

www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#107 JUNE 1999

COMMUNICATIONS

A Low-Cost Software Modem

**Development Considerations
for Internet Applications**

Circuits for RS-485 Networks

**Build a Compact
Portable Scanner**



Not Just One of the Guys



As I write this, I'm waiting for projects to come in for Design99, Circuit Cellar's eleventh annual design contest, sponsored this year by Motorola. If you've kept up with the trials we've experienced with this contest so far, you're aware of how difficult it has been for some would-be contestants to find sample parts.

Fortunately, people did get their hands on the '908GP20 sample packs. So, for all the difficulties we've had, I'm still expecting a lot of entries to be appearing on the *Circuit Cellar* doorstep in the next week. But what I'm not expecting so much is how many of those entries (or whether *if* any of those entries) will be submitted by women.

Does this potential fact frustrate me? Hell, yes. Why? Because as a doctoral student in a "techie" department and even in a "techie" subfield, I have experienced first-hand the hassles of trying to succeed while being a woman. In fact, I can count on one hand the number of American women who have jobs doing what I went to school to do. Interestingly, there are Asian and European women who do hold such jobs. But Americans? No, not really.

Which is not to say that we (American women interested in the technical side of things) weren't there in the beginning. But somehow, over the years, our interests shifted. I'm the only one still pursuing the original "techie" goal. The others, if they stayed in the program at all, started paying more attention to the "feminine" subfields. Why?

I believe it is a mentoring issue. People talk all the time about how women and men communicate differently. Fine. I can buy that, and given what I read, it seems most people agree on that point. So, here's my question: how do you communicate with a student or new employee who is a woman? If you're male, you probably say that you treat her the same as anyone else, but consider that "anyone else" most likely means one of the men you've worked with for however many years. In other words, you expect her to interact with you as if she were a man.

In my case, there are no women (at least in my department) who I can honestly say have research and career interests similar to mine. The end result is that for me to succeed (i.e., defend my dissertation and receive the Ph.D.), I have to do it in the style of my advisors, all male.

Of course, there are so many issues on the table relating to how women are managing their professions in technical fields (e.g., a recent study at MIT proved that women there were paid less, given less lab space, and so on; to MIT's credit, statistics have made a difference and they are making up for the discrepancies).

However, I think that to maintain (if not expand) what female population there is in the universities and in the professional fields, we first have a very basic problem to solve: how to communicate. Until we address that issue, the changes many of us seek aren't going to even have a chance of success.

Eli

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS
THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Sue Skolnick

MANAGING EDITOR

Elizabeth Laurençot

CIRCULATION MANAGER

Rose Mansella

TECHNICAL EDITORS

Michael Palumbo

Rob Walker

CHIEF FINANCIAL OFFICER

Jeannette Ciarcia

ART DIRECTOR

KC Zienka

WEST COAST EDITOR

Tom Cantrell

ENGINEERING STAFF

Jeff Bachiochi

CONTRIBUTING EDITORS

Ingo Cyliax

Ken Davidson

Fred Eady

PRODUCTION STAFF

Phil Champagne

John Gorsky

NEW PRODUCTS EDITOR

Harv Weiner

PROJECT EDITOR

Janice Hughes

EDITORIAL ADVISORY BOARD

Ingo Cyliax

Norman Jackson

David Prutchi

Cover photograph Ron Meadows—Meadows Marketing

PRINTED IN THE UNITED STATES

ADVERTISING

ADVERTISING SALES MANAGER

Bobbi Yush
(860) 872-3064

Fax: (860) 871-0411
E-mail: bobbi.yush@circuitcellar.com

ADVERTISING COORDINATOR

Valerie Luster
(860) 875-2199

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

CONTACTING CIRCUIT CELLAR INK

SUBSCRIPTIONS:

INFORMATION: www.circuitcellar.com or subscribe@circuitcellar.com
TO SUBSCRIBE: (800) 269-6301 or via our editorial offices: (860) 875-2199

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 INK, 4 Park St., Vernon, CT 06066

AUTHOR CONTACT:

E-MAIL: Author addresses (when available) included at the end of each article.
ARTICLE FILES: [ftp.circuitcellar.com](ftp://circuitcellar.com)

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

CIRCUIT CELLAR INK®, THE COMPUTER APPLICATIONS JOURNAL (ISSN 0896-8985) is 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, 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 INK Subscriptions, P.O. Box 698, Holmes, PA 19043-9613 or call (800) 269-6301.

Postmaster: Send address changes to Circuit Cellar INK, Circulation Dept., P.O. Box 698, Holmes, PA 19043-9613.

Circuit Cellar INK® makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors. Furthermore, because of possible variation in the quality and condition of materials and workmanship of reader-assembled projects, *Circuit Cellar INK®* disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in *Circuit Cellar INK®*.

Entire contents copyright © 1999 by Circuit Cellar Incorporated. All rights reserved. Circuit Cellar and *Circuit Cellar INK* are registered trademarks of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.


12 **Low-Cost Software Bell-202 Modem**
Stephen Holland


20 **Designing RS-485 Circuits**
Jan Axelson


26 **A Web-Based Chart Recorder**
Paul Breed

58 **Embedded OSs for Internet Appliances**
David Brooks

64 **Compact Optical Image Scanner**
John Luo

68  **MicroSeries**
USB Primer
Part 2: Classes and Drivers
Jim Lyle

74  **From the Bench**
Look Ma, No Hands
The Qprox Touchless Sensor
Jeff Bachiochi

78  **Silicon Update**
XLR8R
Working with Accelerometers
Tom Cantrell

Task Manager 2
Elizabeth Laurençot
Not Just One of the Guys

Reader I/O 6
Circuit Cellar Homepage

New Product News 8
edited by Harv Weiner

Test Your EQ 83

Advertiser's Index 95
July Preview

Priority Interrupt 96
Steve Ciarcia
Servings Per Issue

INSIDE ISSUE 107

38 **Nouveau PC**
edited by Harv Weiner

41 **Embedded Ethernet Fundamentals**
Aaron Feen

45 RPC **Real-Time PC**
Astronomical Issues—Part 3: Filters and Undersampling
Ingo Cyliax

52 APC **Applied PCs**
Embedded Internet—Part 2: TCP/IP and a 16-Bit Compiler
Fred Eady

EMBEDDED PC

READER I/O

Y2K UPS AND DOWNS

As an avid reader of *Circuit Cellar* and Steve Ciarcia, Priority Interrupt usually gets my first glance each month. After reading "Sitting in the Dark" (*Circuit Cellar* 105), I felt some clarification was necessary.

With 26 years of experience on the technical side of the elevator business and as the founder of Integrated Display Systems, a company that produces the leading networked PC-based elevator management system (with lots of embedded processors and plenty of Y2K potential), I felt qualified to comment.

Over 70% of the elevators in the U.S. are controlled by relay or TTL logic and have no microprocessor or knowledge of the date. Date-aware microprocessors have only been used on elevator systems installed within the last 15 years or so. Of the date-aware group, those that shut down based on a date, Y2K or otherwise, are less than 1% (excluding external security systems, etc.). The algorithm is usually based on the number of days since the last reset and has nothing to do with the year.

Because of the need for absolute safety, elevators shut down on their own for dozens of reasons. On any given day, at least 1.6% of the elevators in the US shut down for valid reasons. Any additional shutdowns on Jan. 1, 2000 will hardly be noticed, let alone bring the country to its knees.

Now, your refrigerator may spray ice cubes all over the floor, but most elevators will remain blissfully ignorant. But, if it happens to be the elevator in your building...

Winslow D. Soule
winssoule@msn.com

ROCK, CANNONBALL, OR WHAT-HAVE-YOU

The cover and Task Manager of *Circuit Cellar* 106 brought back memories of when I lived in northeast Nebraska. Out there, they have what's known as a Nebraska Weather Station (NWS). The NWS is usually set up in the farmyard where it can be seen from a kitchen window.

It consists of a heavy steel tube post, 5" or 6" in diameter, set with about 5' visible above ground. To prevent the post from moving too easily, it's set in a yard or so of concrete. A piece of 1/2" logging chain about 3' long is welded to the top of the post and a cannonball (generally a 22 pounder) is welded to the other end of the chain.

The weather report goes like this:

- if the ball is wet, it's raining.
- if the ball is white, it's snowing.
- if the ball is buried in the snow, get out the shovels, one for each kid.
- if the ball is standing straight out, it's too windy to go out and chore.

Although some people don't believe me, one farmhouse that I drove past every day actually had a NWS. I even saw the cannonball blowing out around 25° a time or two. That's breezier than the standard three-garbage-can-lids rating and makes it rather difficult to stay between the ditches while driving.

Gene Heskett
geneheskett@iolinc.net

Circuit Cellar
HOMEPAGE

www.circuitcellar.com

Newsgroups

The cci newsserver is the place to go for on-line questions and advice on embedded control, announcements, or to let us know your thoughts about Circuit Cellar.

The June
Design Forum
password is:

Grape

New!

- Whether you've got extra components lying around or you're just looking for some used parts, the Circuit Cellar **Parts Bin** is the place to go. The Parts Bin is where engineers can buy, sell, or trade parts, components, and tools of the trade. Every project involves a trip to the parts bin, so if it's not in your parts bin, try ours. See our homepage for submission guidelines.
- All three **searchable CD-ROMs** of the *Circuit Cellar* back issues are ready to be shipped, so stop by our homepage and place your order!

Design Forum

Silicon Update Online: The Modem Squad—Tom Cantrell
Lessons from the Trenches: Logging in the '90s: Part 2—George Martin
Getting Your Product to Work Outside the Lab: Part 1—Enduring Environmental Challenges—George Novacek

NEW PRODUCT NEWS

Edited by Harv Weiner



SINGLE-BOARD COMPUTER

Industrologic has introduced the **SBC-1 Single-Board Computer/Controller** board and development package. Based on the venerable 80C51 processor with 8 KB of EEPROM and 8 KB of RAM, the SBC-1 features a powerful ROM-based monitor and developer interface, including commands that permit it to be operated as a slave to any RS-232 device. Also featured is an onboard, enhanced TinyBASIC with functions and variables defined for the unit's I/O, serial port, and real-time clock.

The SBC-1 includes 50 DIO signals when used in its maximum I/O configuration (24 TTL inputs, two interrupt inputs, 24 open-collector outputs), a 0-5-V 8-bit ADC input, two RS-232 physical ports with DE-9 connectors, and a real-time clock that can be set or read from the monitor or TinyBASIC. The serial upload and download capability can be used to store and retrieve user programs, and the system I/O routines are in a "call table" toolbox for user access. All chips are installed in sockets for easy replacement, if necessary. The board features terminal block connectors to allow quick connection to many of the signals.

The SBC-1 includes the circuit board, a serial port cable for connection to a PC-compatible computer, wall-block power supply, host-computer software with programming examples, and an extensive hardware and software reference manual.

The SBC-1 package is available for **\$179**.

Industrologic, Inc.
(314) 707-8818
www.industrologic.com

SPRING-CONTACT TEST PROBE

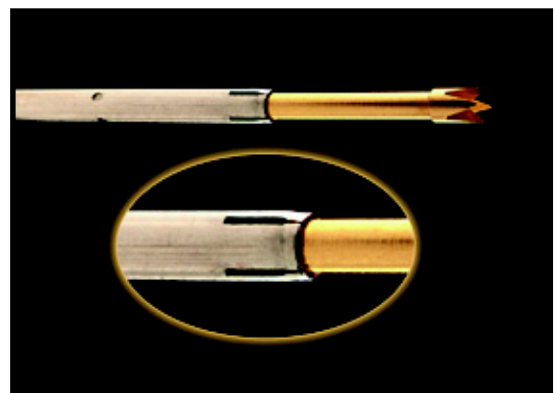
A revolutionary spring-contact probe is available from Interconnect Devices. The **ICT Probe** features four bifurcated beams at the top of the barrel that provide uniform radial pressure on the plunger shaft, holding the plunger parallel with the barrel. This construction provides a constant contact between the plunger and barrel at all times during the actuation process without any biasing. The probe is available for 0.100" (2.54 mm), 0.075" (1.91 mm), and 0.050" (1.27 mm) centers.

The mechanical benefits of this design include reduced probe wear and elimination of sideloading or biasing of the plunger, a wiping action on every deflection to ensure good electrical contact, and a typical pointing accuracy of less than 0.001" (0.025 mm). The electrical benefits include extremely low resistance (as low as 8 m Ω on a 0.100" [2.54 mm] center probe), consistent resistance (deviation of less than 5 m Ω throughout the entire probe life), and a consistent resistance during compression.

