ethernet transmit and receive pairs which are connected to the RJ-45 pins 1–2 and 3–6, respectively.

Directly to the right of the RJ-45 female connector is what's referred to as the Ethernet magnetics. That's a fancy way of saying Ethernet isolation transformer. No, I didn't use any fancy mathematical transforms to specify the isolation transformer characteristics. This little bugger's behavior is specified in application note AN83, one of the app notes I commented on in Part 1. Note that this particular piece of Ethernet magnetics is designed for 10BaseT connectivity. For those of you who walk the walk (and the rest of you, as well), the "T" implies a twisted pair cable, which is usually Category 5, or CAT5, cable. On the transmit side, the primary to secondary ratio is 1:1.414 and the receive side ration is 1:1. This assembly of the CS8900A-CQ Ethernet Controller sports magnetics from Pulse Engineering part number PE-65745. I also have another Ethernet board that is equipped with the equivalent Halo Electronics part TG42-1406N1.

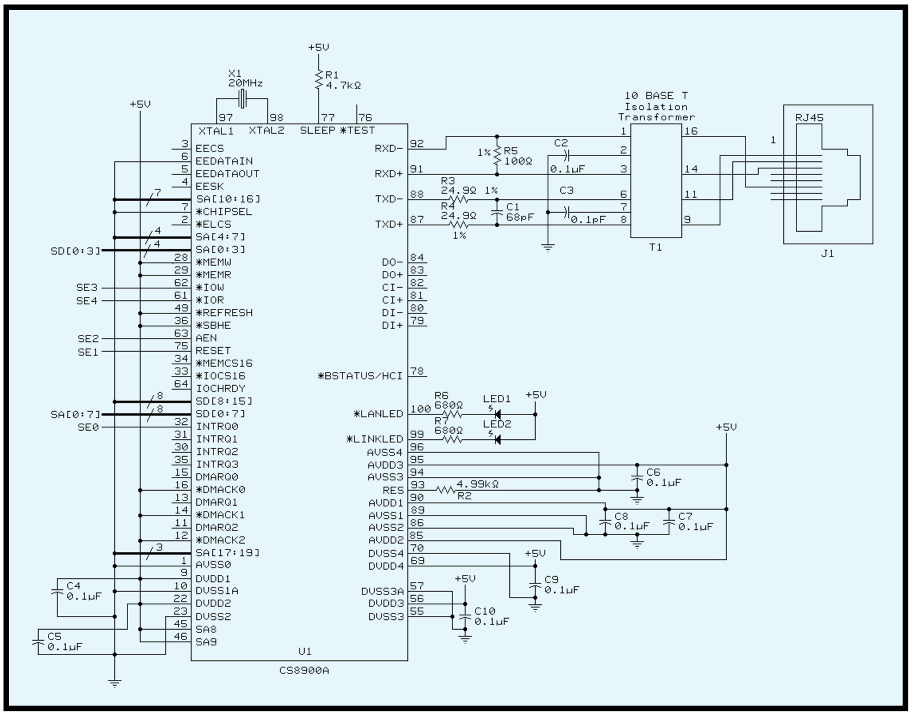


Figure 1—*Although the CS8900A-CQ sports 100 pins, the overall circuit is logical and easy to understand.*

The CAT5 cable I use in the Florida room has an impedance of 100 Ω. Following the six and eight traces from the PE-65745 to the resistor/capacitor network directly to the right of the Ethernet magnetics, you will find a couple of 24.3-Ω precision surface-mount resistors coupled at one end by a 68-pF capacitor and terminated at the CS8900A-CQ transmit pins on the other end.

Why 24.3 Ω? Well, that's what the application note calls for with a 100-Ω cable. If you carefully trace from pins 1 and 2 to the right of the Pulse Engineering transformer, you'll see that a 100-Ω precision resistor is connected across the transformer's pins 1 and 2. This resistor is chosen to match the cable impedance and is placed across the CS8900A-CQ receive pins. The two 24.3-Ω resistors do the same impedance-matching function for the transmit side of the CS8900A-CQ.

There are two other resistors inside this area of resistors and capacitors. One is a standard 4.7-kΩ, 5% part that pulls the active-low CS8900A-CQ SLEEP line high. The other resistor is more critical because it provides bias for the CS8900A-CQ internal analog circuits. This resistor is specified as a 4.99-kΩ, 1% part that is connected as close as possible to the RES pin (pin 93) and the nearest CS8900A-CQ analog ground pin (pin 94).

The CS8900A-CQ power supply bypass caps are found directly behind and under the CS8900A-CQ on the opposite side of the printed circuit board, and the two magnetics caps are directly left of the eight RJ-45 pins. These are all standard 0.1-µF capacitors in surface-mount clothing.

I spoke extensively about the CS8900A-CQ in the first leg of the description of Ethernet and the hardware that comprises it. Directly to the right of the 100-pin CS8900A-CQ, you'll find some familiar faces—surface-mount versions of the standard 7805, a 0.1-µF cap, and a 10-µF tantalum. There are no surprises here, just brute force voltage regulation.

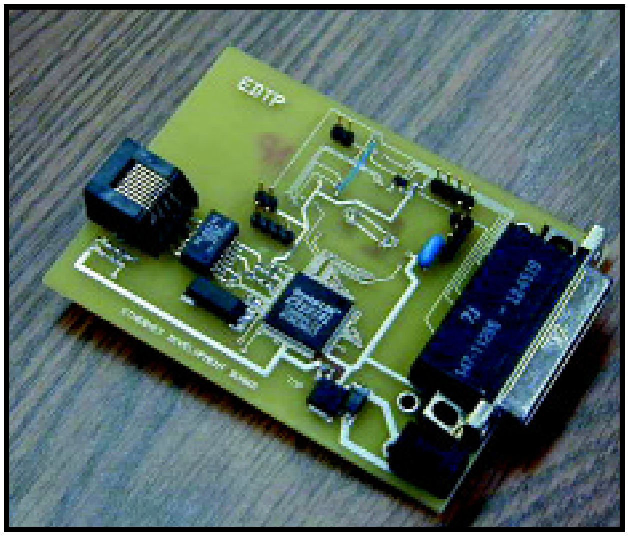


Photo 3—*This is a raw prototype. I'll add some silkscreen and soldermask as I move through the project. And, yes, I put the 100-pin part down manually.*

Directly to the left and just below the CS8900A-CQ is where a 20-MHz crystal resides. This 20-MHz crystal supplies the CS8900A-CQ clock that is used to encode and decode bits in the Ethernet. Additionally, the CS8900A-CQ clock source can be supplied from an external source like an oscillator or the output of a microprocessor clock pin.

Left of the 20-MHz crystal and just below the RJ-45 connector are a couple of LEDs and their current limiting resistors. The LEDs indicate Ethernet link status and data transfer activity.

HOPPING DOWN THE TRAIL

Next time the Applied PC column comes around, I'll put some wire between the Rabbit's monkey board and the Ethernet Controller and move some Ethernet electrons. In addition, I'll explain the software algorithm behind ARP. In the process, I will flesh out some additional Ethernet-oriented Dynamic C.

The ultimate goal is to end up with an Ethernet interface that you can build and use to connect and control devices from embedded microprocessors like the Rabbit.

Let me know if you need to obtain parts and technical information to build your own Ethernet Controller. The folks at Rabbit Semiconductor and Z-World are eager to help too. When we're finished, you'll be able to tell people that Ethernet isn't complicated because you can easily make it 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.

REFERENCE

[1] IEEE, Inc., standards.ieee.org/faqs/OUI.html.

SOURCES

Jackrabbit Development Board
Z-World
(530) 757-8400
Fax: (530) 757-8402
www.zworld.com

Rabbit 2000 microprocessor
Rabbit Semiconductor
(530) 757-8400
Fax: (530) 757-8402
www.rabbitsemiconductor.com

TG42-1406N1
HALO Electronics, Inc.
(650) 568-5800
Fax: (650) 568-6161
www.haloelectronics.com

PE-65745
Pulse Engineering
(215) 355-2900
Fax: (858) 674-8262
www.pulseeng.com

FEATURE ARTICLE

Dan Harrison

Routine Checkup

An Automotive Diagnostic Scan

If your last trip to the mechanic involved opening your checkbook and saying "Aaaagh!", then listen up because Dan designed a scan tool that enables you to troubleshoot today's digitally-enhanced engines in the comfort of your own garage.

Troubleshooting problems in today's car and truck engines is not close to what it was in our father's time. As automotive technology progresses, so too must maintenance and repair methods. In this article, I'll present the design for an onboard diagnostic (OBD) port scan tool, called the OBDScan. OBDScan reads the diagnostic output and clears trouble codes from the vehicle engine control unit for late-model cars and lightweight trucks that support the OBD-II ISO-9141 interface. It consists of an ISO-9141-2-to-RS-232 protocol converter and a Windows 95/98 application (installed in the user's computer).

The advent of digital fuel injection and closed-loop control systems managing fuel, ignition timing, and transmission operation radically changed how automotive troubleshooting and repair is approached. A vast array of sensors and actuators control virtually every aspect of engine operation under the guidance of a computer, the Engine Control Unit (ECU). Troubleshooting engine problems in pre-1996, computer-controlled models often was a nightmare. Each auto manufacturer had its own ECU interface standard and both the electrical interface and data output formats were proprietary and not public.

In 1996, manufacturers were forced to find a solution to the ever-tightening emissions restrictions imposed by the federal government. Today's clean air standards require cars and trucks to have engines that emit low levels of substances defined as pollution. Nitrous oxides, hydrocarbons, and carbon monoxide emissions are closely regulated by the Environmental Protection Agency (EPA). Additionally, federal laws mandate fuel economy standards. In order to monitor the health and status of vehicles' electronic control systems, thereby ensuring proper emission control, the EPA mandated the use of OBD-II in all cars and light-passenger trucks sold in the U.S.

OBD-II gives us a better idea of what is going on inside an engine. OBD-II provides almost complete engine control and monitors parts of the chassis, body and accessory devices, and diagnostic control network.

HISTORY

By the late '70s, many manufacturers (Bosch was the first) used electronics to control a vehicle fuel-injection system and ignition. Electronic engine control became necessary when it was discovered that catalytic converters are the best way to reduce emissions. A catalytic converter operates effi-

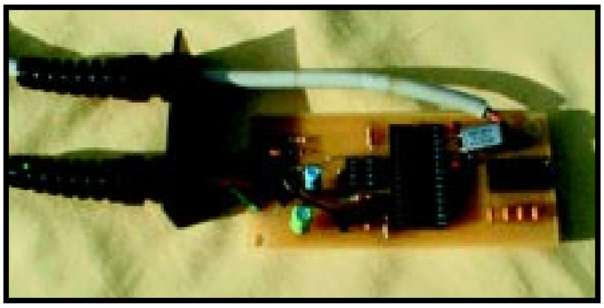


Photo 1—*Before the OBDScan protocol converter board is installed, preliminary voltage tests are run. In addition, a final inspection checks that all components are installed correctly and the solder workmanship is acceptable.*

Photo 2—*Here's the completed and tested OBDScan Protocol Converter installed in a PAC-TEC case. The RS-232 cable is on top and the three-wire cable is the ISO-9141-2 interface.*

ciently only when the engine operates at or near stoichiometry—the balance point when the all-fuel charge is burned and all oxygen is consumed in the burning, when the fuel/air ratio reaches 14.7 to 1.

The venerable carburetor couldn't be adjusted for a precise 14.7:1 fuel/air ratio over the entire operating range, so an alternative had to be found. Electronic systems were developed to precisely control the fuel/air ratio, and as the '80s progressed, embedded microcomputers became the standard all manufacturers chose for engine control. As a result, manufacturers had to develop ways to test the vehicles and diagnose problems generated by the new electronic hardware. The American big three chose to have the ECU generate a serial data output stream, which contained various sensor and software data. Unfortunately for the public, the auto manufacturers considered this information proprietary and used non-standard communications protocols not made public.

As a result of the proliferation of data formats from auto manufacturers and the increasing inability of repair shops to troubleshoot the electronically-controlled engines, the EPA stepped in for the public good. In 1988, the Society of Automotive Engineers (SAE) defined a standard connector plug and set of diagnostic test signals. The EPA adapted most of the recommendations and standards from the SAE onboard diagnostic programs. On February 19, 1993, the EPA promulgated a final rule (58 FR 9468) requiring manufacturers of light-duty vehicles (LDV) and light-duty trucks (LDT) to install onboard emission control diagnostics (OBD) systems beginning in model year 1994.