The ICT Probe is also ideal for contaminated or no-clean environments. The compliant or pressure fit between the plunger and the barrel virtually seals the internal portion of the probe from contaminants. This lack of abrasive and nonconductive grit normally found in a probe prolongs the device's life both mechanically and electrically.

Prices range from **\$0.46 to \$2.34**, depending on model and quantity.

Interconnect Devices, Inc.
(913) 342-5544
Fax: (913) 342-7043
www.idinet.com



NEW PRODUCT NEWS

RTD/TC SIGNAL CONDITIONER

The **MCR-T/UI Universal Signal Conditioner** accepts any RTD/thermocouple signal and converts, isolates, filters, and amplifies in one module. The module addresses any RTD sensor and thermocouple type, enabling RTDs in platinum, copper, or nickel to be converted to a standard industrial analog signal. Because any temperature sensor type, temperature range, and output can be programmed, this module is excellent for use anywhere in the process.

All standard outputs can be programmed, and standard signals of 0–20 mA, 4–20 mA, 0–5 VDC, 0–10 VDC,

+5 VDC, and 110 VDC can be easily selected. The module features a programmable high-/low-transistor alarm output at 24 VDC, 100-mA switching capability, and programming for line break, overrange, and

underrange. The temperature range of the module is -200°C to $+800^{\circ}\text{C}$ for RTDs and -200°C to $+2300^{\circ}\text{C}$ for thermocouples.

All parameters are set by a simple Windows-based configuration software

package. The module is featured in the unique MB-housing design and only takes 17.5 mm of DIN rail space.

Pricing for the MCR-T/UI is **\$299** in single quantities.



Phoenix Contact, Inc.
(800) 322-3225
(717) 944-1300
Fax: (717) 944-1625
www.phoenixcontact.com

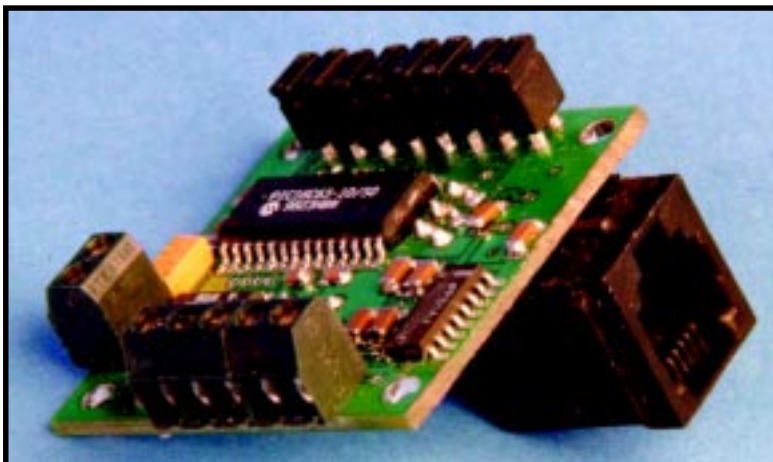
REMOTE TEMPERATURE MODULE

The **JDS-148 Remote Temperature Module** is designed for general-purpose temperature measurements. It interfaces to Dallas Semiconductor's DS-1820 or DS-1821 temperature controllers and provides temperature values over either RS-232 or multidrop RS-485/-422.

Each module automatically scans and records the temperature conversions and communicates back to a host computer with ASCII commands.

The JDS-148 communicates over a 1-Wire interface and features jumper-selectable data rates to 38.4 kbps. It auto-identifies the controller ID and includes 32 unique module addresses. The $1.5" \times 1.5"$ board operates on an input voltage from 8 to 15 VDC, and it has a measurement range from -15°C to 105°C with an accuracy of $\pm 0.5^{\circ}\text{C}$.

The JDS-148 sells for **\$45** (RS-232 or RS-485/-422 version).



J-Works, Inc.
(818) 361-0787
Fax: (818) 270-2413
www.j-works.com

NEW PRODUCT NEWS

DEVELOPMENT BOARD

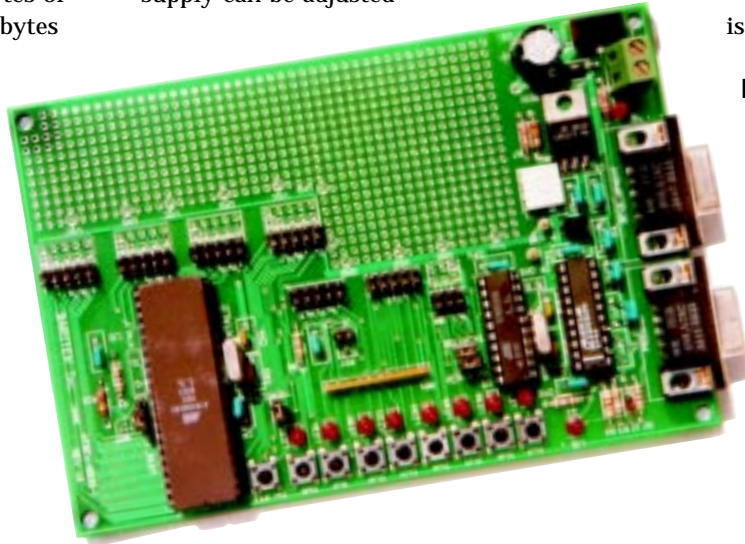
The **ATDEV-8535-1 Development/Prototype Board** for the Atmel AT90LS8535 microcontroller is now available from Baritek. The AT90LS8535 features 8 KB of flash program memory, 512 bytes of internal SRAM, and 512 bytes of EEPROM. This device also has an operating-voltage range from 2.7 to 6 V, eight 10-bit A/D channels, and a programmable UART.

The board features in-circuit programming from the serial port of a PC. All of the microcontroller ports are brought out to 10-pin headers, and a large prototyping area is available for user signal-conditioning or

display devices. Also featured are eight momentary push-button switches and eight LEDs that can be connected to any port of the AT90LS8535 via a 10-pin ribbon cable. The onboard power supply can be adjusted

from 1.25 to 6.0 V to simulate battery operation. The ATDEV-8535 was designed to support Atmel's free development tools for the AT90LS8535.

The ATDEV-8535-1 is priced at **\$89.95**.



Baritek, Inc.
(781) 749-2550
Fax: (781) 749-3151
www.baritek.com

FEATURE ARTICLE

Stephen Holland

Low-Cost Software Bell-202 Modem

Imagine this: rapid and reliable data processing via a low-cost software modem. With the 50-MIPS Scenix SX MCU, Stephen designed just such a modem, and it offers the ease of flash memory and in-system debugging.



In every ATM, point-of-sale terminal, and automated gas pump there's an embedded modem whose only purpose is sending and receiving identity-verification data and purchase information. These are typical of a broad range of applications in which "data communications" doesn't mean transmitting huge pdf files or other attachments between PCs.

Instead, embedded modems for information processing need to connect rapidly and reliably as well as be able to transmit and receive relatively small amounts of data at a rate that a person standing in front of an ATM perceives as fast. For this, low-speed (1200 and 2400 bps) Bell-202- and 212-compatible modems are adequate.

Because embedded modems are often used in places that require small size and low cost, designers must get a least-cost implementation that is as close to a single-chip solution as possible. An attractive solution would be one relatively cheap MCU with all the modem functions inside, eliminating everything except a serial connection and the digital access arrangement (DAA) for the telephone line.

But, this solution wasn't available until recently. Embedded modem designers typically had to go with either a multichip implementation consisting of an inexpensive 8-bit MCU and an external modem chip, or move up the complexity and cost curve to 16- and 32-bit MCUs and DSPs that avoid the silicon penalties by handling the modem functions in software.

There's an 8-bit MCU that performs the types of modem functions needed by the vast majority of embedded applications in software. Using a Scenix Semiconductor SX series MCU, I put together a circuit that, in a form factor smaller than 2" x 3", provides all the basic functionality of a Bell-202 modem, including FSK generation and detection and DTMF generation and detection.

The key to the design is that software modules, or "virtual peripherals," for each function are loaded into a fast (10-ns access time) on-chip flash program memory and executed as needed. This eliminates external modem and memory chips and additional internal silicon development, and results in an extremely cost-effective implementation. In fact, the entire bill of materials totals right around \$7.

MCU ARCHITECTURE

For an 8-bit MCU to do what the 16- and 32-bit MCUs do, it needs many of the same architectural features, including a streamlined, four-stage RISC-like pipelined architecture to minimize code size and maximize performance.

Coupled with extremely fast on-chip instruction and data memory, this arrangement permits every instruction to be executed in a single clock cycle. A 50-MHz clock can provide an instruction throughput rate of 50 MIPS, and a 100-MHz clock gives 100 MIPS.

Another requirement is a deterministic interrupt-response capability that services interrupts in a small and precise number of cycles every time. With older architectures, tasks are only interrupted at instruction boundaries, so the number of cycles required to respond to an interrupt is unpredictable.

This setup not only produces slow interrupt responses but also introduces jitter into the system timing, which limits performance and accuracy. Short

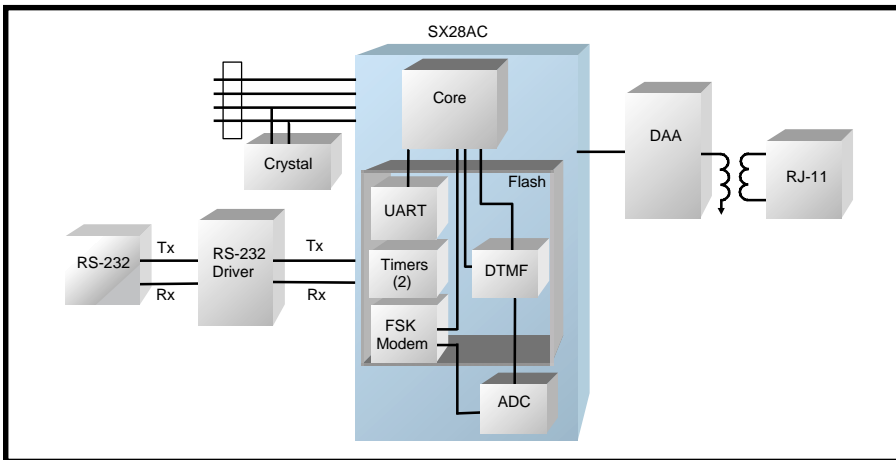


Figure 1—A full-featured modem can be implemented with the SX28AC 8-bit MCU. The combination of high-speed MCU and software peripheral functions results in an extremely compact and cost-effective implementation.

predictable interrupt-response times, with critical registers automatically stored in special hardware stacks during an interrupt, eliminate the problem of jitter and ensure proper execution of virtual peripheral functions.

AT THE BLOCK LEVEL

The modem design in Figure 1 uses the Scenix SX28AC, which runs at up to 50 MHz and has a 50-MIPS data throughput rate. Although the design accommodates connecting to a PC via an RS-232 connector, it can be changed to some other type of serial interface.

The modem functions can be downloaded from the Scenix web site as virtual peripherals and stored in the on-chip program memory. The virtual peripherals used in this design are:

- two 16-bit timers—one for the power-on LED and one for FSK and DTMF tone duration
- DTMF detection
- FSK detection via on-chip hardware comparator
- DTMF generation
- FSK generation
- UART (1200 bps to 115.2 kbps)

Additional features like caller ID, voice recognition, LCD drive, and numerous types of I/O can be added by downloading the appropriate virtual peripheral into the program memory.

To achieve the lowest implementation cost, the design uses a component (rather than a module) approach for connecting to the external world. For example, the cost of a DAA module is typically about \$4. But, a 1200-bps modem doesn't need the sophisticated DAC, compression and decompression, and other functions performed by a DAA module.

Instead, as you see from Figure 2, I used a configuration based on optoisolators and a transformer to provide coupling to the telephone network, which complies with the Bell-202 standard and makes the cost of the components significantly less than that of a DAA module.

DTMF AND FSK DETECTION

Connection with the telephone network is controlled by the Ring and Hook signals, which are passed through optoisolators U5 and U6 to either establish or break the connection.