The regulation required that manufacturers install OBD systems that monitor emission control components for any malfunction or deterioration causing excessive emission thresholds and alert the operator. When a malfunction occurs, diagnostic information must be stored in the vehicle's computer to assist the mechanic in diagnosis and repair. Finally, it called for a standard diagnostic interface to the onboard computer that manufacturers had to meet. But, instead of a standard communication protocol and standard data rates, the EPA allowed three proprietary data formats.

Naturally, these were based on the proprietary systems that GM, Ford, and Chrysler developed. The standards are available (for a steep price) from the SAE and the International Standards Organization (ISO). SAE and ISO have published little data, which means developers need to buy many expensive publications to decipher the federally-mandated interface.

In SAE-J2201, you can find the standard for OBD-II scan tools (see Table 1). J2201 is written at a high level, so to know the OBD-II command/response structure, known as the Application Layer, you need SAE-J1979. This describes data flow and command/response structure well, but the physical and transport layer information is described in ISO-9141 and SAE-J1850, depending on the specific interface. SAE-J1962 describes the standard connector used in all OBD-II–compliant vehicles (see Table 2).

All new cars sold after 1996 are OBD-II compliant. But, how do you know which OBD-II standard your car uses? Generally, GM cars and light trucks use SAE-J1850 VPW (variable pulse width modulation—10.4 Kbps). Chrysler products, European, and most Asian imports use ISO-9141-2 circuitry (10.4 Kbps) and Ford uses SAE-J1850 PWM (pulse width modulation) data signaling (41.6 Kbps). Table 3 is a compilation of the best data to date concerning interfaces. Again, because manufacturers rarely publicize which standards their cars use, there are exceptions to the rules.

If you examine the OBD connector on 1996 and later models, you can determine which format is used with the following information. With J1850 VPW, the connector should have contacts in pins 2, 4, 5, and 16. With ISO-9141-2, the connector should have contacts in pins 4, 5, 7, 15, and 16. And, with J1850 PWM, the connector should have contacts in pins 2, 4, 5, 10, and 16.

When the ECU detects serious problems, it turns on the Malfunction Indicator Light (MIL), which remains on until repair and the MIL is reset. Intermittent failures cause the MIL to light momentarily but turn off before the problem is located. The ECU also stores data taken when the fault condition is detected. This is called freeze-frame data and may be useful when troubleshooting difficult or intermittent problems. Most commercial scan tools can read and display freeze-frame data.

The J-1979 OBD-II scan tool performs various functions related to monitoring the emissions-related functions. It's possible to get a real-time look at various engine parameters, such as manifold pressure, calculated engine load, and ignition timing. The scan tool also can read,

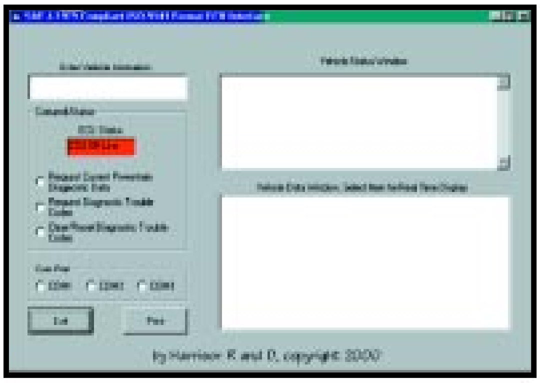


Photo 3—*When the OBDScan executable is run, this screen is displayed. First, you must select a COM port to initiate communication with the protocol converter.*

display, and clear trouble codes. When your MIL turns on but no obvious problems are noticed during a casual inspection, you at least know what the ECU thinks is the problem, even if you have to take it to a repair shop. If it is something you can fix, then you can also clear the trouble code at home. Dealers charge $70 to $100 to scan and clear trouble codes.

UNDERSTANDING ISO-9141

The scan tool design presented in this article uses the ISO-9141-2 physical layer interface, which works for the 1997 Honda Accord and 1996 Chrysler Sebring. The components to the left of the 68HC705P6 in Figure 1 are a TTL-to-ISO-9141 interface. This is a simple communication protocol. The interface can only listen or talk at any one time, because all data is communicated bidirectionally over the K line. This is unlike the typical RS-232, which usually can perform both tasks simultaneously.

In the case of data transmit, the TXD line is level-shifted by Q1 to a 0- to 12-V signal on the K line and is routed back to the RXD line by comparator U3. When receiving data, the ECU toggles the K line between 12 V and ground, which is level-shifted to 0 to 5 V by the comparator. This is a simple, straightforward design. The 5-bps ISO-9141-2 initialization word is shown in Figure 2.

UNDERSTANDING SAE-J1979

The document SAE-J1979 defines the diagnostic test modes and request and response messages necessary to support the OBD regulations. SAE-J1979 describes the application layer of the OBD requirements and is applicable to both ISO-9141 and J-1850 interfaces. There are nine different

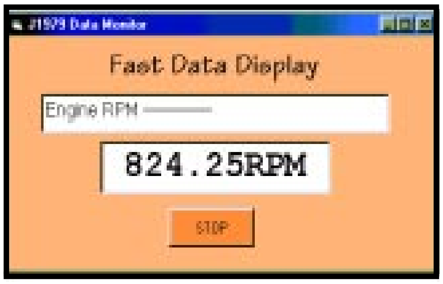


Photo 4—*The OBDScan Fast Data Display window is started by double clicking a data parameter. OBDScan software then samples the parameter as fast as it can (8 to 10 Hz), providing a real-time indication similar to a digital meter.*

modes that are defined by J1979. Table 4 shows the basic J1979 message format for the 10.4-Kbps messages for both ISO-9141-2 and J8150.

This covers most Chrysler products and Japanese and European imports. GM uses J1850 for most of its products. The other format for the 41.6-Kbps data used by Ford Motor Company is not described here.

Mode 01 gives access to emissions-related engine data. This includes data from sensors and calculated data used by the ECU for emissions management. Mode 01 provides engine status and sensor data output.

The SAE mandates that all OBD-II vehicles support a PID, which returns information about which diagnostic parameters are supported. All scan tools use this PID (00) to set up a list of supported data. Table 5 shows the command and response formats for Mode 01. This mode allows various PID values to be requested from the ECU. Table 6 lists the PIDs available.

It is important to emphasize that all vehicles do not support all PIDs. That is why PID 00 gathers information about which PIDs are supported by each vehicle. Obviously, all OBD-II-compliant vehicles must support PID 00.

Table 1—*Most of the documents required to understand and develop OBD-II scan tools are stated here. Buying all these documents individually can cost a small fortune.*

SAE-J1850	Class B data communication network
SAE-J1930	Diagnostic terms, definitions, abbreviations, and acronyms
SAE-J1962	Diagnostic connector
SAE-J1978	OBDScan tool
SAE-J1979	E/E diagnostic test modes
SAE-J2012	Recommended format and messages for trouble codes
SAE-J2178	Class B data communication network messages
SAE-J2186	E/E data link security
SAE-J2190	Enhanced E/E diagnostic test modes
SAE-J2201	Universal interface of OBD-II Scan
ISO-9141-2	CARB requirements for interchange of information

Mode 02 allows access to emissions-related data that was stored as a result of an ECU-detected malfunction. When the ECU detects conditions that could result in degraded emissions control, it may freeze the current data set for diagnostic tests.

Mode 03 enables the scan tool to read any stored power train trouble codes from the ECU. The OBDScan first sends a Mode 01, PID 01 request to get the number of stored trouble codes. Then, it sends a Mode 03 request for the trouble codes.

Each response message contains three trouble codes. If only one trouble code is present, the last two trouble codes will be 00. This keeps all messages the same length for simplicity. If more than three are present, a second or third response message may be required to get all codes.

Each trouble code is transmitted as a two-byte number. The first two bits (7–6) of the first byte will be zeros to indicate a power train code (other codes exist in SAE-J2012). The next two bits (5–4) indicate the first digit of the trouble code, 0–3. The next four bits (3–0) and the second byte indicate the last three digits of the trouble code in Binary Coded Decimal (BCD) format. Be aware that if no trouble codes exist but are requested, the ECU is not required to answer.

Mode 04 resets and clears all trouble code values from the ECU. It is specified to work when the ignition is on and the engine is not running. Some vehicles may respond with the engine running and others will ignore it.

Mode 05 allows access to the onboard oxygen sensor monitoring tests, as required by federal regulation. The data output can be constructed into a graphical representation of the oxygen sensor performance over time to determine if it's working properly.

Mode 06 allows access to the results for onboard diagnostic monitoring tests of components or systems that are not continuously monitored.

Terminal	Assigned function
1	Discretionary
2	Bus + of J1850
3	Discretionary
4	Chassis ground
5	Signal ground
6	CAN high of SAE-J2284
7	K line ISO-9141
8	Discretionary
9	Discretionary
10	Bus of J1850
11	Discretionary
12	Discretionary
13	Discretionary
14	CAN low of SAE-J2284
15	L line ISO-9141
16	Battery positive

Table 2—*The pin identification for OBD-II-compliant vehicles is defined by SAE-J1962. All vehicles sold in the U.S. since 1996 are equipped with this connector.*

Auto	Model year	OBD-II type
Acura	1996 and newer	ISO
Audi	1996 and newer	ISO
BMW	1996 and newer	ISO
Chrysler	1996–1998	ISO
Chrysler	1999 and newer	ISO
Dodge Truck	1996 and newer	ISO
Geo	1996 and newer	ISO
GM International	1996 and newer	ISO
Honda	1996 and newer	ISO
Hyundai	1996 and newer	ISO
Infiniti	1996 and newer	ISO
Isuzu	1997 and newer	ISO
Jaguar	1996 and newer	ISO
Jeep	1996 and newer	ISO
Kia	1996 and newer	ISO
Land Rover	1996 and newer	ISO
Lexus	1997 and newer	ISO
Mazda	1996 and newer	ISO
Mazda made by Ford not supported		
Mercedes	1996 and newer	ISO
Mitsubishi	1996 and newer	ISO
Nissan	1996 and newer	ISO
Plymouth	1996 and newer	ISO
Porsche	1996 and newer	ISO
Saab	1996 and newer	ISO
Saab VPW is not supported		
Subaru	1996 and newer	ISO
Toyota	1997 and newer	ISO
Volkswagen	1996 and newer	ISO
Volvo	1996 and newer	ISO

Table 3—*These vehicles are ISO-9141-2 according to my best information. It's a rule of thumb that foreign cars (European and Asian) and Chrysler products (Jeep, Chrysler, Dodge, and Plymouth) are ISO-9141-2. However, there are a few exceptions.*

Mode 07 allows the tool to obtain test results for power train components that are continuously monitored during normal driving conditions.

Mode 08 enables the tool to control the operation of an on-board system, test, or component.

Mode 09 enables the tool to read vehicle information such as the VIN and calibration IDs. OBDScan supports Modes 01, 03, and 04, the most useful for troubleshooting. You can visit the Harrison R & D web site for the latest software.

THE PROTOCOL CONVERTER HARDWARE INTERFACE

Following the philosophy of KISS, "Keep it simple, stupid," the hardware design is as simple as I could make it. This keeps the cost of the kit to a minimum, and it makes for a more reliable design.

The OBDScan electrical design has three functional blocks, the ISO-9141-2 interface, 6805 microcontroller, and RS-232 interface (see Figure 1). The ISO-9141-2 interface is comprised of U3, Q1, R1, R2, R3, R6, and R11. The ISO-9141 K line is a bidirectional signal with a 0- to 12-V signal amplitude. Because it is bidirectional, only one bus device can talk at a time, or message collisions will occur. The OBD-II spec makes the scan tool the bus master, or initiator, for OBD-II communication. The ECU can only output data on the K line when commanded or requested to do so by the scan tool.

Header bytes	Data bytes	Error detect
68 6A F1	Maximum 7 data bytes	Yes

Table 4—*Here's the SAE-J1979 format for a diagnostic request from a scan tool. The header field is fixed, the data field varies according to the request, and the error detection byte is a checksum for ISO-9141-2.*

After a request for data, the scan tool must wait for the response before sending another command or request. When the OBDScan transmits on the K line, '6805 sets or clears Port A, which is coupled to the base of Q1 through R6. Q1 pulls the K line low, through the 75-Ω resistor R3, when Port A, bit 0 is set high. R7 pulls the line high when Q1 is turned off by clearing Port A, bit 0.

OBDScan receives data through the LM393 comparator U3. A voltage divider, R1, and R2, set the inverting input to 0.5 × VBatt, which is the setpoint for deciding what is a one or zero. Voltage levels greater than 0.5 × VBatt are logic 1 and voltages less than 0.5 × VBatt are logic 0. Software inside U2 is used to read and write data to the ISO-9141 interface using bit banging. In this case, the CPU uses precisely-timed routines to send and receive serial data for either toggling an output line for transmit or reading an input line for receive.

ISO-9141-2 specifies only a 1% tolerance for timing, so the software must be carefully tuned or the ECU will not recognize the data. The processor interface to ISO-9141 is so similar to interfacing an RS-232 UART that a standard UART could be used as the TTL-level interface if the data rate can be set to 10.4 Kbps with eight data bits, no parity, and one stop bit.

The microcontroller (U2) is a Motorola MC68HC705P6 used as the RS-232–to–ISO-9141 Protocol Converter and provides for framing, message headers, checksum, and timing control from the host computer into J1979/ISO-9141-2 format. U2 reads the diagnostic response from the ECU and returns it as RS-232 data to the host computer, initializing the ISO-9141 interface upon sensing a connection. It maintains the connection as long as the unit is on. The '6805 maintains a connection by issuing a request for diagnostic data from the ECU every 2 s. Without this request, the ECU will stop communications until an initialization is performed.

Note that not all ISO-9141-2 vehicles follow the standard completely. Some do not require the initialization sequence prior to communication, but all respond if initialized.

There are 4672 bytes of one-time-programmable EPROM in the '6805 for program storage. About 30% are

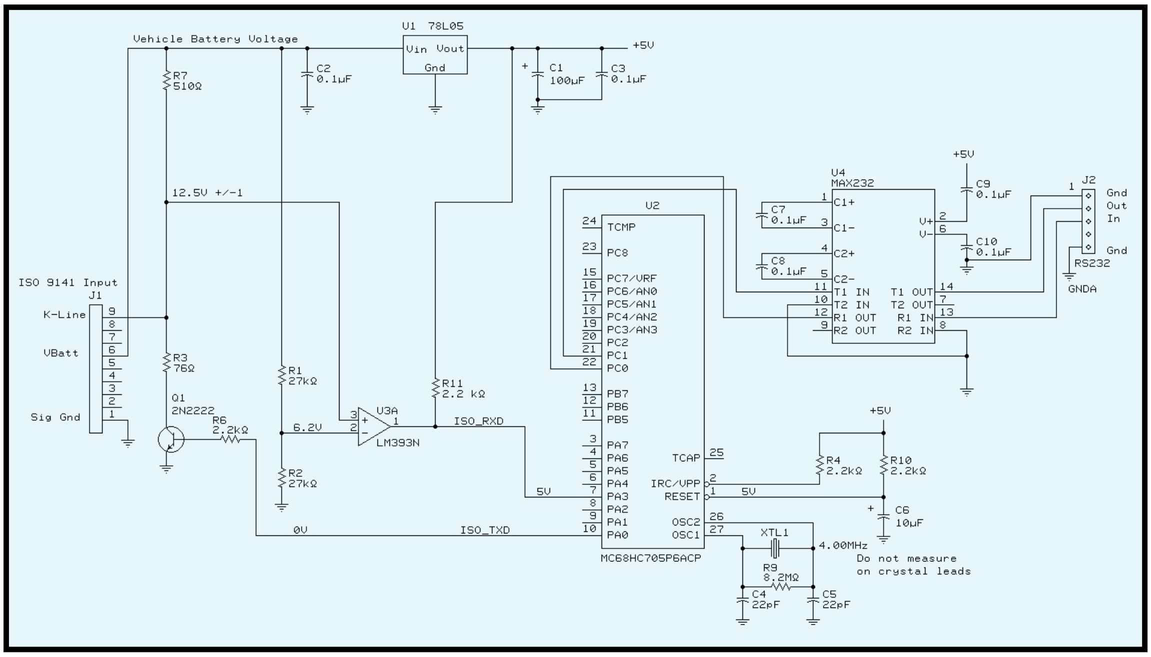


Figure 1—*The OBDScan voltages are for a 12-V nominal lead acid battery. Voltages are approximate and depend on resistor tolerance and actual battery voltage in the vehicle. Layout is not critical, but keep the crystal close to the processor.*

used in the current implementation, leaving plenty of room for future expansion. For those of you building this project from scratch, the object code is available from *Circuit Cellar's* web site or from www.ghg.net/dharrison/OBDScan.html.

The RS-232 design is simple, using a Maxim MAX232 chip for converting the TTL-level logic signals to RS-232 level. The typical driver output voltage swing is ±8 V when loaded with a nominal 5-kΩ RS-232 receiver. Output swing is guaranteed to meet EIA/TIA-232E and V.28 specifications, which call for ±5-V minimum driver output levels under worst-case conditions. Input thresholds are set at 0.8 and 2.4 V, hence receivers respond to TTL-level inputs, as well as EIA/TIA-232E and V.28 levels.

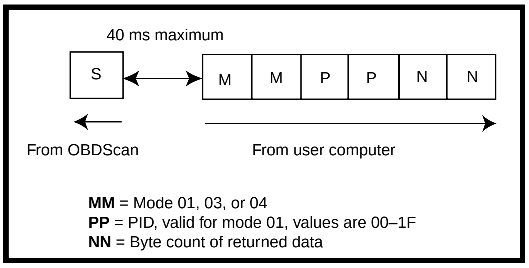


Figure 3—*The protocol converter sends the sync character every 50 ms to signal the start of a 40-ms window in which it can receive a command from the host PC. The host PC sends a fixed format string of six characters.*

The '6805 CPU also uses the bit-banging technique for RS-232 communication. The vehicle battery powers the OBDScan through the diagnostic connector. This can vary from 12 to 15 V depending on whether or not the engine is running. The input power is converted to 5 V for the electronics by U1—a three-terminal voltage regulator; almost any generic regulator will work, OBDScan uses an LM2931 or 78L05. Obviously the firmware has a lot to do, and in this case, the '6805 processor is almost fully used with respect to CPU cycles. All '6805 code was written in assembly language because of timing and memory space considerations.

To get an overall understanding of firmware operation, I'll start with a description of the RS-232 interface protocol to the OBDScan application in the host computer. When power is applied, OBDScan transmits a "P." In approximately 4 s, if the vehicle ECU is on, OBDScan establishes a connection with the ECU and begins transmitting "S" every 64 ms. "S" synchronizes your computer with the OBDScan. It represents an opportunity for you to send a request for diagnostic data.

After receipt of the sync character, your computer has 40 ms to send a request. If OBDScan does not receive a request, it times out for the remainder of the 64 ms and restarts the process. Every 48 sync characters, OBDScan sends a request to the ECU for diagnostic data to keep the link alive, whether or not the user has sent a request. All user diagnostic requests consist of six ASCII bytes.

For example, if you want to send a command to retrieve throttle position, send an ASCII string of 010B06 within 40 ms of receiving the sync character (see Table 6). The OBDScan will respond with the full J1979 response data of V486BAA01TTCC. The *V* indicates ISO9141, *486B* is specified by J1979, *AA* is the ECU address, *01* is the mode of the request, *TT* is an 8-bit binary value for the throttle position (0–100% open, with 255 = 100%), and *CC* is a message checksum. Currently, the supported commands are 01, 03, and 04. SAE-J1979 defines additional commands for reading the VIN, freeze-frame data, oxygen sensor test, and monitoring the results of built-in self-tests in the ECU.

ISO-9141 INITIALIZATION

Upon powerup, the OBDScan protocol converter firmware waits 3 s for the ECU to get ready and then initiates the ISO-9141 initialization sequence. If the ECU does not respond, the OBDScan protocol converter waits another 3 s and tries again. This process repeats until a connection is established. As soon as a reliable, error-free connection exists, the protocol converter sets up an infinite loop in which it waits for user input on the RS-232 port and constantly sends "S."

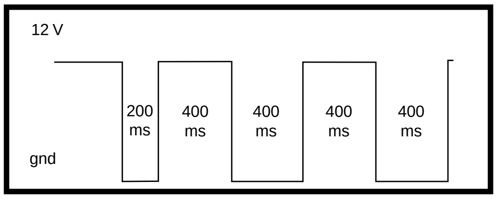


Figure 2—*ISO-9141-2 specifies that an initialization word, transmitted from the scan tool at 5 bps, must be used to establish communications. If the bus is inactive for more than 4 s, the scan tool must initialize the bus again to continue communications. Although not all manufacturers require this, all scan tools must support it.*

COMMAND PROCESSING

When an RS-232 input is received, it is parsed into mode, PID, and byte count. An ISO-9141 transaction is prepared with the proper header and checksum and then transmitted to the ECU. The protocol converter waits up to 1 s for a response from the ECU. If the response happens before the timeout, it reads the number of bytes specified in byte count and sends it back out the RS-232. If the protocol converter times out waiting for ECU data, it initializes repeatedly until the link is reestablished.

Note how ISO-9141 and SAE- J1979 compliance work. The protocol-converter firmware handles internal and external communication timing between message bytes and messages in accordance with the applicable standards. It also calculates all required headers, checksums, address and keyword initializations, performs watchdog timeout on bus messaging, and error detection. The timing requirements are listed in Table 7.

As you see, the built-in time delays between messages are considerable, making any sampled data faster than 10 Hz difficult in ISO-9141.

BUILDING THE PROTOCOL CONVERTER

Verify that your parts match the parts list. The resistor color code for each resistor is shown in the parts list, and all capacitors (crystals and ICs) are marked. The protocol converter is a single-sided board, with all

Scan tool to vehicle	Mode, PID
Vehicle to scan tool	41 h, PID, data

Table 5—*The command and response format for SAE-J1979 Mode 01 is listed here.*

components mounted on top, as shown in Figure 3. Close spacing of foil runs in some places, so be careful soldering to avoid solder bridges. Use the minimum amount of solder and a 25- to 35-W iron with a small tip. Remember to keep the tip clean for good solder connections.

First, install the resistors from the top so the leads are protruding from the foil side of the board. Bend the leads so the resistors stay in place when the board is flipped over for soldering. Then, solder all resistors in place and clip off the leads close to the board. It is a good idea to wear safety glasses when you are cutting component leads to avoid eye injury from flying parts.

Next, install IC sockets and solder them in place. Install the crystal, transistor, Q1, voltage regulator U1, and capacitors. Observe the polarity on the electrolytics. Then, install the two-wire jumpers as indicated on the silkscreen. Before U2 and U3 are installed into the sockets, apply 12 V to the board by connecting the 12-V ground to C1 negative lead and the +12 V to Jumper 1. After that, measure the voltage on U2, pin 28 with respect to 12-V ground; it should be 5.0 V ±0.25 V. If the voltage is not OK, check if U1 was installed incorrectly or if there are solder bridges.

After problems are fixed, install U2, U3, and U4. Be sure to get pin 1 oriented properly or the chips will be destroyed when power is applied. Install the RS-232 and OBD-II cables as shown in Photo 1. You must insert both cables through the strain reliefs before soldering them into the PCB.

Now it's time to drill the 0.6″ holes in the front panel. Insert the combined cable/strain relief into the front panel holes (see Photo 2). The cable wires are small and fragile, so handle them with care until the board is installed in the case. Next, connect the 12-V supply as instructed and connect the RS-232 cable to a computer. Using your computer, run HyperTerm from the Run menu and configure for the COM port to which the protocol converter is connected. You should see "P" transmit every few seconds as the protocol converter attempts to initialize an ISO-9141 interface. If you have a scope, measure the signal on the collector of Q1. It should look like the signal in Figure 2 repeated every 4 s.

Drill the 0.125″ holes in the bottom shell of the case and press in the PCB holders, and install the PCB on the holders. Push the PCB down until it snaps into place. Route the cables around the parts on the PCB and insert the front panel into the bottom shell. Then use the supplied hardware to fasten the top and bottom shell together. At this point, you're ready to proceed to testing.