When the connection is made, data received from the network is coupled

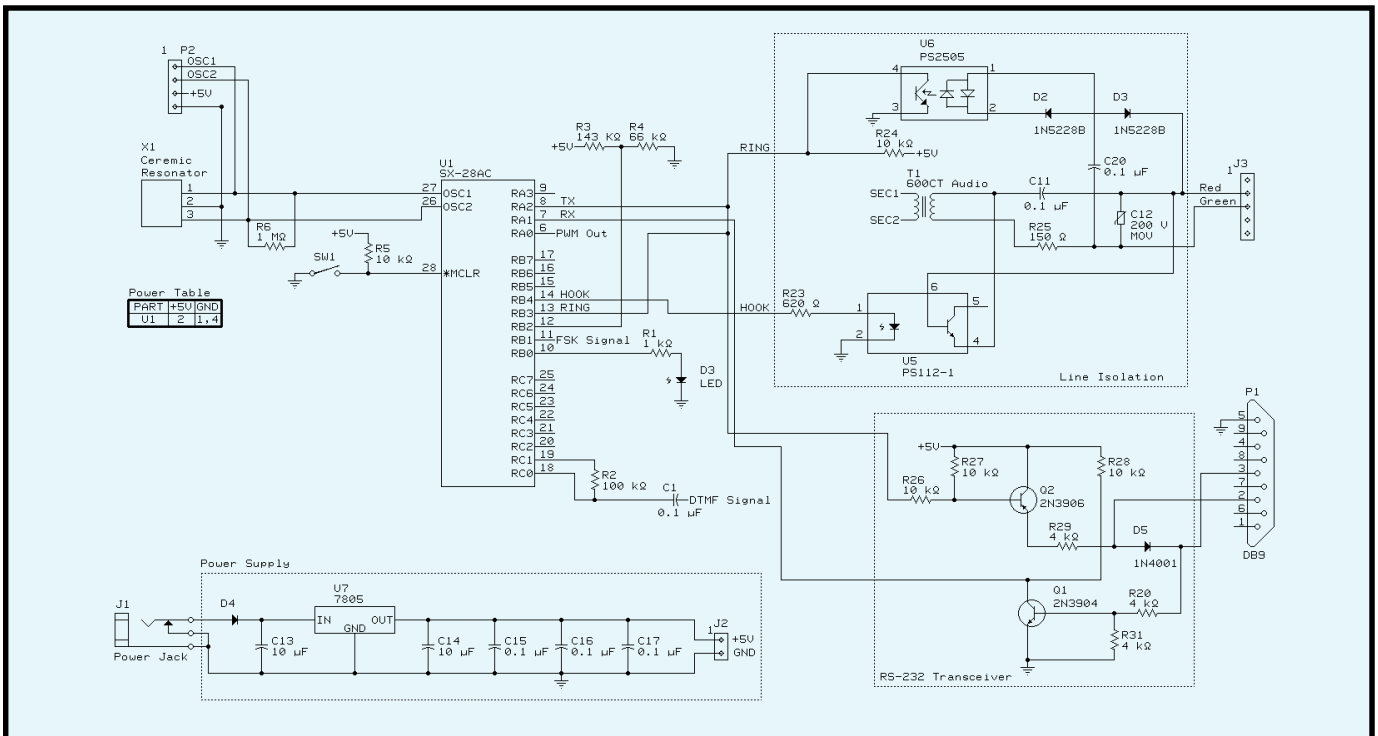


Figure 2—Using discrete components rather than sophisticated modules for the DAA and RS-232C interfaces dramatically decreases the modem bill of materials. At 1200 bps, this configuration meets the Bell-202 standard.

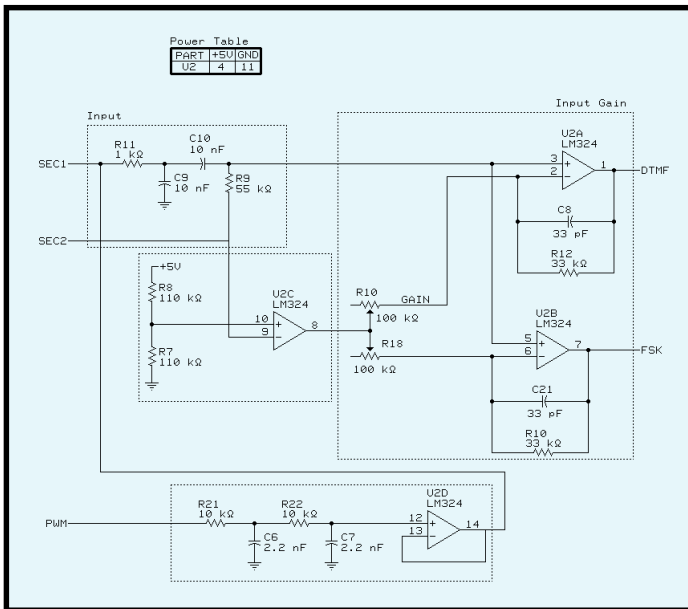


Figure 3—The op-amp splits the DTMF and FSK portions of the single input stream. It also shifts the DC offset of the PWM output to meet telephone network requirements.

through audio transformer T1, as well as input filter and DC offset circuit to an operational amplifier, U2 (see Figure 3). The op-amp splits the single bit-stream of the input into separate DTMF and FSK signals and applies them to the MCU, where they are converted from analog to digital and processed.

The DTMF signal is applied to pins RC0 and RC1 of the SX28AC. The MCU converts this from analog to digital by sampling it in software using a digital form of the Geortzel discrete Fourier transform (DFT) algorithm. Eight separate DFTs, each with an accuracy of greater than $\pm 1\%$, simulta-

neously sample the signal looking for the row and column frequencies that identify digits within the DTMF matrix.

To be properly selective on the target frequencies, a performance level of at least 50 MIPS is required; the sine and cosine reference signals for each DFT operation are being generated in software within the interrupt service routine to enable multiple virtual peripherals to run in parallel. A clock frequency of at least 50 MHz is required to give the DFT accuracy better than the required DTMF specification of $\pm 1.5\%$ (i.e., the higher the clock rate, the better the accuracy).

The same DFT technique would be possible at a much lower MIPS rate, but at the sacrifice of needing to be executed in straight-line code to maintain accuracy. Additionally, the standard for the DTMF tone duration lets it be as short as 48 ms, which the SX28AC can easily handle.

The current DTMF algorithm can detect tones as short as 14 ms. By contrast, most other 8-bit MCU implementations need at least 150 ms to detect the tone because of the straight-line nature of the code.

The FSK signal from the op-amp is applied to pin RB1 of the SX28AC. Detecting the FSK data (1200 Hz representing a logic 1, and 2200 Hz representing a logic 0) and its conversion to a digital format is done by an on-chip hardware comparator and a reference level set by an external resistor voltage divider (R3, R4) using a form of zero-crossing detection. The comparator is one of the basic sets of silicon peripherals included in the SX28AC.

The digital DTMF and FSK data is processed by the software UART and transmitted to a PC through a simple implementation of an RS-232 interface. Because the connection to the PC isn't meant to be over a long distance, a two-transistor (Q1, Q2) circuit operating at 5 V rather than the more-typical 9 V is sufficient to drive the line.

DTMF AND FSK GENERATION

Digital data received from the PC through the RS-232 interface and software UART is converted to analog signals for transmission over the telephone network using PWM techniques.

To minimize I/O pin usage, a single PWM output pin is used. This arrangement is possible even for the two DTMF tones because the SX28AC sums the DTMF signals internally to create a single signal.

To ensure the smooth frequency shifts that are required by the Bell-202 specification, all the FSK shifts are phase-coherent. Although this requirement is easily accomplished in hardware, it requires a level of processing power only available in the SX series to work in software.

The PWM output of the MCU is applied to the op-amp, where its DC offset is moved to accommodate the needs of the telephone network. The result is applied to the acoustic transducer and on to the network.

Even though it provides all the functionality of a complete 1200-bps modem, this design uses only nine of the 20 I/O pins available with the SX28AC MCU.

It also doesn't use all of the device's 50 MIPS of processing power. This setup makes it easy to add features

like an LCD interface, motor controller, or I²C-to-EEPROM interface by loading the appropriate virtual peripherals into the on-chip memory and using the extra I/O pins.

APPLICATION DEVELOPMENT

Adding functions to an SX28AC-based design is as simple as going to a web site and downloading the desired virtual peripherals. New software

Listing 1—Developing the Bell 202-compatible modem program was simplified by using an Edit window on a PC. This code describes the initialization sequence for the design.

```

reset_entry
    mov     m,#$0f
    mov     ra,%%0110      ;init ra
    mov     !ra,%%0010     ;ra0-1 = input,ra2-3 = output
    mov     rb,%%00000000 ;init rb
    mov     !rb,%%00001110 ;rb1-3 = input,rb4-7 = output
    mov     rc,%%00000000 ;init rc
    mov     !rc,%%11111101 ;rc0,rc2-7 = input,rc1 = output
    mov     m,$0d          ;make rc0 cmos-level
    mov     !rc,%%11111110
    clr     fsr             ;reset all ram banks
:loop    setb    fsr.4
        clr     ind
        ijnz   fsr,:loop
        bank   dtmf_gen      ;Initialize variables
        mov    delay,#23
        clr    flags
        mov    !option,%%00011111
                ;enable wreg and rtcc interrupt
        jmp    @main         ;Jump to main code
        ...

```

development and programming are equally painless.

The chip is programmed over its oscillator pins (26, 27) using the SX-Key development system (a 0.5" × 1.5" module that connects at one end to a PC by a standard DE-9 connector and at the other end to the system board by a four-pin header interface connector). The SX-Key software, which includes an editor, programmer, and debugger, can be installed on a PC and run under Windows.

With the SX-Key software loaded and the PC connected to the circuit through the hardware module, software development begins immediately. The first screen is a device setup window.

It lets you set parameters such as the number of pins and on-chip memory size of the SX-series MCU you're using, the clock source (an external frequency generator in this design, although a range of internal oscillator frequencies can also be selected), and



Photo 1—The Debug window of the SX-Key development system provides visibility into the contents of all device registers, allowing individual bits to be changed as necessary.

additional options like stack extensions and an on-chip watchdog timer. A configuration window lets you select the desired communication port and specify the erase and program times for the flash program memory.

With the chip parameters established, an Edit window lets you write the program. All timing-dependent

operations must be run within an interrupt service routine, so the code needs to be written using an interrupt indicator.

An on-chip hardware timer produces a real-time clock/counter (RTCC) signal that generates interrupts. The RTCC can be set to establish how many clock cycles occur before an interrupt must be serviced.

The general program flow is:

- (load RTCC countdown)
- run application code
- RTCC times out, issues interrupt
- interrupt service routine
- run peripheral routine
- load RTCC countdown
- RETI (return from interrupt)
- run application code

Listing 1 shows the initialization part of the program for this modem design.

After resetting the SX chip, the I/O ports and the default values for some software variables must be initialized. The final line of code in the reset entry code typically loads the Option

register with the proper value to enable RTCC clock rate and interrupts.

In this design, the contents of the RTCC are incremented each clock cycle. If the RTCC overflows, an interrupt request follows. The RTCC overflow interrupt is essential to the virtual-peripheral concept, as it provides a jitter-free time base that is used as the time slice for the peripherals. Much like an RTOS, this master clock can then increment other clocks specific to each task.

The completed program and the virtual peripherals are loaded into the MCU's on-chip program memory through the SX-Key module. A Debug window (see Photo 1) displays the result of programming the chip and provides interactive debugging options.

The left column shows the contents of registers 00-0F of the selected RAM bank in hex and binary numbers. The contents of registers 10-1F of the remaining RAM banks are displayed on the right side.

In the middle of the Debug window, the contents of the M and W registers

are shown as well as interrupt and skip flags. Beneath that is listed the address, opcode, and assembler mnemonic of the selected part of the program memory.

Buttons for emulation functions are arranged across the window bottom:

- step—executes one instruction
- walk—executes multiple instructions, one at a time
- run—executes instructions at full speed
- stop—halts execution of a walk or run operation
- reset—resets the MCU
- exit—closes the Debug window

To debug the program, just reset the contents of any of the registers. All the changed registers are marked with red during debugging so you don't lose the track of what you're doing.

When you're satisfied that you've corrected the bugs, reprogram the chip. Because you're working with flash memory, you can go through as many iterations as you want to get it right.

AT LAST, A CHOICE

For about \$7, I was able to design a compact fully functional embedded modem that meets the needs of all kinds of applications. It was possible because designers have a choice not only of modems, but also of every imaginable type of embedded system. ☒

Stephen Holland is a senior applications engineer at Scenix Semiconductor. Before joining Scenix in early 1998, he worked for over six years in the electronics and computer industries in Canada and Hong Kong. You may reach him at stephen.holland@scenix.com.

SOFTWARE

Software for this article is available via the Circuit Cellar web site.

SOURCE

SX28AC

Scenix Semiconductor, Inc.
(408) 327-8888
Fax: (408) 327-8880
www.scenix.com

FEATURE ARTICLE

Jan Axelson

Designing RS-485 Circuits

Jan knows that RS-485 is perfect for transferring small blocks of information over long distances, and she finds the RS-485 standard extremely flexible. Here, she shows us several circuits for RS-485 networks.