PID	Description
00	PIDs supported
01	Number of trouble codes and tests available
03	Fuel system status
04	Calculated load
05	Coolant temperature
06	Short-term fuel trim Bank 1
07	Long-term fuel trim Bank 1
08	Short-term fuel trim Bank 2
09	Long-term fuel trim Bank 2
0A	Fuel pressure
0B	Manifold pressure
0C	Revolutions per minute (rpm)
0D	Vehicle speed
0E	Ignition timing advance
0F	Intake air temperature
10	Air flow
11	Throttle position
12	Secondary air status
13	Location of 02 sensor
14	02 Sensor 1 Bank 1 V
15	02 Sensor 2 Bank 1 V
16	02 Sensor 3 Bank 1 V
17	02 Sensor 4 Bank 1 V
18	02 Sensor 1 Bank 1 V
19	02 Sensor 2 Bank 1 V
1A	02 Sensor 3 Bank 1 V
1B	02 Sensor 4 Bank 1 V
1C	OBD type
1D	Location of 02 sensor
1E	PTO status
20	SAE-2190 support

Table 6—*Here are the PID codes as defined in SAE-J9179. Not all vehicles support all parameters. All OBD-II vehicles respond to PID 00, which gives information about which PIDs are supported.*

w0	Bus idle time before initialization	2000 ms min.
t0	ISO initialization	2440 ms
p2	ISO inter-message time	50 ms
p4	Inter-byte time	5 ms
t6	SAE delay before login	300 ms

Table 7—*Take a look at the ISO-9141-2-specified timing delays. Notice the inter-byte time of 5 ms, and the inter-message time of 50 ms. These two timing parameters keep the data rate at about 10 updates per second when in real-time display mode.*

GETTING STARTED

The OBDScan application is Windows 95/98-compliant and written in Microsoft Visual Basic 6.0. The executable program is available to download from the Harrison R & D web site for free. To install OBDScan, follow the directions in the file. The program will install the executable file and all required DLLs.

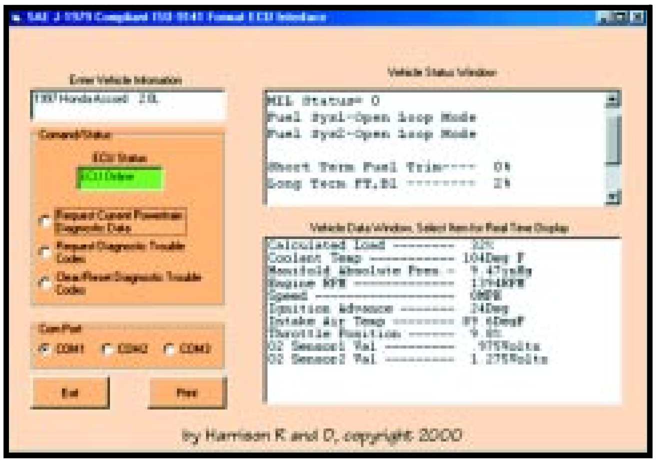
Before connecting the protocol converter, launch OBDScan by either double clicking the icon or selecting it from the Programs menu. You should see the screen shown in Photo 3.

Select the COM port that will be used for communication. If the selected port is not available or is being used by another program, you will receive an error message. The Windows operating system allows one program at a time to control any individual COM port, so be sure to quit other applications. After you successfully select a COM port, the ECU status box will change from red to yellow with the message "ECU Init," meaning that it's waiting for the protocol converter to initialize the ECU.

After completing this phase, you're ready to move on to communication with a vehicle. Check if your vehicle is listed in Table 3. The first step is to locate the OBD-II connector, which is required by law to be within 1 meter of the steering wheel. It's usually located under the dash, but if not, check behind ashtrays and in console compartments. Be sure your computer is within cable reach of the OBD-II connector. You can use up to 50″ of RS-232 extender if required. With the ignition off, plug in the protocol converter's OBD-II connector to the mate in the vehicle.

Start the engine, and the ECU status indicator will become green when the ECU has been initialized. Next, select Request Diagnostic Trouble Codes. You should see a display in the vehicle status window like "MIL is

Photo 5—*Here is the actual screen display from a 1997 Honda Accord.*


SAE J-1979 Compliant ISO 9141 Format ECU Interface
Enter Vehicle Information
Command/Status
ECU Status
ECU Online
Request Current Powertrain Diagnostic Data
Request Diagnostic Trouble Codes
Clear/Reset Diagnostic Trouble Codes
Com Port
COM1
COM2
COM3
Exit
Print
Vehicle Status Window
Vehicle Data Window, Select Item for Real Time Display
by Harrison R and D, copyright 2000

```
1997 Honda Accord 2.0L
```

```
MIL Status= 0
Fuel Sys1-Open Loop Mode
Fuel Sys2-Open Loop Mode

Short Term Fuel Trim----  0%
Long Term FT,B1 --------  2%
```

```
Calculated Load ---------  32%
Coolant Temp ------------ 104Deg F
Manifold Absolute Pres -  9.47inHg
Engine RPM -------------  1394RPM
Speed ------------------  0MPH
Ignition Advance -------  24Deg
Intake Air Temp -------- 89.6DegF
Throttle Position ------  9.8%
O2 Sensor1 Val ---------  .975Volts
O2 Sensor2 Val ---------  1.275Volts
```

OFF." There are no trouble codes set if your MIL light is on or have trouble codes set. Definitions for the SAE trouble codes are in the appendix of the software manual.

As defined by SAE-2012, diagnostic trouble codes (DTC) consist of a three-digit numeric code preceded by an alphanumeric designator. If the alphanumeric designator is P0, then the trouble code is SAE defined. However, if it's P1, the DTC is defined by the manufacturer and you will need a shop manual. For example, if the trouble code P0150 is displayed, it is SAE controlled and indicates a problem in the O2 sensor. If the DTC is P1298, you would have to consult the shop manual for an explanation.

To clear any trouble codes and turn off the MIL, select Clear/Reset Diagnostic Trouble Codes. Then select Request Current Power Train Diagnostic Data. You will see data appear in both the vehicle status and data windows. This data is mostly optional and varies from vehicle to vehicle. Most manufacturers provide a reasonable selection of parameters, but a few 1996 models provide only two or three parameters (probably because it was the first year).

Any data appearing in the vehicle data window can be selected for real-time display. For example, to see throttle position, click on the line in the vehicle data window (see Photo 4). This window will continuously display throttle position at 8 to 10 Hz until you press the Stop button in the fast data display window. The fast data display provides a troubleshooting benefit with large characters for easy viewing and rapid updating for fast-changing data. You can print data from the main screen.

Photo 5 shows a representative set of data read from my vehicle. Users of OBDScan can expect this basic set from all ISO-9141-2-equipped vehicles. If Read Trouble Codes was selected, the status field would contain any trouble codes set by the vehicle computer.

FINAL ANALYSIS

The OBDScan is designed to be both a useful repair tool and an aid to learn about and explore the ODB-II Diagnostic Interface. Demystifying the OBD-II port and putting owners back in control of their property is important. After all, you paid a hefty price for that late model car or truck, so don't let the automaker virtually weld the hood. ■

Dan Harrison holds a B.S. in Physics and Mathematics and is working towards a Masters degree. He has worked at the Johnson Space Center in Texas for 22 years and is currently branch chief of the Electronic Design and Development branch of the Engineering Directorate. His designs have flown aboard space shuttles and the International Space Station. His interest in electronic engine control started several years ago while restoring a Nissan 280ZX. You can reach him at dharrison@ghg.net.

SOURCE

OBDScan
Harrison R & D
(281) 485-7105
www.ghg.net/dharrison

FEATURE ARTICLE

Tom Napier

One Small Step

Part 2: Liftoff!

After describing the model rocket and its capabilities in Part 1, Tom gets down to the details. He notes that despite frustration along the way, the project was satisfying because it confirmed years of speculations about rockets. So, Tom's ready for liftoff.

In Part 1, I described how a 1-ounce instrumentation package can fly on a model rocket and record its roll rate and acceleration. I also presented a schematic of the electronics package. Now, it's time to explain the details.

THE ELECTRONICS

Referring to Figure 1 in Part 1, diode D1 in series with the 6-V battery limits the supply voltage to approximately 5.5 V. It also protects the components in case the battery is inserted backwards. The 47-µF capacitor C1 retains power in case the battery briefly breaks contact. Only the PIC runs directly from the supply rail, the other chips are powered via transistor Q1, which is turned on during flight by a PIC port pin.

A low-power, single-supply, dual op-amp (U2) conditions the sensor outputs. One half amplifies and offsets the signal from the accelerometer U1 to give a working range of –1 to 12 G. The other half buffers and amplifies the signal from the LED. I used an OP290, but an OP295 gives a greater output swing.

The 10-kΩ resistor (R9), between the amplifier output and ground, was an afterthought. The output was clamping at 0.5 V, which puzzled me because it is specified to swing down to 50 µV above ground. What the datasheet does not state is that the output cannot sink the 7 µA of feedback current without help from an external resistor. (The other half of the amplifier does not need a load because its feedback resistors are connected to ground.)

The accelerometer has a bandwidth of 1 kHz, but the amplifier's bandwidth is limited to 150 Hz by filter capacitor C5. At 500 samples per second, you are sampling at just over the Nyquist rate.

The 16C71 has a programmable option that supplies pull-up current to the Port B input pins. Shorting plugs pull these pins low until they are pulled off to signal an event. The pull-up current also drives the serial input pin high when there is no input connected. Plugging in the computer pulls the input low, generating an interrupt and starting the control program.

INTERFACING

The old RS-232 standard calls for signal levels between 6 and 12 V and –6 and –12 V. Because RS-232 inputs are supposed to work with 3-V signals, many computers compromise by using 5- and –5-V levels.

My portable computer is a 1983 vintage Tandy Model 100. It tries to output –5 V but fails if the output is loaded by the standard 3000-Ω RS-232 input. Therefore, I designed the input circuit to generate a low level from anything less than about –2 V. A greater positive signal turns off the grounded-base transistor Q3 and allows the PIC's internal pullup of about 0.25 mA to pull the input pin high. Diode D3 offsets the –0.6 V that appear at the collector of the transistor when the input is low. Diode D4 protects the transistor's base-emitter junction from breakdown in case a 12-V input is connected.

The serial output cheats. It assumes that the computer pulls its input low when an open circuit is presented to it. Thus, the logger circuit only has to pull up the output to 5 V when it wants to send a zero bit. Of course this beats putting a negative supply on the rocket!

CALIBRATION AND CONTROL

Plugging in the computer puts the system in command mode, letting you record calibration information in the EEPROM. Acceleration calibrations of –1, 0, and 1 G are achieved by pointing the rocket nose down and typing G0, laying it on its side and typing G1, then pointing it nose up and typing G2. The calibration values are used during data processing to correct the accelerometer readings for drift.

Similarly, the roll sensor can be pointed away from (S0) and towards (S1) the sun to record results. Typing X transmits data frames continuously as hex bytes until you turn off the computer. Because the EEPROM can be written to at least 100,000 times, there is no harm in making a complete record and reading it prior to flight. Gently shaking the rocket generates surprisingly high G levels.

Any block of data in the EEPROM can be displayed by typing B followed by a two-digit hex number from 00 to 7F. For example, B00 displays the calibration information. The data is transferred to RAM registers 10H to 1FH and can be modified by the R command followed by the register number in hex. This displays the present contents of the register.

Typing a new hex number overwrites the register and displays the next one. To write the RAM registers to the EEPROM, type W and the block number in hex. Typing a space moves to the next register with no change. The enter key exits from the editor.

In addition to calibration information, the first EEPROM block stores the desired recording rate and a byte that selects this rate or the default rate of 100 samples per second. The sampling rate is set by typing T and a digit from 0 to 4, where 0 = 500 sps, 1 = 200 sps, 2 = 100 sps (the default setting), 3 = 50 sps, and 4 = 20 sps.

Typing D0 activates the default setting, typing D1 activates the user setting. Using a blank EEPROM automatically sets the default sampling rate. The upper eight bytes of block 0 can be prerecorded with the date, serial number, and engine to be used.

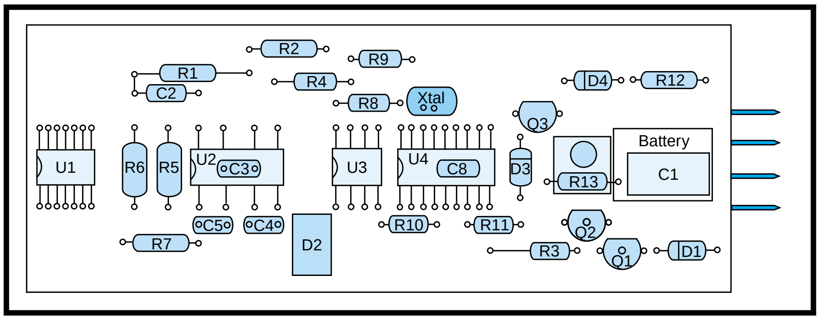


Figure 1—*The rocket parts are mounted on the copper side of the 0.9″-wide PC board. Q1, C1, and R13 are mounted on the underside. The two 22-pF capacitors (not shown) connected between the crystal and ground are also under the board. A four-way ribbon cable glued to the spine board connects the event-input pins on the PIC to the 8-pin connector at the foot of the board.*

CONSTRUCTION

Payloads for model rockets have to be small, lightweight, and rugged enough to withstand fast accelerations. I chose a body tube just less than 1″ in diameter, which forced the PC board to be about 0.9″ wide and several inches long. I cut a 0.9″ × 6″ board from Radio Shack part 276-170. It was cut off-center to leave three-hole copper strips on the left and four-hole strips on the right. I cut a scrap piece 0.3″ wide and epoxied it on edge to the plain side of the board where it acts as a combined reinforcing spine and ground bus. The battery holder, when glued to the copper side of the board, is the correct height to locate the board inside the tube.

When designing your payload structure, try to distribute the mass evenly around the spin axis. An offset center of gravity can cause precession or tumbling in flight. My full-size rockets sometimes carry ballast weights bolted inside the skin, but this isn't practical in a model.

As shown in Figure 1, most of the components, including the socket for the 16C71, are mounted on the copper side of the board. Because there was enough space, I used normal 0.25-W resistors. The accelerometer is a 14-pin surface-mount device. Only five of its pins are used, so I glued it to the board with silicone rubber and connected the active pins with short bits of wire-wrap wire. After everything has been tested, more silicone rubber will stop these wires and the chip pins from moving.

One event pin is hard-wired to indicate payload separation. The other three sensor connections can be wired to sense other events. A stereo jack socket is glued to the board and wired to the serial interface.

I installed the EEPROM in a socket, giving me the option of taking a tube of EEPROMs to the launch site rather than a portable computer. It takes only moments to swap EEPROMs between flights, which is an advantage when you have an audience of impatient children who are more interested in seeing rockets fly than in the finer points of data transfer.

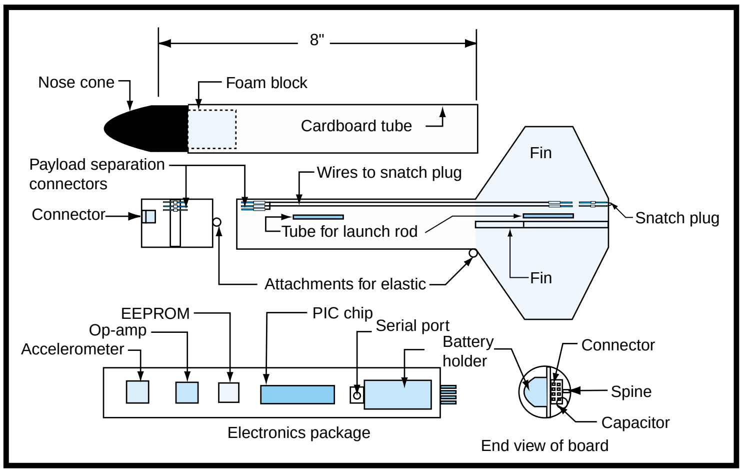


Figure 2—*The instrumentation package is mounted in a rocket about 18″ long and 1″ in diameter. The snatch plug wires trigger data recording at liftoff.*

INSTALLATION

Figure 2 shows the installation in a typical rocket. The lower end of the circuit board fits into a balsa plug about 2″ long (you can file this down from a 1″ square block). The slot is deep enough that the battery holder rests on the plug. The other end of the balsa plug fits into the lower part of the rocket where it is supported on a reinforcing ring glued inside the body tube. A 0.25″ ring of body tube fits around the middle of the balsa plug and forms the attachment point for the lower connectors.

In flight, the circuit board is protected by an 8″ tube, which also fits on the balsa plug. The top end of this tube has a nose cone glued into it. A plug of polystyrene foam about 1″ long is glued to the base of the nose cone. This holds the board in place and protects it in the event of a ballistic landing. The tube has one hole in its side to allow the serial jack to be plugged in and another hole to allow the roll sensor to see the sun. Masking tape ensures that the tube stays on during flight.

I embedded a section cut from a ribbon cable connector into the balsa plug. It mates with an eight-pin connector glued to the circuit board. This provides the start signal and event sensing. Three sockets glued to the surface of the plug mate with pins glued to the lower tube to provide the start-up and payload separation signals. Wires from two of the sockets run down the side of the rocket to a point near the launch lug, which guides the rocket up the launch rod. Two sockets glued here connect to the launch detect snatch plug.

All connectors designed to separate during flight are made from individual pins and sockets from D-type connectors. The sides of the sockets are pried apart to reduce the separation force. My snatch plug is two pins linked by a loop of bus wire, which also links it to the base of the launch rod.

This plug must be mounted where it can't foul any part of the rocket at liftoff. It should have little slack because the rocket moves less than 1″ in the first tenth of a second after ignition. A tenth of a second later, it reaches peak acceleration, is travelling at 25′ per second, and is about to leave the launch rod. You need a quick startup to record this part of the flight.

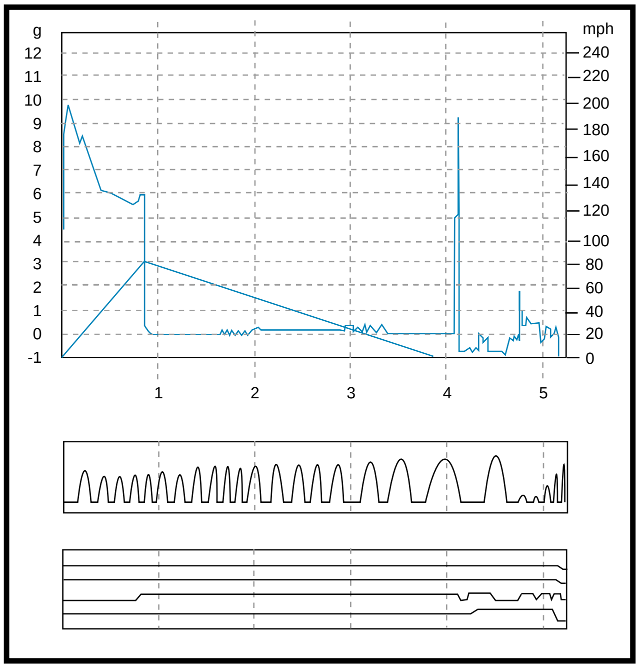


Figure 3—*These graphs measure a flight using a B6-4 engine. The upper graph plots acceleration and velocity. The spike at 4 s is the payload separation. The middle graph shows the output from the sun sensor. The bottom graph presents the four event recorder traces. The upper two were unused, the next is the mercury switch output, and bottom line shows the payload separation.*

The instrumentation package is self-contained in its tube. It separates from the lower tube during the descent but remains attached by an elastic cord. In this configuration, the drag is sufficient to allow a safe landing on grass. The package can be mounted on different bodies, allowing experiments with various fins, engines, and tube sizes. You could even tape it to your car door to check your brakes.

FIRMWARE

The firmware has two jobs. One responsibility is to accept preflight instructions and to execute the post-flight download. The second job is to record the flight data. When it's not doing one of these jobs, the PIC is in sleep mode and consuming negligible power. This means that both liftoff and connecting the serial interface must wake the PIC. Connecting the launch snatch plug brings the reset pin low. Liftoff removes the reset, starting the recording program.

Connecting the serial port applies a low level to the serial input, which is also the interrupt pin. This wakes the PIC and diverts operation to the monitor program, allowing the recorded data to be read and commands to be entered.

In default mode, the PIC samples the accelerometer, roll detector, and event sensors 100 times per second. The four event sensors form bits 12–15 of a data frame. Bits 8–11 are the analog roll sensor voltage, and bits 0–7 are the accelerometer voltage. Eight frames are stored in RAM and then transmitted to the EEPROM as a block.

Block 0 contains five calibration frames and three control frames. The calibration frames store the settings of the accelerometer in its three positions and the "towards sun" and "away from sun" voltages generated by the roll sensor. You may load these before flight. This information is not used by the controller, but is important for postflight processing.

The first byte of the control frame indicates which should be used, the default or the user rate. It must be programmed to zero to specify that the second byte contains the user sampling rate. Blank EEPROMs usually contain mostly ones. These bytes are read at powerup to set the recording rate. The third byte is reserved to set the recording format.

Downloading transmits a text file with 40 characters per block to suit the 40-character display of my Model 100. Each frame is transmitted as four hex characters. The first seven frames are followed by a space, the last one by a line feed.

One irritating feature of the Model 100 is that it displays all incoming data. It loses characters if you don't put a 0.5-s time delay between lines.

DID IT WORK?

Amazingly yes, although the first test flights revealed software bugs and a construction problem. I had reversed a bit test, thus recorded at the user rate, which was set at 200 sps, rather than the 100-sps default rate.

Another problem was that at the end of the flight, the software turned off the EEPROM before it had time to write the last data block. The roll sensor output was too small. I mounted the LED looking straight out, because I overlooked the fact that usually the sun is above the horizon. However, tilting the LED and cutting a longer slot in the body tube ameliorated the problem.

The LED-viewing angle could still be a problem when the wind comes from the same direction as the sun. But, rockets are launched into the wind to return the payload near the launch site. This method tilts the nose further towards the light source, shading the LED.

Figure 3 shows a flight with a small engine, a B6-4. The curve in the upper graph, which peaks then levels off, is the acceleration. The curve that slants up from the lower left shows velocity in miles per hour. The peak G level is greater than 9 G, and the rocket is moving at 80 mph at motor burnout. A C6 engine boosts speed to greater than 160 mph.

The pronounced spike that occurs just after the 4-s mark is the engine's expulsion charge firing. The G record then becomes chaotic because the payload swings and bounces below the body tube. Note that there is no indication where the rocket turned over at apogee (the point when the velocity goes to zero).

I discovered that the engine maker's specification of the delay before the expulsion charge fires (the last digit of the engine's type number) must be measured from ignition, rather than from burnout. For example, a B6-2 engine separated the payload only 1.4 s after takeoff while the rocket was still travelling rapidly upwards.

The middle graph of Figure 3 shows the roll sensor output. The rocket reaches six turns per second then slows down as speed decreases.

I only used two event channels during my test flights. One showed the payload separation and the other recorded the position of the mercury switch, which, as expected, only detected the engine burnout and some shaking on the way down.

I wrote a short program on my Amiga 3000 to generate Figure 3. Windows users will have to write their own display software.

THE BOTTOM LINE

This project was both frustrating and gratifying. My early hardware and software designs were correct in the outline, but it took about three weeks to get them from 99% to 100% complete. I was not familiar with the 16C71, and getting everything to work took some digging in the databook. In addition, the original hardware needed to be modified extensively before it flew.

In the end, everything worked well and I confirmed much about model rocket behavior that I had pondered for a decade. ■

Tom Napier is a physicist and engineer who parlayed his design experience into an electronics consulting business. His compulsion to share his knowledge drives him to write magazine articles, and he regrets that he cannot offer free design assistance to individual readers. He will start using e-mail once the bugs have been worked out.

SOURCES

Microcontroller and EEPROM
Microchip Technology Inc.
(888) 628-6247
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

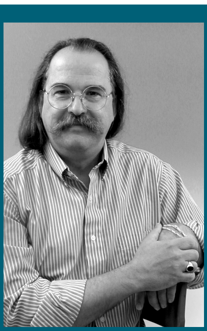
Accelerometer and amplifier
Analog Devices Inc.
(617) 329-4700
Fax: (617) 329-1241
www.analog.com

Rocket parts and engines
Estes Industries Inc.
www.estesrockets.com

FROM THE BENCH

Jeff Bachiochi

They Just Haven't Caught on Yet



Jeff's report from this year's MASTER conference shows that Microchip is successfully pulling off a one-of-a-kind performance when it comes to educating engineers.

This is the fourth MASTER (Microchip's Annual Summer Technical Exchange Review) conference, and as far as I know, the other manufacturers still haven't caught on yet. If you've watched the microcontroller market over the last decade (in particular, Microchip's growth) you may have wondered, "How'd they do that?" You don't get to be a major player without doing a few things right. There are certainly plenty of factors that determine a company's success. I'm not writing this to patronize Microchip. In fact, I won't mention the company again. I do, however, want to talk about a philosophy.