When a network needs to transfer small blocks of information over long distances, RS-485 is often the interface of choice.

The network nodes can be PCs, microcontrollers, or any devices capable of asynchronous serial communications. Compared to Ethernet and other network interfaces, RS-485's hardware and protocol requirements are simpler and cheaper.

The RS-485 standard is flexible enough to provide a choice of drivers, receivers, and other components depending on the cable length, data rate, number of nodes, and the need to conserve power.

Several vendors offer RS-485 transceivers with various combinations of features. Also, there are options for methods of terminating and biasing the line and controlling the driver-enable inputs.

In this article, I show you several circuits for RS-485 networks. Even if you use prebuilt cards or converters, understanding the options will help you choose the right product and configure it to get the best results for your application.

RS-485 IN BRIEF

But first, a quick look at RS-485. The interface popularly known as RS-485 is an electrical specification for multipoint systems that use balanced lines. RS-485 is similar to RS-422, but RS-422 allows just one driver with multiple receivers whereas RS-485 supports multiple drivers and receivers.

The specification document (TIA/EIA-485-A) defines the electrical characteristics of the line and its drivers and receivers. There are brief suggestions relating to terminations and wiring, but there's no discussion of connector pinouts or software protocols (as there is for RS-232).

An RS-485 network can have up to 32 unit loads, with one unit load equivalent to an input impedance of 12k. By using high-impedance receivers, you can have as many as 256 nodes.

An RS-485 link can extend as far as 4000' and can transfer data at up to 10 Mbps, but not both at the same time. At 90 kbps, the maximum cable length is 4000', at 1 Mbps it drops to 400', and at 10 Mbps it drops to 50'. For more nodes or long distances, you can use repeaters that regenerate the signals and begin a new RS-485 line.

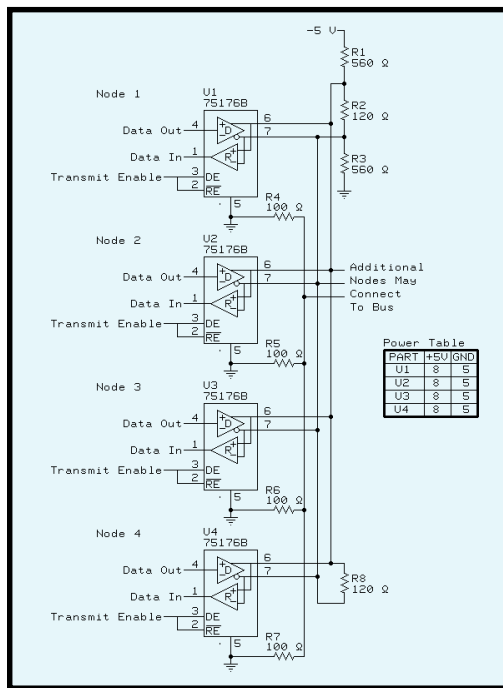
Although the RS-485 standard says nothing about protocols, most RS-485 links use the familiar asynchronous protocols supported by the UARTs in PCs and other computers. A transmitted word consists of a start bit followed by data bits, an optional parity bit, and a stop bit.

Two ways to add RS-485 to a PC are on an expansion card and by attaching an RS-485 converter to an existing port. Converters for RS-232 are widely available and Inside Out Networks has developed a USB-to-RS-485 converter, also available from B&B Electronics. On microcontrollers, you can connect an RS-485 transceiver to any asynchronous serial port.

Many network circuits also require a port bit to control each transceiver's driver-enable input. Ports designed for RS-232 communications can use the RTS output. If that's not available, any spare output bit will do.

Most serial-communications tools, including Visual Basic's MSComm, support RS-485 communications with

Figure 1—This general-purpose RS-485 network can have up to 32 nodes. Biasing resistors ensure that there are no false start bits when no drivers are enabled.



RTS controlled in software. The COMM-DRV serial-port drivers from WSCS have automatic RTS control built-in.

The main reason why RS-485 links can extend so far is their use of balanced, or differential, signals. Two wires (usually a twisted pair) carry the signal voltage and its inverse. The receiver detects the difference between the two. Because most noise that couples into the wires is common to both wires, it cancels out.

In contrast, interfaces like RS-232 use unbalanced, or single-ended, signals. The receiver detects the voltage difference between a signal voltage and a common ground.

The ground wire tends to be noisy because it carries the return currents for all of the signals in the interface, along with whatever other noise has entered the wire from other sources. And noise on the ground wire can cause the receiver to misread transmitted logic levels.

The datasheets for interface chips label the noninverted RS-485 line as line A and the inverted line as line B. An RS-485 receiver must see a voltage difference of just 200 mV between A and B. If A is at least 200 mV greater than B, the receiver's output is a logic high. If B is at least 200 mV greater than A, the output is a logic low. For differences less than 200 mV, the output is undefined.

At the driver, the voltage difference must be at least 1.5 V, so the interface tolerates a fair amount of non-common-mode noise and attenuation.

Vendors for RS-485 transceivers include Linear Technology, Maxim, National Semiconductor, and Texas Instruments. These companies are also excellent sources for application

notes containing circuit examples and explanations of the theory behind them.

RS-485 is designed to be wired in a daisy-chain or bus topology. Any stubs that connect a node to the line should be as short as possible. Most links use twisted pairs because of their ability to cancel magnetically and electro-magnetically coupled noise.

GENERAL-PURPOSE LINK

Figure 1 shows a general-purpose RS-485 network. Each node has a Texas Instruments SN75176B transceiver that interfaces between RS-485 and TTL logic levels.

The chip has a two-wire RS-485 interface, a TTL driver input and receiver output, and TTL enable inputs for the driver and receiver. Similar chips include Linear Technology's LTC485, Maxim's MAX485, and National Semiconductor's DS3695.

The circuit has two 120-Ω terminating resistors connected in parallel, at or just beyond the final node at each end of the link. One end of the link also has two 560-Ω biasing resistors.

The terminations reduce voltage reflections that can cause the receiver to misread logic levels. The receiver sees reflected voltages as output switches, and the line settles from its initial current to its final current. The termination eliminates reflections by making the initial and final currents equal.

The initial current is a function of the line's characteristic impedance, which is the input impedance of an infinite open line. The value varies with the wires' diameters, the spacing between them, and the insulation type.

For digital signals (which consist mainly of frequencies greater than 100 kHz), the characteristic impedance is mostly resistive; the inductive and capacitive components are small. A typical value for 24-AWG twisted pair is 120 Ω.

The final current is a function of the line termination, the receivers' input impedance, and the line's series impedance. In a typical RS-485 line without a termination, the initial current is greater than the final current because the characteristic impedance is less than the receivers' combined input impedance.

On a line without a termination, the first reflection occurs when the initial current reaches the receiver. The receiver's input can absorb only a fraction of the current. The rest reflects back to the driver. As the current reverses direction, its magnetic field collapses and induces a voltage on the line. As a result, the receiver initially sees a greater voltage than what was transmitted.

When the reflected voltage reaches the driver, which has a lower impedance than the line, the driver absorbs some of the reflection and bounces the rest back to the receiver. This reflection is of opposite polarity to the first reflection and causes the receiver to see a reduced voltage. The reflections bounce back and forth like this for a few rounds before they die out and the line settles to its final current.

If the line terminates with a resistor equal to the line's characteristic impedance, there are no reflections. When the initial current reaches the termination, it sees exactly what it was expecting—a load equal to the line's characteristic impedance. The entire transmitted voltage drops across the load. In a network with two parallel terminations, the drivers drive two lines with each ending at a termination.

The biasing resistors hold the line in a known state when no drivers are enabled. Most RS-485 transceivers

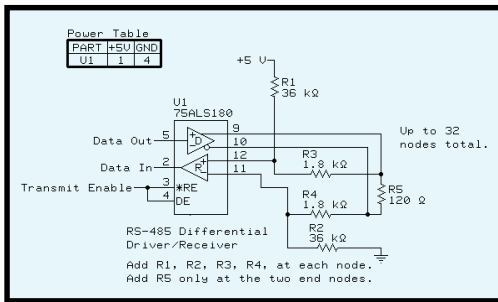


Figure 2—The biasing resistors in this network hold the receiver's inputs in an idle state when no drivers are enabled, or if the node disconnects from the network, or if the signal lines are shorted.

have internal biasing circuits, but adding a termination defeats their ability to bias the line. A typical internal circuit is a 100-k Ω pullup from line A to V+, and a 100-k Ω pulldown from line B to ground.

With no termination and when no drivers are enabled, the biasing resistors hold line A more positive than line B. When you add two 120- Ω terminations, the difference between A and B shrinks to a few millivolts, much less than the required 200 mV. The solution is to add smaller resistors in parallel with the internal biasing so that a greater proportion of the series voltage drops across the termination.

The size of the biasing resistors is a tradeoff. For a greater voltage difference and higher noise immunity on an idle line, use smaller values. For lower power consumption and a greater differential voltage on a driven line, use larger values.

When the receiver is disabled, the receiver's output is high impedance. If the output doesn't connect to an input with an internal pullup, adding a pullup here ensures that the node doesn't see false start bits when its receiver is disabled.

To comply with the specification, all of the nodes must share a common ground connection. This ground may be isolated from earth ground.

The ground wire provides a path for the current that results from small imbalances in the balanced line. If the A and B outputs balance exactly with equal, opposite currents, the two currents in the ground wire cancel each other out and the wire carries no current at all. In real life, components don't balance perfectly; one driver will be a

little stronger and one receiver will have a slightly larger input impedance.

Without a common ground, the circuit may work, but the energy from the imbalance has to go somewhere and may dissipate as electromagnetic radiation.

The RS-485 specification recommends connecting a 100- Ω resistor of at least 0.5 W in series between each node's signal ground and the network's ground wire, as Figure 1 shows. This way, if the ground potentials of two nodes vary, the resistors limit the current in the ground wire.

SIMPLIFIED LOW-POWER LINK

Adding terminations increases a link's power consumption. With two parallel 120- Ω terminations and a differential output of 1.5 V, the current through the combined terminations is 25 mA (disregarding the effects of biasing, attenuation, etc.).

Without terminations, the load is the parallel combination of the receivers' input impedances and varies with the number of receivers. The maximum 32-unit loads have a combined parallel impedance of 375 Ω to ground or V+.

For some shorter and slower links, you can save power and components by not using terminating and biasing components. This option is feasible if the line is electrically short, which means it behaves as a lumped, rather than distributed, system. On a short line, the reflections die out long before the receiver is ready to read the signal.

A general guideline is that a line is short if the rise time of its signals is greater than four times the signals' one-way delay. The one-way delay is the amount of time needed for a signal to travel from the driver to the receiver.

It's a function of the line's physical length and the speed of signals in the line. In copper wire, a typical speed is two-thirds the speed of light, which works out to 8 in./ns. Cable manufacturers often specify a value for products likely to be used in network wiring.

The rise time is specified in the driver's datasheet. The slowest chip I've found is Maxim's MAX3080, with a minimum rise time of 667 ns. With cables of up to 100', the rise time is greater than four times the one-way delay (4×150 ns), so the line behaves as a short line and doesn't need terminating or biasing. Another advantage is that the internal biasing pulls idle lines to nearly V+ and ground, so you get greater noise immunity.

The downside to using this chip is that the slow rise time means that it's rated for use only at 115,200 bps or less.

SHORT-CIRCUIT PROTECTION

The previous circuits ensured that the line was in a predictable state when idle or open. The circuit in Figure 2 also protects the network as much as possible if the signal lines are shorted. Instead of a single pair of biasing resistors for the entire line, the circuit has four biasing resistors at each node.

The circuit uses Texas Instruments 75ALS180B transceivers, which have full-duplex RS-485 inputs and outputs. The separate transmit and receive pairs enable the receiver to have its own series biasing resistors. The two RS-485 lines connect just beyond the biasing circuits.

If the signal lines short together, the 1.8-k Ω series resistors in combination with the 36-k Ω biasing resistors hold input A more positive than B. Of course, the node can't communicate with the network if the line is open or shorted, but at least it remains in an idle state (with no false start bits) until the problem is fixed.

Another way to accomplish the same thing is to use transceivers with built-in fail-safe protection for open and short circuits. Chips that have this feature take varying approaches.

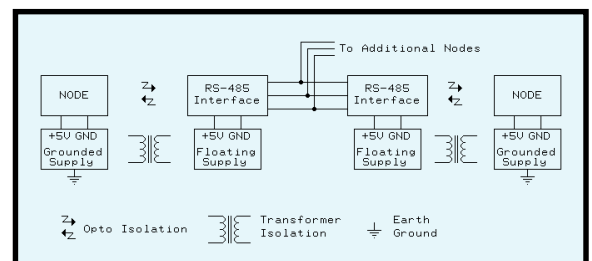


Figure 3—A galvanically isolated link has no ohmic connection to earth ground or to the other circuits the network connects to.