As the name implies, the MASTER conference is an exchange of information. What the name doesn't imply, however, is just how intensive this exchange is. The conference includes 20 one-hour classes on a wide variety of subjects, including hardware and software (some in-depth classes require two- to four-hour blocks). Everyone can learn something here, whether you have a soldering iron or a keyboard attached as a permanent appendage.

As if this information overload wasn't enough, evening activities are designed to promote networking among consultants, customers, third-party support, FAEs, and design houses. Evening activities are a bit more relaxing while, as David Letterman says, "enjoying a fine beverage of your choice."

GUIDANCE COUNSELOR

When you look over the total of 32 possible classes on the registration form (90 hours of class time), you quickly realize that you will only touch the tip of the iceberg. And, because there are new courses added each year, after four years I can still find areas in need of improvement. Narrowing the choices down to what will fit into three days is the most difficult task. Let's look in on a few classes during this year's technical exchange.

HANDS-ON RFID

This class is basically about passive RFID tags. Although active tags were mentioned, because they are for longer range, are expensive, and require batteries, they weren't the center of attention. Passive tags are generally used for distances of less than a meter and are already produced by companies such as Temic, EM, Philips, SCS, TI, Bistar, and Gemplus. As their name suggests, passive tags do not require a power source. Well, that's not entirely correct.

The circuitry does require power. However, this power comes from the RFID reader's carrier signal. The passive tag is tuned to this carrier and sucks in power to run its circuitry. This circuitry does not transmit back to the reader, instead it shorts out its tuned circuitry, removing itself as a load to the reader. The reader is designed to detect a change in carrier load. This change looks like amplitude modulation. The passive tag's circuitry can use a number of encoding and modulation methods to pass data to the RFID reader.

RFID carrier frequency is another hot topic. The FCC limits the output power used in these ranges—125 kHz, 13.56 MHz, and 2.45 GHz are three ranges used where the tag characteristics greatly differ. At 125 kHz, the tags have good penetration characteristics but are high in cost and have low data rates. At 2.45 GHz, the tags have no penetration characteristics but are low in cost and have high data

Photo 1—*A crowd gathers to watch a robot attempting to traverse a roadway consisting of a black line on a white floor.*

rates. On the other hand, the 13.56-MHz tags are a compromise yielding good penetration, while maintaining a low cost and high data rate.

Anti-collision is an issue in which multiple passive tags may enter the detection envelope. For many applications this envelope may only be centimeters, so anti-collision is not necessary because there isn't room for multiple objects to be within range at the same time. For larger envelopes, the RFID readers must have a way to tell passive tags not to speak simultaneously. Various methods of controlling this are handled by collapsing the carrier field. The passive tags can sense this and respond (or delay response) depending on their programming. Factors of optimizing the detection envelope include carrier power, tuned antenna circuit Q, antenna diameter, SNR, and data rate.

Playing with hardware is worth a thousand words. The time I set aside for hands-on work using some development equipment really helped visualize how different encoding schemes are used to make unique tagging systems based on the application requirements.

CONNECTING TO THE INTERNET

Everyone wants to be a "dotcom." It's like the gold rush of the 1800s all over again. The Internet is certainly here to stay. And because it is, more and more companies want their piece of the action. In order to integrate your product for communication with the Internet, you need to know about the protocols used. In an attempt to simplify the network connection, it is broken down into layers—applications, transport, network, and link.

The application, or top layer, is your data in any number of formats. You may wish to use HTTP, FTP, or e-mail via the TCP transport layer, TFTP or SNMP via the UDP transport layer, or may require a ping or traceroute via the ICMP. The transport layer breaks the data into packets and wraps its particular format around the data.

The third (network) layer again wraps the transport layer's packet with additional information, including source and destination IP addresses. This is the packet-routing information.

The bottom, or link layer, consists of the actual protocols and device drivers for physical connections like Ethernet packets over fiber, 10BaseT, or wireless connections, Serial Line Internet Protocol (SLIP) over modem connections. The network layer's packet is encapsulated with the protocol's required fields.

Visit http://rfc.akc.com for the Request For Comment documents.

ANALOG NOISE

Everyone who has designed using both analog and digital components must be concerned with keeping the digital noise from showing up in their analog signals. This is a real problem when using higher resolution A/Ds. Noise that would not cause problems in the digital realm can cause early loss of hair when it shows up as wandering A/D conversions. You expect an A/D to be stable to ±1 bit. Where can you look when it's obvious that the system noise is to blame? (The savvy designer should've used good design practices.)

To fix problems, first review the chosen components. This will allow an analysis of the minimum theoretical noise you can expect. It is a good idea to go through these calculations to determine if you can achieve your design goals with the circuit design you have in mind.

Now look at your layout and separate the digital and analog circuitry. Proper layout techniques solve many issues. Segregating the analog circuitry far away from the digital reduces radiated noise. This is caused by PCB traces that look like antennas to fast-edged digital signals or by high current signals that induce a magnetic field which, in turn, creates currents in other traces. In addition, make sure that both the power and ground connections to each area have their own path back to the power supply and that all chips are properly bypassed to keep them from borrowing current from their neighbors. Use ground and power planes when possible. You might need to filter the power to the analog circuitry if you're using a switching power supply.

If signal conditioning is implemented in front of the A/D inputs, there's probably signal gain. Noise picked up is multiplied by the gain along with the signal, so keep high impedance connections as short as possible. The sampling speed should not break Nyquist's rule. It is prudent to have a low-pass (anti-aliasing) filter at the A/D's input with its knee positioned to reject harmonics.

When using your scope to probe for noise, your scope ground must be kept short and connected to a ground as close to your signal as possible. A 6″ ground clip will pick up a great deal of noise that probably isn't even on the signal you are probing.

USB

The Universal Serial Bus (USB) is destined to eliminate the serial and parallel port from future PC motherboards. To empower this move, some USB designers have already begun manufacturing USB serial and parallel port dongles. After all, would you spring for a new PC if you couldn't use your old peripherals with it?

Advantages of the USB include live connect and disconnect, automatic driver installation, and 100 mA of minimum power available through the bus. Present USB specifications (rev1.1) allow 1.5 MBps at low speed and 12 MBps at full speed. However, to compete with firewire (1394 A), USB rev2.0 will allow 480 MBps. These are bit times, and actual data throughput will vary with packet size.

The physical layer consists of a power and ground pair and a differential signaling pair. USB is not a daisy-chain system. It is based on a star topography with the host or a hub upstream controlling communication and a hub or device downstream responding to commands.

USB is not a simple protocol. In order for the host PC to know what kind of device is being connected to the USB pyramid, a standardized transfer must take place whenever the host detects that a new device has been attached. This initial transfer, or enumeration, takes place at a known address and lets the new device tell the host about itself. The host can reassign the device to a new unused address and install the correct driver, allowing access to the device functions. Although there are device drivers for standard peripherals, if your device has any non-standard functions, you will need to provide a Windows/Apple/Linux driver for your device. In addition, you may have to provide application software.

USB transfers occur in 1-ms frames. Each frame is packed with control, interrupt, isochronous, and/or bulk transfers. The USB low-speed protocol is used for peripherals that have small amounts of data to pass (i.e., mouse or keyboard) and can only use control or interrupt transfers. The full-speed protocol is used when a peripheral needs to send timely or bulk data, however, all four transfer formats are allowed.

SIGNAL CONDITIONING

For most digital engineers, connecting some sensors to an A/D input is a challenge. Many sensors output signals in the millivolt range, and A/D inputs have full-scale ranges up to V_{CC}. To maximize the A/D converter resolution, use scale and gain to translate (or condition) the sensor's full-scale output to the A/D's full-scale input. Operational amplifiers are the building blocks used in designing signal conditioners. The ideal op-amp would accept any input voltage and not require input current. The outputs would swing to any voltage and supply infinite current. The op-amp would have infinite gain for all signals from DC to infinity without producing offset voltage errors. It could be powered from any voltage source while requiring no current and reject the noise created in the power supply.

But the perfect op-amp doesn't exist. Although the designers try to implement the best possible characteristics, many op-amps are designed to improve some parameters by sacrificing others. The circuit designer must, therefore, know what parameters are important to the application.

The op-amp's power supply voltage will directly affect the maximum and minimum input and output voltages. Although newer op-amps are touting rail-to-rail swings, be sure to look at their specifications carefully, because no op-amp will operate with true rail-to-rail swings and many vary widely based on load. Today it is becoming increasingly popular to use single-supply op-amps. However, this brings with it a wrath of new problems for signals that are bi-polar.

Problems can arise when an op-amp attenuates an AC signal because of bandwidth limitations. There is a tradeoff between stage gain and bandwidth, so the designer must not be fooled by the GBWP (Gain Bandwidth Product). This represents the point where an input frequency will have no gain (gain = 1) through the op-amp, even though it is configured for infinite gain (open loop). Because the op-amp's gain increases 20 dB per octave below this frequency, the designer must make sure the necessary gain falls within a range that can be achieved at the maximum frequency of the application.

Slew rate has another output limitation. An op-amp's slew rate is how fast its output can change in V/µs. This can be easily confused for a high gain signal, which is bandwidth limited and clipped at the rails.

This class includes a hands-on portion where the basic op-amp circuits (noninverting and inverting) and instrumentation are used to create signal conditioners with specific transfer functions. A software design package, similar to FilterLab and AmpLab, allows the designer to select configurations and component values to solve a number of application issues. Different op-amp models can be substituted into the circuit where the circuit parameters are automatically recalculated, allowing the designer to judge the level of suitability. A Spice file can also be created from the design to allow dynamic simulation of the circuit. However, even a Spice simulation isn't foolproof, and prototyping should always follow the design and simulation steps.

OTHER TOPICS

Interfacing was the hot topic at this year's MASTER conference. Other topics included anti-aliasing filter design, designing with digital potentiometers, CAN bus, and using I^{2}C in master and slave mode. I must admit that the sponsoring manufacturer did have classes available on the products they offer, as well as classes on third-party programmer's tools. I won't suggest that this manufacturer didn't wave its own flag, however, most information was unbiased.

Last year, an after-hours robot building competition was introduced. The Lego Mindstorm Robotics Lab was used by teams of volunteers to create a ball-collecting robot. As a result of the limited number of after-hours available and the learning curve needed to use the Mindstorm, few teams were actually successful. This year, a preassembled robotics platform was used. The majority of effort now became one of software. Consequently, more robots qualified on the rigorous line tracking raceway (see Photo 1). Interestingly, a small transmitting video camera was placed on some of the robots. A projection TV gave the teams a robot's eye view of the raceway. Technology sure is fun.

Although August in Arizona doesn't sound great, I hope this article convinces you to go back to school for a few days. And, if I can convince other manufacturers to follow suit and provide intensive programs on a variety of subjects, we can all get a quick burst of education to help supplement the confines of our workplaces.

THE LAST WORD

The conference was also the place to meet *Circuit Cellar* readers. I got to meet and talk with guys and gals from all over this small planet—from the USA, Australia, Brazil, France, Mexico, and more. I want all of you to know how much I appreciate hearing your kind words. It is comforting to see that we all have something in common. ◼

Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on Circuit Cellar*'s engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

REFERENCES

D. Anderson, *Universal Serial Bus System Architecture*, Addison Wesley, ISBN 0-201-46137-4, 1997.

W.M. Tan, *Developing USB PC Peripherals*, Annabooks, ISBN 0-929392-64-7, 1999.

RESOURCES

125 kHz
ISO 11784/11785—Animal tagging

13.57 MHz
ISO 14443, ISO 15693—Proximity and vicinity
Microchip AN707—MCRF 355/360 Applications
Microchip AN710—Antenna circuit design
Microchip—RFID System Design Guide

Connecting to the Internet
Microchip AN724—PPP dialup
Microchip AN731—iReady design
Microchip AN732—Flash memory reprogramming
RFC1127—Server protocol
RFC894 and 1042—Ethernet packets
RFC1055—SLIP logon
RFC1661—PPP negotiation
RFC1662—Packet checksum calculation
RFC791—P Header
RFC792—ICMP Structure
RFC768—UDP Header
RFC793—TCP Header
RFC2616—HTTP Application

Analog Noise
Microchip AN688—Layout tips for 12-bit A/D converter applications
Microchip AN699—Anti-aliasing, analog filters for data acquisition systems

USB
www.microsoft.com/hwdev/busbios/usbpnp.htm
www.microchip.com/usb
www.usb.org
www.microsoft.com/hwdev/usb
www.apple.com/usb

SOURCES

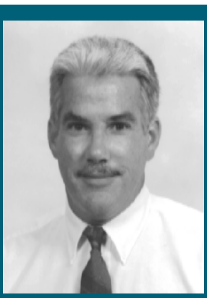
MASTER Program
Microchip Technology Inc.
(408) 786-7200
(408) 786-7302
www.microchip.com

USB
USB Implements Forum, Inc.
www.usb.org

SILICON UPDATE

Tom Cantrell

Who's Nexus?



Although Tom's no Perry Mason, it didn't take him very long to uncover the who, what, when, where, and why behind Nexus. No matter what you call it, implementing a global standard is no easy task.

As I went through the datasheets and press releases for last month's article about Motorola's latest M-Core moves, "Dial M-Core for Micros?," I repeatedly came across references to Nexus.

Thanks to the web, extensive detective work wasn't required to find out who (or what) Nexus is. For more information, just head over to their web site and read all about it. As I navigated my way around, I realized that Nexus is really GEPDIS. Well, now that explains it.

I can certainly sympathize with the marketeers who made the switch. No doubt Nexus makes for smoother sounding press releases, but GEPDIS, as in Global Embedded Processor Debug Interface Standard, says it all.

Is it possible? Anytime you're talking global standards, it's a big deal. Not to mention that there's more embedded processor debug interfaces than you can shake a stick at.

Combining the two clauses might seem optimistic, bordering on oxymoronic. Nevertheless, considering the list of heavyweights involved (see Table 1), it's wise to look closer.

WHAT'S DEBUGGING ME

It may sound audacious, but the goal of a standard debug interface is a worthy one. In fact, I think it's more than worthy. I consider it mandatory, lest our digital dream machines become our worst debug nightmare. With certainty I say, it's a debug crisis!

Remember the first computers? They were vacuum-tube Goliaths that could barely compute their way out of a paper bag, and that's if they managed to continue running for more than a few minutes at a time. Talk about a hardware crisis.

As we all know, silicon and recent hardware synthesis technology have come to the rescue. Now the hardware crisis boils down to choosing between Verilog, VHDL, and shopping chip suppliers to find the lowest price.

Then, after computers came into being that could handle more than a few lines of code, there was a software crisis (a term I first heard used by Intel's Andy Grove in the '80s). So many megahertz and megabytes, so little time to write software.

But the advent of high-level languages, automatic code generating tools, and third-party OEM software has put a big dent in the software crisis. Consider our desktop PCs where higher-ups seem to have no trouble filling more memory and consuming more MIPS with bloatware. These days, a single programmer with a budget for advanced tools and OEM software can point, click, and purchase megabytes of code (i.e., RTOS, TCP/IP stack, and such) all before lunchtime.

So, you've got your fancy chips and your fancy code, now what? If it didn't seem to work right the first time, welcome to the debug crisis.

GOOD OLD DAYS...

I say this with a trace of nostalgia. Back then, debugging a balky system wasn't a cakewalk, but it was possible. In fact, much of it was straightforward, at least for the simple microprocessors of the time. In particular, external address and data buses offered a way to easily observe and control. With chips like the Z80 and 68K, it was possible to bring a system to life with little more than a few pages of test code and a scope.

With the emergence of single-chip MCUs (i.e., no external bus), the market for emulators took off.

Emulators traditionally rely on a special bond-out version of the MCU that, as the name implies, connects the internal buses to pins, and then by a cable to the target socket.

Although still viable for older commodity micros, traditional emulation technology won't cut it for today's advanced ICs. With everything buried on-chip, external activity on the pins yields little insight about the inside. As the number of architectures and variants grows, manufacturers don't want to be bothered with juggling different custom bond-out chips. Triple-digit clock rates, mixed signal I/O, and tiny surface-mount packages spell doom for the traditional pod/cable/probe lash-up.

Now, consider the emergence of ASICs and SOCs that may incorporate one or more processing elements (e.g., a CPU and DSP). Don't bother calling Emulators-R-Us.

The bottom line is, if you plan to integrate everything but the kitchen sink on one chip, the only course of action is to integrate debug logic, too.

JTAG, YOU'RE IT

Integrating debug logic is exactly what micro suppliers have started doing during the past few years. The idea is that a little extra debug logic embedded on-chip can go a long way towards solving the debug crisis.

Not surprisingly, Motorola, the number-one MCU supplier, was one of the first to take the plunge with the Background Debug Mode (BDM) and On-Chip Emulation (OnCE) approach.

In general, these and similar schemes from other manufacturers exploit the fact that a powerful host (as today's PCs surely are) can do the heavy lifting, thereby offloading and minimizing the debug logic that is required on-chip.

With debug smarts on both the host and chip, the main issue remains about how to connect the two. Often, the debug logic hitches a ride on one of the controller's serial ports. However, this approach has a couple of limitations. The limited speed of the link may create a bottleneck and the port's use for debug will conflict with its possible use in the application.

JTAG (a.k.a., IEEE 1149.1) to the rescue. Originally, the five-wire clock serial interface was designed merely for hardware debugging and testing. With each chip in a system connected in a JTAG daisy-chain, test equipment can perform what's known as a boundary scan, essentially setting or checking the level of each pin.

It's not difficult to imagine how boundary scans and JTAG are helpful for debugging PCB prototypes or automating production tests. But it wasn't long before folks had the bright idea that JTAG could go beyond its pin-centric roots to serve as the generic link between the host and the target during system integration and debug, as well.

Compared to hijacking a UART or such, JTAG has key advantages. It's undoubtedly fast, capable of running at many megahertz. In one real-world example, engineers at Motorola measured end-to-end throughput of 1 MBps on a 40-MHz M-Core part (admittedly with the addition of an *RDY pin to help speed things up). [1] That's about 1000 times faster than the 9600-bps UART that folks made do with back when everything managed to fit in 64 KB.

Another advantage (as mandated in the standard) is that JTAG pins must be dedicated to testing and debugging, thereby reining in the chip designer's inevitable tendency to multiplex them with some application I/O function. So, whenever a chip includes JTAG, you're assured that during development (and even after the application is deployed) there will be a dedicated debug connection available.

Finally, JTAG is quickly becoming a standard feature. After surveying the market, Motorola asserts that the majority of leading MCU vendors are getting on the JTAG bandwagon. Widespread adoption is crucial to boost the number of available seats, which, in turn, attracts the interest and monetary investment of third-party tool developers.

Applied Dynamics International
Applied Microsystems Corp.
Digital Logic Instruments
Green Hills Software, Inc.
Hewlett-Packard Co.
Hitachi Semiconductor Inc.
Hitex Development Tools
HIWARE
Infineon Technologies
Metrowerks Corp.
Mitsubishi Electric Corp.
Motorola Inc.
Nohau Corp.
Noral Micrologics
PLX Technology, Inc.
STMicroelectronics
Tektronix, Inc.
Yokogawa Digital Computer Corp.

Table 1—*With more than a few top-tier chip and tool suppliers onboard, the Nexus Consortium goes beyond a mere marketing marriage of convenience.*

	Static development features				
Development feature	Class I	Class II	Class III	Class IV	Nexus feature
Read/write user registers in debug mode	V	V	V	V	
Read/write user memory in debug mode	A	A	A	A	Read/write access
Enter a debug mode from reset	A	A	A	A	Development and control status
Enter a debug mode from user mode	A	A	A	A	Development and control status
Exit a debug mode to user mode	A	A	A	A	Development and control status
Single-step instruction in user mode and re-enter debug mode	A	A	A	A	
Stop program execution on instruction/ data breakpoint and enter debug mode (minimum of two breakpoints)	A	A	A	A	Breakpoints/watchpoints

A = required feature implemented via API *P = required feature implemented via development port or port pin*
V = required vendor-defined feature implemented via API *O = optional feature*

Table 2—*Nexus defines four classes of capability across a range of prices and performance, from simple static control to real-time dynamic debug. The former includes basic operations such as accessing memory and stopping, starting, and single-stepping the processor.*

CLASS ACT

The Nexus folks are sensitive to the pragmatic issues of getting there from here. It's tempting to push for an all-things-to-all-designers standard that checks off every item on a wish list. It would also be a mistake to bite off more than the market can chew.

Thus, the standard incorporates flexibility and scalability by classifying four different levels of debug capability (see Tables 2 and 3). They cover the whole spectrum, from Class I, which is close to what's found on today's commodity MCUs, to Class

	Dynamic development features				
Development feature	Class I	Class II	Class III	Class IV	Nexus Feature
Ability to set breakpoint or watchpoint	A	A	A	A	Breakpoints/watchpoints
Device identification	A	A and P	A and P	A and P	Device ID message (see section 6)
Ability to send out an event occurrence when watchpoint matches	P	P	P	P	Watchpoint message (see section 6)
Monitor process ownership while processor runs in real-time		P	P	P	Ownership trace
Monitor program flow while processor runs in real-time (logical address)		P	P	P	Program trace
Monitor data writes while processor runs in real-time			P	P	Data trace (writes only)
Read/write memory locations while program runs in real-time			A and P	A and P	Read/write access
Program execution (instruction/data) from Nexus port for reset or exceptions				P	Memory substitution
Ability to start ownership, program, or data trace upon watchpoint occurrence				A	Development control and status
Ability to start memory substitution upon watchpoint occurrence or upon program access of device-specific address				O	Development control and status
Monitor data reads while processor runs in real-time			O	O	Data trace (reads and writes)
Low-speed I/O port replacement and high-speed I/O port sharing		O	O	O	Port replacement/sharing
Transmit data values for acquisition by tool			Opt.	Opt.	Data acquisition

Table 3—*Dynamic debug features include familiar breakpoints and watchpoints as well as advanced capabilities, such as memory substitution and I/O replacement.*

IV, which can handle anything thrown at it by the chips of the future.

As the entry point to Nexus, Class I provides a baseline of so-called static debug features that includes single-step reading and writing target registers and memory and instruction and data breakpoints. The static designation means that the processor must be stopped (in debug mode) during tool access. In other words, Class I debug is intrusive. In return, it consumes the least silicon, meeting the primary goal of minimizing barriers to entry and overcoming inertia for chip, tool, and application designers.

To that end, Class I does not need a lot of pins. And in fact, it demands little beyond a stock JTAG port for tool connection.

Upping the ante with dynamic debug features, Class II adds the watchpoints, process ownership, and program trace that are required for monitoring instruction execution in real time. Class III adds similar capabilities for data access and tracing. Finally, Class IV offers generic memory substitution (i.e., overlay memory in emulator-speak) and port replacement.

However, upper-class dynamic features (such as program and data trace) put a strain on JTAG. For example, the way program trace works is that the target sends a message to the tool every time there's a change of program flow, whether as a result of a branch or an exception. Even though the messages are compressed (see Figure 1), it's likely that a large amount of Branch Trace Messaging (BTM) and Data Trace Messaging (DTM) is required. Furthermore, JTAG was not designed for full-duplex operation or with provisions for handling unsolicited messages from the target.

AUX BOX

To boost bandwidth, Nexus Class II, III, and IV implementations can take advantage of what the standard calls an auxillary port (i.e., extra pins). So, debug capability is scalable across a range of price and performance for both chip and tool suppliers.