Linear Technologies' LTC1482 has a carrier-detect function that brings the receiver's output high when the differential input voltage is too small to be a valid logic level. The chip has a carrier-detect output that indicates when the line is in an invalid state. National Semiconductor's DS36276 has internal circuits that bring the receiver's output high if the line is shorted or open.

Maxim's MAX3080-89 series provide short-circuit biasing by redefining the threshold for logic 0. Instead of specifying all differential inputs of less than 200 mV as undefined, these chips define a differential voltage of -50 mV or greater as a logic 0.

Voltages equal to or more negative than -200 mV remain defined as logic 1s. The only undefined region is from -50 to -200 mV. With these definitions, a shorted line (which results in a differential input of 0 V) is a logic 0, which results in a high output at the receiver.

ISOLATED LINK

The entire RS-485 network has to share a ground, but the network can be galvanically isolated from other circuits the network connects to as well as from earth ground.

All RS-485 components must be able to operate with common-mode voltages between -7 V and +12 V. Some components have higher ratings. The common-mode voltage at the receiver equals half the sum of the two signal voltages, referenced to the receiver's signal ground. The voltage varies with the differential signal voltages, the difference in ground potentials between the driver and receiver, and noise on the line.

Where the ground connection is long, isolating the ground can ensure that the components don't exceed their ratings. Isolation also protects the circuits the network connects to if the network circuits are damaged by high voltage.

Complete isolation requires isolating the power supplies and the network's signals. The power supplies typically use transformer isolation, whereas the signals use optoisolators (see Figure 3).

A one-chip way to achieve isolation is to use Maxim's MAX1480, which contains its own transformer-isolated supply and optoisolated signal path.

Figure 4—This circuit's automatic driver-enable ensures that the previous driver is disabled by the time the next node begins to transmit. The driver-enable line follows the data with a short delay before disabling the driver. Biasing circuits hold the line in the correct state when the driver is disabled.

AUTO-SWITCHING LINK

One of challenges in designing an RS-485 link is controlling the driver-enable lines. Because all of the nodes share a data path, only one driver can be enabled at a time. Before transmitting, a driver must be sure that the previous driver has been disabled.

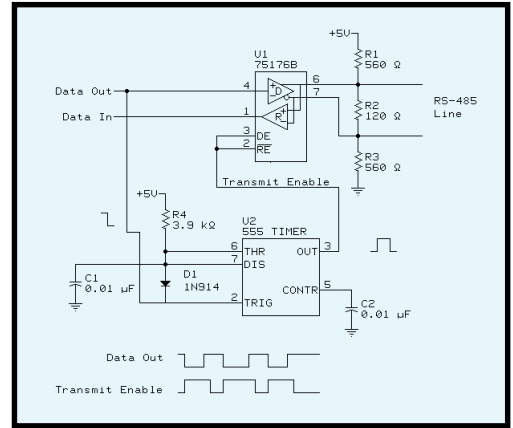
Many RS-485 networks use a command/response protocol; one node sends commands and the node being addressed returns a response. The UART in the node being addressed detects the final stop bit in the middle of the bit width, or slightly sooner or later if the sender's clock doesn't match exactly.

A very fast node may be ready to send a reply within a few microseconds after detecting the stop bit. To prevent the need for a delay before responding, the sending node's driver should be disabled as soon as possible after the leading edge of its final stop bit.

In most systems, the transmitting driver is enabled on the leading edge of the start bit and remains enabled for the entire transmission. It is disabled as soon as possible after the final stop bit. In the delays between transmissions, biasing holds the line in an idle state.

There are various ways that the transmitting node can determine when a transmission has finished and it is safe to disable the driver. The node may read back what it sent, or it may use a hardware or software timer to estimate the time needed to transmit.

Figure 4 shows a completely automatic way to control the enable line so the driver is disabled as quickly as possible, soon after the leading edge of the stop bit. With this circuit, the program code doesn't have to toggle a signal to enable and disable the driver, and a transmitting driver doesn't need to allow extra time to be sure that the previous driver has been disabled.



Unlike other methods of automatic control, there are no jumpers to set for a particular bit rate. I learned of this method when I saw it in R.E. Smith's IRSFC24 Isolated RS-485 board.

Instead of keeping the transmitter enabled for the entire transmission, the circuit in Figure 4 enables the driver on the leading edge of the start bit or any logic low at the driver's input. It also disables the driver ~40 μs after the leading edge of the stop bit or any logic high at the driver's input. When the driver is disabled, biasing resistors ensure the receiver's output is a logic high.

The delay is generated by a 555 timer configured as a monostable (one shot). The enable inputs of the driver and receiver are tied together so the receiver is disabled when the driver transmits.

The timer's output controls the transceiver's enable inputs. A falling edge at Data Out indicates a start bit and triggers the timer. The timer's output goes high, enabling the driver and bringing line B more positive than line A. Diode feedback to the Trig input holds the timer's output high for as long as Trig remains low.

When Data Out goes high, the RS-485 line switches, bringing line A more positive than line B. The same logic high also causes the timer to begin timing out. About 40 μs after the rising edge, the timer's output goes low, disabling the driver.

The delay ensures that the driver's RS-485 output switches without delay, while the driver is enabled. When the driver is disabled, the biasing components continue to hold A more positive than B.

Similarly, any falling edges in the transmitted data enable the driver and

any rising edges disable the driver after the delay. On the final stop bit, the driver is disabled no later than 40 μ s after the stop bit's leading edge.

At rates of 9600 bps or less, the bit width is greater than 100 μ s, which means the driver is disabled at around the middle of the bit width. At faster bit rates, the driver will still be disabled no more than 40 μ s after the stop bit's leading edge. For networks needing very fast response time at faster bit rates, decrease R4 for a shorter delay.

A downside is that the final voltage for logic zeros is the biasing voltage, which is usually less than the differential voltage when the driver is enabled. But because the biasing voltage needs to be great enough to prevent errors from noise on an idle line, it should do the job for active logic states as well. \square

Jan Axelson has been involved with computers and electronics for over 20 years. Her books include Serial Port Complete, Parallel Port Complete, and The Microcontroller Idea Book. You may reach her at jan@lvr.com or on the web at www.lvr.com.

REFERENCE

- J. Axelson, *Serial Port Complete: Programming and Circuits for RS-232 and RS-485 Links and Networks*, Lakeview Research, Madison, WI, 1998.

SOURCES

RS-485 transceivers

Linear Technology
(408) 432-1900
Fax: (408) 434-0507
www.linear-tech.com

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

National Semiconductor
(800) 272-9959
(408) 721-5000
Fax: (408) 739-9803
www.national.com

Texas Instruments, Inc.
(800) 477-8924, x4500
(972) 995-2011
Fax: (972) 995-4360
www.ti.com

TIA/EIA-485-A

Global Engineering Documents
(800) 854-7179
(303) 397-7956
Fax: (303) 397-2740
www.global.ihs.com/sitemap.html

USB-to-RS-485 converter

Inside Out Networks
(512) 301-7080
Fax: (512) 301-7060
www.ionetworks.com

B&B Electronics Manufacturing Co.
(815) 433-5100
Fax: (815) 434-7094
www.bb-elec.com

COMM-DRV serial-port drivers (with RS-485 support)

WCSC
(800) 966-4832
(281) 360-4232
Fax: (281) 360-3231
www.wcscnet.com

RS-485 interface with automatic enable control

R.E. Smith
(513) 874-4796
Fax: (513) 874-1236
www.rs485.com

FEATURE ARTICLE

Paul Breed

A Web-Based Chart Recorder

From needing a forklift to fitting in your pocket, chart recorders have undergone some big changes. Paul's recorder has something even newer—it accesses up-to-the-minute information using an embedded web server.



Sixteen years ago I was finishing my engineering degree and working part time at JPL.

The engineer that I was working with wanted to record a number of temperatures as we tested a newly repaired microwave instrument. I was instructed to go the equipment depot and get an 8-channel strip-chart recorder.

The only available recorder was a 24-channel unit that was 3' wide and 4' tall, and it weighed far more than I could possibly carry. It was delivered to our building with a forklift.

Over the next few weeks, we ran a forest of paper through that machine, which had the maddening habit of clogging the pen just when the signal was getting interesting. Today, we

can do the same job with equipment that fits in our pockets.

Recently, I needed to record some temperatures for an engineering project so I created a virtual strip-chart recorder using a Netburner embedded web server and an RS-232 DVM. Although this is overkill for the application, it illustrates the concepts behind dynamic graphics on a web server.

When I set out to do this project, my goals were to:

- create a virtual strip-chart recorder to record temperatures
- enable the user to scroll and zoom on the chart
- enable the user to change the recording parameters
- do all of this using a standard web browser

Most of these requirements were straightforward, but displaying the chart was the hard part. There were two possible solutions. I could either embed a Java applet in an HTML page and have the Java applet scroll and zoom on the chart, or I could have the embedded web server generate embedded images on-the-fly.

Although Java may be up and coming, it's difficult to save images to disk or to print from an embedded Java applet. I chose to generate embedded images on-the-fly.

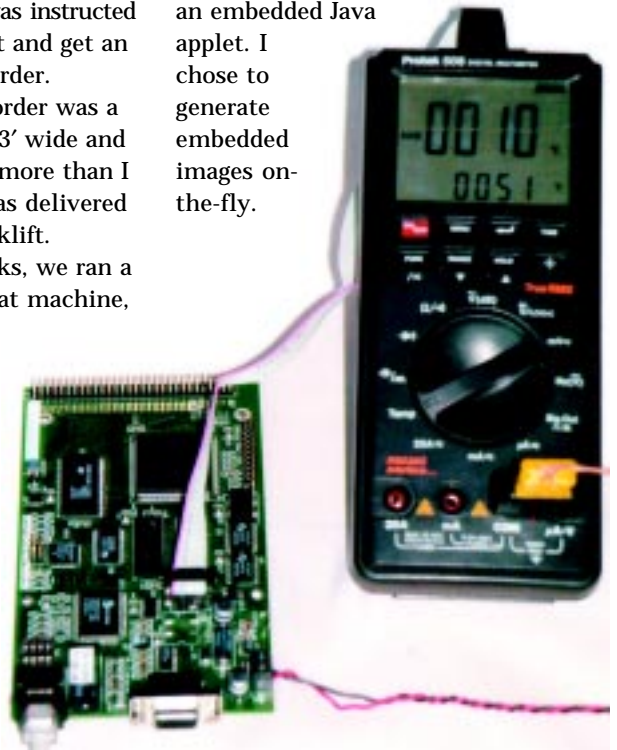


Photo 1— A Netburner embedded web server and an RS-232 DVM make a fully functional chart recorder.

SYSTEM HARDWARE

The data-gathering front end for this project was a Protek 506 DMM that offers a number of measurement capabilities and an almost RS-232 interface. The brains and web server were implemented on a Netburner development board.

I originally thought that I could just plug the two units together and be done with the hardware design. It is never that simple.

The Protek DMM has an RS-232 interface that doesn't directly generate RS-232 levels. In its normal application attached to a PC, it seems to steal voltage from the PC's hardware-flow control lines.

Because the Netburner board only implements the RX and TX lines, no such voltage was available. The quick fix of tying +5 V to pins 6 and 7 of the DVM quickly solved this problem [1].

After the hardware shown in Photo 1 was complete, the rest was a simple matter of software. But, before we can discuss the software, let me provide some web background.

WEB BROWSER 101

When a browser is given a URL, it first breaks down the URL into its component parts [2]. The protocol specifies how the browser should get the requested document, the server part specifies where it should get the document, and the document part specifies what document to get.

The browser sends a request (usually HTTP) to the server asking for the document [3]. After the browser has the document, it looks inside it to see if it contains embedded images or applets. It then generates requests for each of these embedded elements.

When it has all of the document and the embedded pieces, it renders the result onscreen. So, if an HTML document has some text and two embedded pictures, it requires three HTTP transactions to properly render the document: one request for the base document and a separate request for each of the embedded pictures.

DEVELOPMENT ENVIRONMENT

The beauty of this project is the lack of PC software. After the applica-

tion is downloaded into the embedded system's flash memory, the only PC software that is necessary is a standard web browser.

The embedded application was developed using the GNU/EGCS ColdFire cross-compiler running on a Windows workstation [4]. The standard Netburner web server and RTOS were used as is.