In particular, 1, 2, 4, 8, or more pins can be allocated independently for input and output. Typically, messaging from the target to the tool is the bottleneck, so more pins can be assigned. For example, the standard

```
Example of how the target processor generates the address to send in a trace
 message:

Previous absolute address (A1) = 0x003FC01,
Absolute address associated with new trace occurrence (A2) = 0x0003F365

A1 = 0000 0000 0000 0011 1111 1100 0000 0001
A2 = 0000 0000 0000 0011 1111 0011 0110 0101
A1 + A2 = 0000 0000 0000 0000 0000 1111 0110 0100

The unique portion of the address (M1) sent in the message (high-order
    zeros are suppressed):

M1 = 1111 0110 0100

Example of how the tool recreates the address based on its previously calculated
    address and the address contained in the trace message:

Previously calculated address (A1) = 0x003FC01,
Address in message (M1) = 0xF64

A1 = 0000 0000 0000 0011 1111 1100 0000 0001
M1 = 0000 0000 0000 0000 0000 1111 0110 0100
A1 + M1 = 0000 0000 0000 0011 1111 0011 0110 0101

Address recreated by the tool = 0x0003F365
```

Figure 1—*To use limited target-to-host bandwidth in the best way, branch and data trace messages are compressed by sending only the bits that differ from the previous address. Periodically (every 256 messages), or if something goes awry (e.g., message overrun), a full address is sent to maintain and restore synchronization between tool and target.*

Pin name	Connector A	Connector B Opt. 1–IEEE 1149.1	Connector B Opt. 2–Auxiliary Port	Connector B Opt. 3–Combined	Connector C	Comments
MCKI			1		1	Auxiliary port
MDI			2		4	Auxiliary port
*MSEI			1		1	Auxiliary port
MCKO			1	1	1	Auxiliary port
MDO			4	2	8	Auxiliary port
*MSEO			1	1	2	Auxiliary port
*EVTO	1	1	1	1	1	Auxiliary port
*EVTI	1	1	1	1	1	Auxiliary port
*RSTI			1		1	Auxiliary port
PORT					16	Port replacement
IEEE 1149.1 pins	5	5		5		IEEE 1149.1
*RDY	1	1		1		
VREF	1	1	1	1	1	System signals
*RESET	1	1	1	1	1	System signals
CLOCKOUT	1	1				System signals
Vendor-defined	1	1	1	1	2	
UBATT					2	
GROUND	8	13	13	13	38	
Total signals	12	12	16	15	42	
Total pins	20	30	30	30	80	

Table 4—*Nexus defines three connector options (A, B, and C) by using either JTAG, a scalable auxiliary port, or both.*

recommends 1, 2, or 4 pins for a Class III or IV input port (tool to target), but 4, 8, or 16 pins are recommended for a Class III or IV output (target to tool).

To handle the variations, Nexus defines three connectors and multiple configuration options (see Table 4). Connector A (20 pins) is the entry level. Aside from the five JTAG pins, seven optional signals are defined. *RESET allows the tool to reset the target processor. CLOCKOUT is typically driven by the target processor clock and allows the tool to detect target activity and determine a suitable rate for the JTAG clock (TCK), or to clock transfers from target to tool directly.

*RDY allows JTAG transfers to be sped up, as opposed to the polling alternative. V_{REF} is the signaling level of the target debug interface, so a single tool can handle various voltage (e.g., 3-V vs. 5-V) targets. There's also a vendor-defined pin, power for the target being the likely option.

*EVTO and *EVTI provide the critical ability to synchronize debug with external events. For instance, the target can be configured to assert *EVTO to the tool when a particular event like an instruction or data access occurs. This is great for triggering a scope or logic analyzer. Similarly, the tool can drive *EVTI asynchronously, and the target debug logic will output a message of interest, such as the value of the program counter or a particular register or memory location.

The next option, Connector B (30 pins), is the most versatile, supporting three different configurations. The first just maps the 20-pin Connector A (i.e., JTAG plus options) to a subset of the 30 pins.

The second configuration of Connector B replaces JTAG with the aforementioned auxiliary port comprised of separate input (MCKI, MDI, *MSEI) and output (MCKO, MDO, *MSEO) channels.

The third Connector B configuration is a hybrid that restores JTAG by eliminating the auxiliary input port (tool to target) and reducing the output port (target to tool) data pins from four to two.

The final option, Connector C, calls for 80 pins. That may seem like a lot, but don't forget that almost half are grounds. In addition to extra auxiliary port bandwidth (four MDI pins and eight MDO pins), the big claim to fame for Connector C is the 16 pins that are devoted to port replacement.

Port replacement is a variation of the bond-out chip scheme that's been used in traditional emulators in the past. Instead of a special custom bond-out chip, the production chip has an emulator mode that makes the internal buses available on existing pins. Of course, the question then becomes how to reproduce the application function of those pins in the target system.

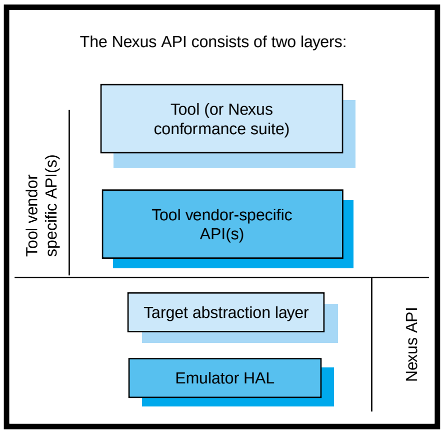

The Nexus API consists of two layers:
Tool (or Nexus conformance suite)
Tool vendor-specific API(s)
Tool vendor specific API(s)
Target abstraction layer
Emulator HAL
Nexus API

Figure 2—*Nexus defines APIs to help decouple high-level tool functions from low-level hardware details of the target chip and emulator hardware.*

The port replacement scheme does that with an extra chip that replicates the function of the pins called to debug duty. The combination of two chips achieves the same end as a custom bond-out chip because it both delivers every pin function of the target device and offers extra pins for debug.

Just how faithfully the port replacements mimic their target counterparts is a concern. In the Nexus scheme, the tool is basically limited to simple parallel I/O operations (i.e., sample inputs and set or clear outputs). Thus, chip makers who want to take advantage of port replacement should take care to multiplex the debug function with simple I/O. Trying to port-replace a complicated high-speed function like PWM or UART could get ugly unless the tool has a lot of horsepower to devote to the cause.

LET'S GET MODULAR

The basic premise of Nexus is to enable a more open and universal development environment for the benefit of everyone—chip designers, tool suppliers, and their customers.

From the chip supplier's perspective, a small additional investment in Nexus silicon helps ensure the availability of development tools, something that shouldn't be taken for granted and has been known to trip up processor wannabes.

Meanwhile, tool designers will be able to leverage their know-how across several architectures without wasting time reinventing the wheel for each new chip that comes to town. I'm sure they won't miss agonizing over whether or not to build, sell, and support their own over-priced, under-popular tools or to crawl around with hat (and checkbook) in hand, trying to convince third-party tool suppliers to bless their chip.

You will have a lot of freedom to mix and match different chips and tools with less of the fire drills that typically accompany such a switch.

With chip and tool vendors both dabbling in debug, there's the danger of wasteful duplication, overlap, and something falling through a crack. To impose a who-does-what discipline, Nexus explicitly defines (with C header files) two APIs, one each for chip and tool suppliers (see Figure 2).

The chip supplier crafts the Target Abstraction Layer (TAL), which maps the standard Nexus debug semantics (e.g., nx_ReadMem, nx_SetEvent, and such) to a particular chip's underlying debug hardware. In turn, the tool supplier comes up with an emulator Hardware Abstraction Layer (HAL) by which a host establishes basic communication with the emulator (e.g., a device driver, if the host is a PC). Together, the two layers comprise the Nexus API, upon which a tool vendor can layer their fancy IDEs.

PLAYING CATCH UP

Remember the old saw, "If the phone system hadn't become automated, everyone on earth would be an operator?" I imagine a world where everyone spends so much time debugging gadgets that they don't have time to enjoy them.

The fact is, without fundamental improvements on the tool front, it's going to get ugly. Trying to brute-force a way out of the debug crisis by throwing more engineers at it will not keep up with the march of silicon or the grandiose aspirations of designers.

I'd like to be able to rely on a few powerful, versatile, easily-upgraded pieces of gear that could work with any chip, and together as well.

Who cares what the Nexus/GEPDIS folks want to call it, let's just wish them luck doing it. ■

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for several years. You may reach him by e-mail at tom.cantrell@circuitcellar.com.

REFERENCES

[1] D.Gonzales, "M-Core Architecture Implements Real-Time Debug Port based on Nexus Consortium Specification," Motorola, www.ieeeisto.org/Nexus5001/northcon_99.pdf.

[2] D.Gonzales and B.Branson, *Real-Time Debugging Highly Integrated Embedded Wireless Devices,* Motorola, www.chipcenter.com/analog/main.html.

SOURCE

Global Embedded Processor Debug Interface Standard
Nexus 5001 Forum
(732) 981-3434
Fax: (732) 562-1571
www.nexus-standard.org

CIRCUIT CELLAR Test Your EQ

Problem 1—What does the array 'hello' contain after four calls to overstrike()? What are the values returned by overstrike()?

```
char hello[] = "hello";
char *ptr = hello;
char overstrike (void) {
    return (++*ptr = 'x');
}
```

Problem 2—Many processors have a special instruction that can be used for implementing process synchronization functions. If your processor does not have such a function, can you still implement a semaphore function for process synchronization?

Problem 3—An embedded 8051 processor (mask ROM) needs a nonvolatile memory to store configuration information. A maximum of 1 KB of configuration data is needed which will be modified one byte at a time no more than once per day. The product will be used 1 hour per day and will be required to last at least 10 years. The best technology to use for the configuration memory is:

A) FlashROM
B) EEPROM
C) Battery-backed CMOS static RAM
D) Battery-backed dynamic RAM
E) Bubble memory
F) None of the above

Problem 4—A 1 MB x 8 bank of memory is required for an ultra-low power battery operated data logger. The data logger will be implemented using a PIC processor which will remain in sleep mode but wake up once per minute to record 100 bytes of data. The datalogger will run unattended for at least one year at a time and data will be downloaded at least once a week from the datalogger then the memory bank reset. The smallest possible power consumption for the finished product is desirable. Which memory technology is best suited for this task?

A) FlashROM
B) EEPROM
C) CMOS static RAM
D) Dynamic RAM
E) Bubble memory
F) None of the above

What's *your* EQ?—The answers and 4 additional questions and answers are posted at www.circuitcellar.com.

You may contact the quizmasters at eq@circuitcellar.com.

8 more EQ questions each month in Circuit Cellar Online see pg. 2

PRIORITY INTERRUPT

Imputed Liability?

Cocktail parties with business people are all alike. Without exception, in a professional crowd someone will jokingly yell out, "They should shoot all the lawyers first!"

I always laugh and agree. At the same time, I'll admit to all of you that I've availed myself of many legal services in the past. Of course, it can always be argued that if lawyers weren't creating a full-employment economy for lawyers in the first place, then I wouldn't have needed them as often.

I only bring this issue up because of all the brouhaha about Napster and copyright infringement. As an avid consumer of information and technology, I want the Internet to be as free from obstacles and regulations as possible. As a supplier and publisher of some of that same copyrighted material, I want to know that it can be protected. Basically, I see both sides of the argument.

In order to understand what is going on with Napster, some of the legal jargon defining copyright infringement has to be explained. A copyright is a federal law that protects the intellectual property of artists and authors when they publish their works. A copyright lasts for the life of the author plus 50 years. "Copyright infringement" simply means the unauthorized use (or posting) of copyrighted material. The real issue in this case revolves around something in copyright law called "imputed liability." Imputed liability doesn't mean that you are specifically violating copyrights by your own action, it just means that you were in the position to control someone else doing it. Under the law that makes you just as liable.

The RIAA and Metallica are contending that Napster is liable because it controls the service through which Napster users are committing individual copyright infringement. (The RIAA could sue each individual for downloading MP3 files of copyrighted music, but the cost and difficulty of pursuing legal action against each "infringer" could be prohibitive. It's more effective to go after the "deep pockets.")

There are various defenses to this argument, which is not a new one. The record industry said that cassettes would destroy the record industry. The motion picture producers said the VCR would end the making of movies. It hasn't happened in both cases because the evolutionary choice to copy the materials to a new medium wasn't driven by a lust for copyright infringement. It was driven by convenience and economics.

Although it's not always the case, file sharing can prove beneficial. Would Microsoft have risen so quickly if the original Microsoft BASIC not been copied to millions of home computers in the early days? Was the end result of all that file sharing the creation of a culture bent on software piracy or was it instead a wider public ready to pay for a regulation upgrade after they had sampled the merchandise? It's not pretty, but that's what happened.

The popularity of MP3 file sharing has a lot to do with overpricing in the music industry. Are you old enough to remember being able to hear a record before buying it in a record store? These days, if it isn't played on a radio station, you basically have to buy the CD and that's your first sample. It wouldn't be so bad if the music industry wasn't also charging more for cheaper-to-produce CDs than other media. That certainly doesn't help music industry popularity.

The music industry wants to eradicate services like Napster. The courts want to apply laws fairly to all. The truth is that neither the music industry nor the laws can keep up with the technology. I don't have a good answer for the present dilemma. All I can say is that the web has become the great equalizer. It takes millions of dollars to hire a staff, design a product, and launch a new company. These days, all it takes to destroy that business model is a 19-year old who designs an application that turns the world upside down. Welcome to the Internet.

Steve

steve.ciarcia@circuitcellar.com