Some of my code samples are specific to the Netburner environment, but the concepts are the same for any embedded TCP/web server product.

Using the Netburner environment, the development steps are:

- copy one of the development examples into a new directory
- add the desired HTML pages
- add the desired program functionality
- compile and compress the code and HTML files into a flash image
- download the flash image to the ColdFire board using the Ethernet link
- restart the ColdFire board
- test the result
- repeat steps 2-7 until done

Listing 1—This code gathers samples from the DVM and stores them in a circular buffer. It also detects errors in the DVM connectivity and settings.

```
while (1)
{
    char buffer[40]; /* buffer to accumulate DMM chars in */
    DWORD now;
    int n,cnt;
    /* Set up interval to wait for next sample */
    next_read+=gSampleInterval;
    now=TimeTick;
    if (now<next_read)
        OSTimeDly(next_read-now);
    /* An RTOS function that yields for some number of ticks */
    else
        next_read=TimeTick;
    write(fddvm,"\r",1); /* Ask DVM for a new sample */
    cnt=0;
    /* Read data from DVM */
    while(1)
    {
        n=ReadWithTimeout(fddvm,buffer+cnt,40-cnt,TICKS_PER_SECOND);
        if (n<=0)
        { /* DVM is probably off */
            next_read=TimeTick+gSampleInterval;
            DVMState=DvmDead;
            break;
        }
        else
        {
            cnt+=n;
            buffer[cnt]=0;
            if (buffer[cnt-1]=='\r')
            {
                if ((strcmp(buffer,"TEMP",4)==0))
                { /* Last DVM reading was a Temperature */
                    int t;
                    t= atoi(buffer+5);
                    DVMState=DvmOk;
                    StoreSample(t);
                }
                else
                {
                    DVMState=DvmSetWrong;
                }
                break;
            }
        }
    }
} /* While reading */
} /* While forever */
```

Table 1—A URL like `http://10.1.1.95/INDEX.HTM?S0000000AFFFFFFF1FFFF23B000000000` has a number of data fields.

Field	Meaning
<code>http://</code>	URL protocol field
<code>10.1.1.95</code>	IP address of the embedded server
<code>INDEX.HTM?</code>	Root name of the document
<code>S</code>	Size of the requested graph ('T'iny, 'S'mall, 'M'edium, 'L'arge)
<code>0000000A</code>	Top or maximum of the graph (10 in this case)
<code>FFFFFFF1</code>	Bottom of the graph (−15 in this case)
<code>FFFF23B0</code>	Starting edge of the graph (−15 h 40 min. in this case)
<code>00000000</code>	Ending edge of the graph

After the code is deployed, it's also possible to update the running code over the network without having any physical access to the unit. I believe that good, rapid development is an iterative process and the speed of your tools is important.

Using the EGCS/GNU tools and Ethernet for the flash download, the entire recompile (compress and download process) takes about 30 s (on a P-II 450).

SOFTWARE FUNCTIONALITY

The system software only has to perform two functions—gathering and storing the data, and presenting the data as a web page.

The data gathering is done with the simple task code shown in Listing 1. It waits 1 s, sends a read request to the DVM, and stores the response in a large circular buffer. The values are stored as simple integers.

Most of the software work went into the user interface and display code. The user interface consists of the web page shown in Photo 2. The page is an HTML form with an embedded image.

This form allows the user to set the extent of the graph and control the size of the embedded image. The form

was originally laid out and tested as an HTML file on the PC.

Because multiple users may want to display different views of the same data, I wanted to store display information for each user. The most logical way to do this is to store the page settings in the URL used to access the web page. In an embedded environment, there's nothing special about the document section of the URL. You're free to encode whatever you want in that name.

When the web page is first accessed, it assigns default values for the display limits and graph sizes. When a user changes these settings, the choices are encoded in the new URL. Table 1 shows the URL used in this application and how it is encoded.

All of the numerical values are encoded in eight hex digits and all times are in seconds. Using this scheme to store settings in the URL enables multiple users to save different display formats by bookmarking the page.

Photo 2 shows a plot of temperature outside my house in New Hampshire. Putting a simple web page on the embedded server is trivial; this page was a little more complex.

Listing 2—These data-rendering function calls are selected from comments in the HTML code. They are used to render dynamic values in the web page.

```
void DoMaxD(int sock, PCSTR url); /* Maximum display value */
void DoMinD(int sock, PCSTR url); /* Minimum display value */
void DoMaxT(int sock, PCSTR url); /* Right end of the graph */
void DoMinT(int sock, PCSTR url); /* Left end of the graph */
void DoSize(int sock, PCSTR url); /* Graph size selection combo */
void DoShowStatus(int sock, PCSTR url);
/* The status table on bottom */
void GetImageKey(int sock, PCSTR url);
/* The gif image URL */
```

Variable name	Meaning
MaxD	Maximum displayed temperature
MinD	Minimum displayed temperature
MaxT	Maximum displayed time
MinT	Minimum displayed time
-gSize	Size combo box
-submitv	Name of button used to submit the form (Set, Show All, Show Last hour)

Table 2—The variables encoded in the form response are used to control the display.

This page has to parse the URL and fill in the proper values in the form elements, reference the proper embedded image to display the graph that the user wants, allow changes to the settings, and render the embedded GIF.

Filling in the proper values is quite straightforward. The Netburner system enables you to embed a specially formatted comment in the HTML page wherever you want to embed dynamic content. A dynamic snippet of this HTML code is:

```
<INPUT TYPE="text"
NAME="MaxD" VALUE="<!--
FUNCTIONCALL DoMaxD -->"
SIZE=5>
```

Whenever this code is encountered in a web page, it causes the web server to call the C function DoMaxD. The

actual web page used in this example has seven of these function calls, shown in Listing 2.

DoMaxD has to fill in the current value of the maximum display on the chart, so it calls a function to extract the GraphData and SizeData structures from the URL. Then it's just a simple matter of displaying the proper value (shown in Listing 3) and referencing the proper embedded image to display the graph the user wants. To display the page, the system downloads the HTML page and the embedded image.

Both the page and the graph need to know the value of the user's display settings (encoded in the URL). So, the seventh function in the list, GetImageKey (see Listing 4), is a little different. It takes the URL value used to render the HTML page and adds this URL to

Listing 3—The DoMaxD function is called when the HTML code contains a comment of the form <!--FUNCTIONCALL DoMaxD-->. This mechanism is used to put dynamic content into an otherwise static web page.

```
void DoMaxD(int sock, PCSTR url)
{
    GraphData gd;
    SizeData sd;
    GetData(&gd,&sd,url);
    char buffer[20];
    sprintf(buffer,"%d",gd.MaxD);
    writestring(sock,buffer);
}
```

Listing 4—The GetImageKey function renders the setting components of the URL (the part past the "?") to the page being displayed. It passes the URL setting to the HTML page so that it can be sent as part of the GIF image request that draws the graph.

```
void GetImageKey(int sock, PCSTR url)
{
    const char * cp= url;
    while ((*cp) && (*cp!='?')) cp++;
    if (*cp) writestring(sock,cp);
}
```

the end of the image URL so that the GIF image rendered on the form is the proper image.

The HTML image specification for the graph is `<IMG SRC ="IMAGE.GIF <!--FUNCTIONCALL GetImageKey -->" >`. When the user wants to change any of the settings, they press one of the buttons, which causes the HTML browser to generate a post request. This request passes the values of all the inputs on the form to

the web server. This form encodes the variables shown in Table 2 and sends them to the web server.

This process is done in the DoGraphPost function. After processing all of the form elements, DoGraphPost constructs a new URL that represents the user's selections and redirects the browser to reload that URL. This code is shown in Listing 5.

The final step in the display processing is the generation of the GIF

Listing 5—This code processes the variables sent in the POST request from the web browser. It extracts the individual control variables and uses them to generate a new URL that reflects the new settings.

```
int DoGraphPost(int sock, char * url, char * pData, char *
                rxBuffer)
{
    GraphData gd;
    gd.MaxD=100;
    gd.MinD=-10;
    gd.MaxT=0;
    gd.MinT=-24*3600;
    char buffer[80];
    char * cp;
    char siz;

    /* Extract basic variable */
    ExtractIntPostData("MaxD",pData,gd.MaxD);
    ExtractIntPostData("MinD",pData,gd.MinD);
    ExtractTimePostData("MaxT",pData,gd.MaxT);
    ExtractTimePostData("MinT",pData,gd.MinT);
    ExtractPostData("gSize",pData,buffer,80);
    siz=buffer[0];

    /* Extract button pressed to submit form */
    ExtractPostData("submittv",pData,buffer,80);
    cp=buffer;
    while ((*cp) && (isspace(*cp)))cp++;
    /* Values for buffer are set Show Last Show All */
    if (buffer[1]!='e')
    {
        if (buffer[5]=='L')
        /* Show Last Hour */
        gd.MaxT=0;
        gd.MinT=-3600;
        }
    else
    /*Show All */
    gd.MaxT=0;
    if (gNextPosition> gMaxStored) gd.MinT=gMaxStored;
    else
    gd.MinT=gNextPosition;
    gd.MinT*=gSampleInterval/TICKS_PER_SECOND;
    if (gd.MinT<60) gd.MinT=60;
    gd.MinT=gd.MinT*-1;
    }
}

/*Send requester back to proper web page */
sprintf(buffer,"INDEX.HTM?%c%08X%08X%08X%08X",siz,gd.MaxD,gd.MinD,
        gd.MinT,gd.MaxT);
RedirectResponse(sock,buffer);
}
```

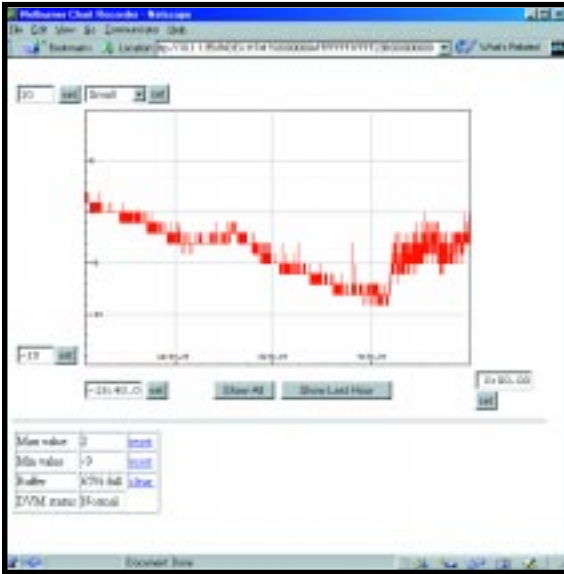


Photo 2—Here's a web-browser view of the web chart recorder. This particular graph shows a plot of the temperature outside my house in New Hampshire.

image that displays the data. When the browser loads the main page and its embedded IMG tag, the browser parses this IMG tag and generates a separate request to get the GIF file. The server parses the URL and creates a GIF image that is sent to the user's browser over the network connection.

packages available that programmatically render GIF images, but most of them have some problems for embedded work. I chose to adapt the library GD because source was readily available, and it contained a run-length compressor that avoids the Unisys patent [5].

Before I can discuss the graph drawing-code, I need to tell you about building a GIF-generation library.

There are a number of

Listing 6—Here is the interface for the GIF-generation class. This class encapsulates a small subset of the public-domain GD graphics library.

```

class DrawImageObject
{
  BYTE * m_pImageBuffer;
  BYTE * m_pColorArray;

  int m_xSize;
  int m_ySize;
  int m_nColors;
  int m_curx;
  int m_cury;

private:
  int GIFNextPixel();
  void compress(int init_bits, int fd);
public:
  DrawImageObject(int xsize, int ysiz, int ncolors);
  ~DrawImageObject();
  void PutPixel(int x, int y, BYTE color);
  BYTE GetPixel(int x, int y);
  void SetColor(BYTE index, BYTE red, BYTE green, BYTE blue);
  void Line(int x1, int y1, int x2, int y2, BYTE colorindex);
  void Box(int x1, int y1, int x2, int y2, BYTE colorindex);
  void FilledBox(int x1, int y1, int x2, int y2, BYTE fillc,
    BYTE outlinec);
  void Text(const char * pText, int x1, int x2, const char *
    fontrecord, BYTE color);
  int TextXsize(const char * pText, const char * fontrecord);
  int TextYsize(const char * pText, const char * fontrecord);
  void WriteGIF(int fd); /* Writes GIF to file descriptor */
};

```


Listing 7—The *MyDoGet* code looks at the document URL and performs different actions depending on the document name part of the URL. If it is *IMAGE.GIF*, the request is sent to the image-rendering system. If it contains *RESET*, it processes one of several actions and redirects the browser to a new web page.

```
int MyDoGet(int sock, PSTR url, PSTR rxBuffer)
{
    if (strlen(url))
    {
        if(httpstricmp(url,"IMAGE.GIF"))
        { /* Browser is asking for GIF image */
            GraphData gd;
            SizeData sd;
            GetData(gd,sd,url);
            SendGifHeader(sock);
            DrawGraph(sock,sd.xsize,sd.ysize,gd.MinD,gd.MaxD,gd.MinT,gd.MaxT);
            return 1;
        }
        else
        if(httpstricmp(url,"RESET"))
        { /* Browser is asking us to reset limit values */
            char buffer[80];
            char * cp=url;
            while ((*cp) &&( *cp!='?')) cp++;
            sprintf(buffer,"INDEX.HTM%s",cp);
            /* Decide what kind of reset request it was */
            if(httpstricmp(url,"RESETMAX.HTM"))
            {
                gMaxMeasured=-9999;
            }
            else
            if(httpstricmp(url,"RESETMIN.HTM"))
            {
                gMinMeasured=9999;
            }
            else
            if(httpstricmp(url,"RESETBUF.HTM"))
            {
                gNextPosition=0;
            }
            RedirectResponse(sock,buffer);
            return 1;
        }
    }
    return (* oldhand)(sock,url,rxBuffer); /*Use old handler */
}
```

Here's a brief update on the patent issue. The normal GIF image is compressed using the LZW compression algorithm. Many years after Compu-Serve promoted the GIF format as the de facto graphics exchange format, Unisys figured out that they held the patent on the underlying compression. As a result, people are searching for ways to circumvent this patent.

The technically correct solution is to use the new patent-free graphics standard, PNG. Unfortunately, not all older browsers support it. The additional problem is that it has a much bigger code impact in an embedded environment.

I modified the GD system in several steps. First, I removed everything

except the GIF compressor and then encapsulated the global statics in a C++ class.

Next, I added some general-purpose line-drawing routines from my toolbox. I then compressed the large GD fonts and wrote some font-rendering routines to use the new format.

The result is a simple GIF generation class with the public interface (see Listing 6). This class is simple, but it was sufficient to create the graphs for this project.

GRAPH-DRAWING CODE

When the web server gets a request for a URL, it usually tries to satisfy that request from its compressed store of HTML files and images. In the case

of the GIF image, there is no compressed store. The image is entirely generated programmatically.

Besides capturing the GIF request, I also need to capture the reset and clear requests from the data summary box at the bottom of the displayed page. This process is done in the function MyDoGet, shown in Listing 7.

Once the GET that is redirecting code recognizes the request as a request

for IMAGE.GIF, it sends a GIF header, and it then calls DrawGraph:

```
di.WriteGIF(sock);  
return 0;
```

DrawGraph performs four functions. It initializes the image and color map, draws and labels the x axis, draws the data, and then draws and labels the y axis.

Listing 8—This code draws the tick marks and numbers on the x-axis. It adjusts the spacing of the numbers to fit on the displayed scale.

```
yh=di.TextYsize("T",TinyFont); /* Draw x-axis scale and text */  
mark_inc= maxg-yaxis.map_2_to_1((5*yh));  
if (mark_inc <=1)  
{  
    mark_inc=1;  
    semi_inc=0;  
}  
else  
if (mark_inc <=5)  
{  
    mark_inc=5;  
    semi_inc=1;  
}  
else  
if (mark_inc <=10)  
{  
    mark_inc=10;  
    semi_inc=2;  
}  
else  
{  
    mark_inc=25;  
    semi_inc=5;  
}  
  
start_pos=((((maxg+ming)/2)/mark_inc)*mark_inc);  
/* Move from start position to beginning of drawable area */  
while (start_pos > ming) start_pos-=mark_inc;  
/* Draw large ticks and text labels*/  
for (i=start_pos; i<maxg; i+=mark_inc)  
{  
    int y=yaxis.map_1_to_2(i);  
    if ((y<GYSIZE) && (y>0))  
    {  
        di.Line(0,y,0+BIGTICK,y,4);  
        di.Line(0+BIGTICK,y,GXSIZE-1,y,5);  
    }  
    if ((y<GYSIZE-yh) && (y>yh))  
    {  
        sprintf(buffer,"%d",i);  
        di.Text(buffer,0+BIGTICK,y-(yh/2),TinyFont,4);  
    }  
}  
if (semi_inc) /* Draw small ticks */  
{  
    for (i=start_pos; i<maxg; i+=semi_inc)  
    {  
        int y=yaxis.map_1_to_2(i);  
        if ((y<GYSIZE) && (y>0))di.Line(0,y,0+SMTICK,y,4);  
    }  
}
```

In the graph initialization, two `transmap` objects—`xaxis` and `yaxis`—are created, which permit easy mapping from the natural units on time and temperature to the graph dimensions of pixels.

After the graph object is initialized, the program draws the `x` axis by measuring the size of the display text and figuring out how close together it can place the `x`-axis tick marks (see Listing 8). The data is drawn (see Listing 9) and sent out to the web browser in Listing 10.

FUTURES

In its current implementation, the graphics class is a RAM hog. It creates an uncompressed image buffer before it compresses the image. It's possible to write a GIF generator that doesn't buffer the whole image prior to compressing it, but this arrangement is more complex and error prone. Because 4 MB of embedded DRAM costs less than \$8, it wasn't worth the effort.

Once embedded developers become comfortable using the Internet technologies and capabilities in their envi-

ronment, the possibilities are endless. You might set limits and send e-mail when the limits are exceeded, or even have the system dial up an ISP via modem and send an e-mail message with an attached GIF image showing the problem. ☐

Paul Breed is currently chief architect at Netburner. He has been designing and building embedded widgets of all sorts since 1982. You may reach him at paul@netburner.com.

Listing 9—This code draws the temperature data in the graph. It uses the temperature data stored in the circular buffer `gMeasurements`. It also uses two-axis `map` objects to scale the data properly.

```
if (gNextPosition!=0) /* Draw actual data */
{
    int sp,ep;          /* sp=starting point ep=ending point*/
    int lastx;
    int lasty;
    int gNow;
    gNow=gNextPosition-1;
    if (mint < -gNow)
        sp=-gNow;
    else
        sp=mint;
    ep=maxt;
    lasty=yaxis.map_1_to_2(gMeasurements[(gNow+sp)%gMaxStored]);
    lastx=xaxis.map_1_to_2(sp);
    for (i=sp+1; i<ep; i++)
    {
        int y=yaxis.map_1_to_2(gMeasurements[(gNow+i)%gMaxStored]);
        int x=xaxis.map_1_to_2(i);
        di.Line(lastx,lasty,x,y,1);
        lastx=x;
        lasty=y;
    }
}
```

Listing 10—This code sets up the graphic response object, sets some default colors, and draws an outside border in preparation for drawing the actual graph data.

```
int DrawGraph(int sock, int GXSIZE, int GYSIZE, int ming, int
    maxg, int mint, int maxt)
{
    int xw,yh,mark_inc,semi_inc,start_pos;
    char buffer[20];
    int i;

    transmap yaxis(ming,maxg,GYSIZE,0); /* Initialize drawing system */
    transmap xaxis(mint,maxt,0,GXSIZE);
    DrawImageObject di(GXSIZE ,GYSIZE,256);
    /* Draw box around whole image */
    di.SetColor(0,255,255,255);
    di.SetColor(1,255,0,0);
    di.SetColor(2,0,255,0);
    di.SetColor(3,0,0,255);
    di.SetColor(4,0,0,0);
    di.SetColor(5,192,192,192);
}
```

SOFTWARE

Two sets of source code are available via the Circuit Cellar web site for this project. `NBGifGraph.zip` is a complete build project for the Netburner environment. There's also a sample of the stripped-down GIF compressor example program that runs on a PC in any other embedded environment.

REFERENCES

- [1] Hung Chang Products, *Protek 506 Digital Multi Meter User's Manual*, Seoul.
- [2] T. Berners-Lee, RFC 1738 Uniform Resource Locator, University of Minnesota, December, 1994.
- [3] T. Berners-Lee, R. Fielding, and H. Frystyk, RFC 1945 Hypertext Transfer Protocol—HTTP/1.0, UC Irvine and MIT, May, 1996.
- [4] EGCS, www.cygnus.com/prebuilt/binary/images/www.calm.hw.ac.uk/davidf/coldfire.htm
- [5] corp2.unisys.com/LeadStory/lzwfaq.html

SOURCES

Development kit
Netburner
 (619) 530-0293
 Fax: (619) 530-0240
www.netburner.com

Protek 506 DVM
 Hung Chang Products Co.,Ltd
 +02 395-8611-20

Active Electronics
 (781) 932-0050

EMBEDDED COMPUTER PLATFORM

The **Instant Instrument** is a complete PC-based instrument platform designed for specialty applications that use embedded computers, including test and measurement, data acquisition, communications, and medical electronics. The unit includes the complete enclosure, display, power supply, battery, and computer subsystem with additional space and slots for custom circuitry. The designer creates only the application software (Windows, DOS, Unix) and provides any specific hardware required.

The Instant Instrument is modular and comprises the DIPU (display, input, processing unit) and optional ASCU (application-specific circuitry unit) modules. The DIPU contains a five-slot PCI/ISA bus, a scalable PCI/ISA SBC starting at a 133-MHz '586 with 32-MB DRAM, a PC/104 bus connector, a 10.4" TFT VGA display with optional touch capability, an internal disk drive, and the standard PC interface ports. The optional ASCU module is both mechanically and electrically attached to the DIPU and provides an addi-



tional three ISA-bus connectors for oversize (up to 10.5" x 12") circuit boards or other components such as disk drives, pumps, valves, actuators, or other electronic peripherals.

Additionally, the DIPU module can be attached directly to the user's existing equipment enclosure to instantly provide a command and control module for the whole system and enable network or Internet connectivity for any device or machine.

The Instant Instrument is constructed of cast aluminum and stainless steel. The DIPU measures 10" x 8.8" x 12", and the ASCU 2.7" x 10.75" x 13". It is designed for ruggedness, EMC control, and corrosion resistance. As well, this platform features internal batteries, power supply, and charger circuitry to power the unit for approximately 2 h for portable use or backup in case of power failure.

The Instant Instrument starts at **\$4900** for the base unit without touchscreen.

Instant Instrument
(813) 289-5555
Fax: (813) 289-5454
www.instantinstrument.com

DATA ACQUISITION BOARD

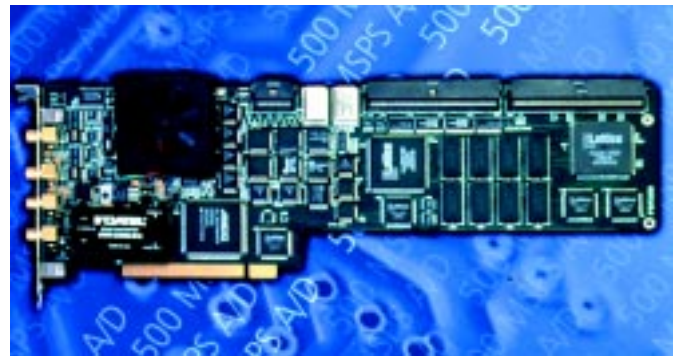
The **PDA500**, from Signatec, is one of the fastest 8-bit PCI A/D boards on the market. It captures waveform frequencies and transfer data at up to 500 MBps, and provides a full 500-MHz bandwidth. Applications include communications, ultrasound, radar, sonar, mass spectroscopy, and spectrometry.

The PDA500 has two input channels and a multiplexer to select the digitization source. A digitally controlled attenuator gives the unit 14 input voltage settings. Up to four PDA500 boards can be connected in a master/slave configuration to achieve synchronized data samples. The PDA500 implements the Signatec Auxiliary Bus (SAB), which permits both block and packet data transfers at a sustained rate of 500 MBps to other processing, playback, and storage devices without any PC intervention.

The PDA500 is available in 1- and 4-MB onboard memory configurations. An excellent software package, which includes device drivers and example applications for DOS and Windows 95/NT, is provided at no additional cost. An extensive library of C-language functions with source code is supplied with the board to ease the development of application software.

The PDA500 is priced from **\$6100 to \$7500**, depending on memory configuration and SAB implementation.

Signatec, Inc.
(909) 734-3001
Fax: (909) 734-4356
www.signatec.com



Nouveau PC

MULTIMEDIA ENHANCED SBC

The **VIPer826** is a half-size, industrial single-board computer for PC/104-Plus, ISA passive back-plane, or stand-alone operation. Its Cyrix architecture provides a highly integrated multimedia (data, video, and sound) capability that delivers Pentium-class performance at a reasonable cost.

The board features a Cyrix MediaGX MMX processor operating at speeds up to 266 MHz, integrated 16-KB Level 1 cache, and up to 128 MB of unbuffered SDRAM memory. Also included is VGA support for simultaneous CRT and TFT flat-panel operation, PC 97-compliant 16-bit audio with AC 97 CODEC assistance, and Ultra DMA/33 IDE support. Also onboard are serial/parallel, USB, floppy, keyboard, and mouse ports. 10/100Base-TX Ethernet connectivity is optional.

The Award Hi-flex BIOS, in boot-block flash memory with emergency recovery code, supports serial/parallel port remapping, keyboard disable, and console redirection. Other features include a PC/104-Plus expansion header, bootable CompactFlash disk support, a CPU temperature monitor, watchdog timer, and power-fail circuit.

The single-unit price of the VIPer826 with a Cyrix 233-MHz MediaGX processor without Ethernet and memory is **\$650**, including a two-year warranty.

Teknor Industrial Computers, Inc.
(800) 354-4223
(450) 437-5682
Fax: (450) 437-8053
www.teknor.com



INTEGRATED VEHICLE COMPUTER

The **PC/Piranha** is a Windows CE computer especially designed for the rigors of use on vehicles. A highly integrated thin-client platform, the PC/Piranha takes advantage of the latest technology and miniaturization of electronics to bundle features found in many devices into a single compact unit. This computer can be used in wide-area wireless and in-premise, local-area wireless applications.

The PC/Piranha features an 80-MHz 32-bit RISC processor with up to 32-MB DRAM and 16-MB flash memory. Three display technologies are available, including a DMTN sunlight-readable LCD display. An analog resistive touchscreen and voice-recognition input capabilities are also included.

Expansion is facilitated through two Type II PC-Cards or one Type II and one CompactFlash card. I/O is available through a 104-key scan keyboard interface, serial ports, an IR port, and audio input/output ports.

The PC/Piranha uses the standard Microsoft Windows CE 2.1 operating system and can run a wide variety of applications. Network options include wireless LAN, wireless WAN, vehicle bus, CAN interface, and GPS satellite navigation.

In single quantities, the PC/Piranha sells for **\$1495**.

Kinetic Computer Corp.
(978) 439-0500
Fax: (978) 439-0501
www.kin.com

SERIAL-TO-TCP/IP CONVERTER SOFTWARE

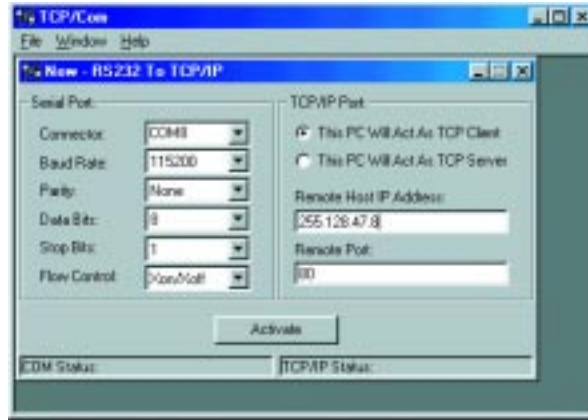
TCP/Com is a software package that enables any RS-232 serial port on a PC to interface directly to a TCP/IP network. Any other computer on the same network is able to access the serial ports on the PC where TCP/Com is running via a TCP/IP socket connection.

For example, a user can connect a serial device (barcode reader, modem, electronic balance, or instrument) to a COM port on a PC, run TCP/Com, and then connect to the device from any other PC on a network through a TCP/IP socket connection. This would allow the use of telnet, TCPWedge, or any other TCP/IP communications software to read or write to the serial device directly from any PC located on the same network. TCP/Com

can also be used to pass serial data across a corporate intranet or over the Internet.

TCP/Com is easy to use. The user selects the serial port and communications parameters for that port, enters an IP address and socket number, and activates the software. TCP/Com features 115-kbps serial communications, 16 simultaneous COM ports, and 16 IP addresses that can be either a client or server.

TCP/Com sells for **\$259** in a two-license package (for use on two PCs simultaneously). A free test version can be downloaded from the company's web site.



TAL Technologies, Inc.
(215) 763-7900
Fax: (215) 763-9711
www.taltech.com

Nouveau NPC

Embedded Ethernet Fundamentals

Ethernet connectivity is desirable in many real-world applications. To get your embedded device connected, listen up as Aaron overviews the hardware and software considerations you'll need to think about.

People have always sought methods to communicate over great distances. Although the technologies have dramatically improved over the last three millennia, identifying the right communication solution for an embedded design is a nontrivial task.

One increasingly popular choice is Ethernet. Ethernet is well worth considering if you encounter one of these major design requirements:

- distance—communicating with another system, whether it's a kilometer or a continent away
- interoperability—communicating with one or more systems based on completely different hardware and software
- connection sharing—sharing data and/or connections with other devices

For example, a medical device might share its information with any other device on a hospital's network,

regardless of what hardware and software the other equipment is based on.

Or, if you're an avid windsurfer like me, you might want a device that measures the current wind speed. Thanks to Ethernet, your high-tech manometer can dynamically generate a web page that lets you (and your fellow windsurfers) view the current wind conditions via the Internet.

Clearly, Ethernet connectivity is desirable in a variety of real-world situations. That's why a lot of people think of Ethernet as the serial port of the '90s.

ETHERNET HARDWARE

Ethernet networks were originally designed and physically arranged in a bus

structure (see Figure 1a). In this configuration, the network devices are placed in series. An Ethernet packet travels past each device until it encounters the termination resistor at the end of the network.

However, most Ethernet installations today use a star architecture like the one in Figure 1b, in which all nodes connect to a central hub. Although the physical structure is different, logically the network is still a bus. The hub's job is to repeat any information received from a single node to all other nodes (electrically speaking, this is the same as being on a single bus).

Although a star configuration may increase the amount of wiring, it offers two advantages. First, a network fault (e.g., a broken wire) in a star network only affects a single node, whereas the same failure in a bus-oriented network may cripple the entire system. Also, adding or removing a node is as easy as changing a single connection.

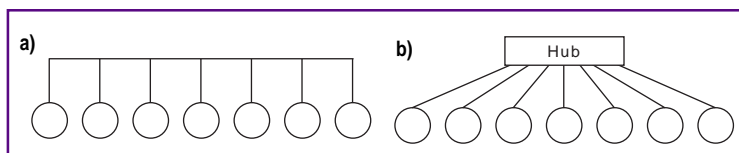


Figure 1a—Ethernet networks wired with coaxial cable typically use a bus architecture. **b**—However, those networks wired with twisted pair are often in conjunction with a star architecture.

Name	Data rate	Wiring	Comments
10Base-2	10 Mbps	Coaxial cable	Limited to <200 m between hub and node
10Base-T	10 Mbps	Twisted-pair wire	Limited to <100 m between hub and node
100Base-T	100 Mbps	Twisted-pair wire	Requires high-quality Category 5 wire

Table 1—Different types of Ethernet networks are referenced by their speed and type of cable. I listed some of the most common implementations here.

Regardless of the physical network structure, all devices are electrically connected to the same bus, so you need to be concerned about collisions. Collisions occur when two devices attempt to transmit on the same bus at the same time.

Ethernet resolves the situation by using a carrier sense multiple access/collision detect (CSMA/CD) scheme. You can think of it as the “cocktail party protocol.”

If a group of people are talking at a cocktail party and two people start to speak at the same time, they both stop and wait a certain amount of time for the other person to begin first (if they’re polite!). If one person’s internal timer decides that enough time has passed and no one else has begun speaking again, that person will take the floor and begin speaking. Ethernet uses the same method to share the network.

Using CSMA/CD has two implications. First, there’s no minimum guaranteed time in which data will be transmitted. As network traffic increases, the number of collisions increases. As collisions increase, the average time it takes to transmit on the network increases.

Ethernet has traditionally been considered suboptimal for applications requiring isochronous data transfers (e.g., real-time voice telephony and video teleconferencing). But as improvements are made in Ethernet controller and infrastructure technology, isochronous communication is rapidly becoming a reality.

The second ramification when using CSMA/CD is that because Ethernet is a shared network, no single device uses the full 10, 100, or 1000 Mbps (known as Ethernet, fast Ethernet, and gigabit Ethernet, respectively). Even if only one device is transmitting on the network, the overhead associated with an Ethernet frame (e.g., destination address, CRC)

prevents the full bandwidth from being available for data transfers. Despite these limitations, even 10-Mbps Ethernet provides more bandwidth than most embedded designs require.

Ethernet controllers generally consist of two components—a media access controller (MAC) and a physical layer controller (Phy). The Phy provides the correct electrical interface; the MAC formats and addresses the data. All of this is outlined in the IEEE 802.3 Ethernet specification.

Most Ethernet and many fast-Ethernet solutions have integrated the MAC and Phy onto a single chip. Few, if any, gigabit-Ethernet solutions have integrated both onto one chip. When selecting an Ethernet controller, make sure you understand whether or not you’re getting the complete solution.

The different Ethernet network types are referenced by their speed and cable. Some common types are given in Table 1.

Although coaxial cable permits farther transmission between network components, twisted pair has become the dominant solution thanks to its smaller size, lower cost, and the ease with which nodes can be added or subtracted from the network.

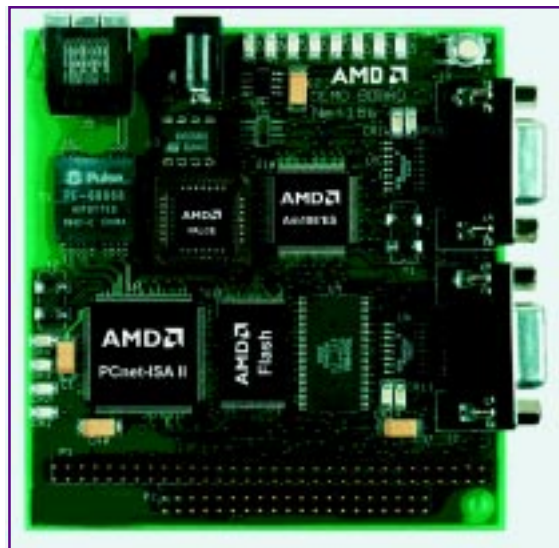


Photo 1—AMD’s Net186 evaluation board implements a full embedded Ethernet design while keeping board space to a minimum.

Layer	Name	Function	Example
7	Application	Application software	Netscape Navigator
6	Presentation	Formatting, encryption, and compression	...
5	Session	Establishes and terminates connections	TCP
4	Transport	Ensures error-free delivery of message	IP
3	Network	Transmission and routing of frames	...
2	Data Link	Frames and addresses packets	Driver software + MAC
1	Physical	Mechanical and electrical interface	Phy

Table 2—The Open System’s Interconnection (OSI) 7-layer model provides a framework for developing communications protocols.

SOFTWARE

Ethernet software is referenced by its position in a standardized, seven-layer model. This model was developed by the International Standards Organization (ISO) and is known as the open systems interconnection (OSI) model (see Table 2).

By following this model, the hardware is abstracted from the software. In theory, you can substitute any Ethernet controller for any other Ethernet controller and leave the upper layers of software untouched (except for replacing the second layer’s driver software, which I discuss later).

But, models are rarely 100% representative of reality. Although most designs implement the various functions described by the OSI model, few real designs strictly enforce the separation of the layers.

The lowest OSI layer, the physical layer, is implemented in hardware by the Ethernet solution selected. It is responsible for the mechanical and electrical interface that transmits the raw bits over the wire.

The remaining six layers are primarily performed in software, which is why I tend to believe that Ethernet hardware design is easy, relative to the software effort involved. Too many engineers add Ethernet hardware to a design and remove it once they see how much software is required!

But don’t despair. The key to success is selecting a hardware vendor whose solution is already supported by software.

The functions of the next layer up, the data link layer, are performed by a driver, which provides instructions that tell the microprocessor how to work with the Ethernet controller. The driver (with help from the MAC) is also responsible for framing and addressing the information.

The next three layers (session, transport, and network) are referred to as the protocol stack and are often purchased from a third-party vendor. The protocol stack is responsible for establishing an error-free virtual connection across the

network. An interesting bit of history is how one of the most widely used protocol stacks originated.

Years ago, the branches of the U.S. armed forces each developed individual computer networks. The Department of Defense quickly realized that in times of war, all these networks would be unable to communicate with each other. So, they funded the development of a scheme that eventually became the transmission control protocol/Internet protocol (TCP/IP).

TCP/IP enables different networks to communicate. This “network of networks” has evolved into the Internet. So if your

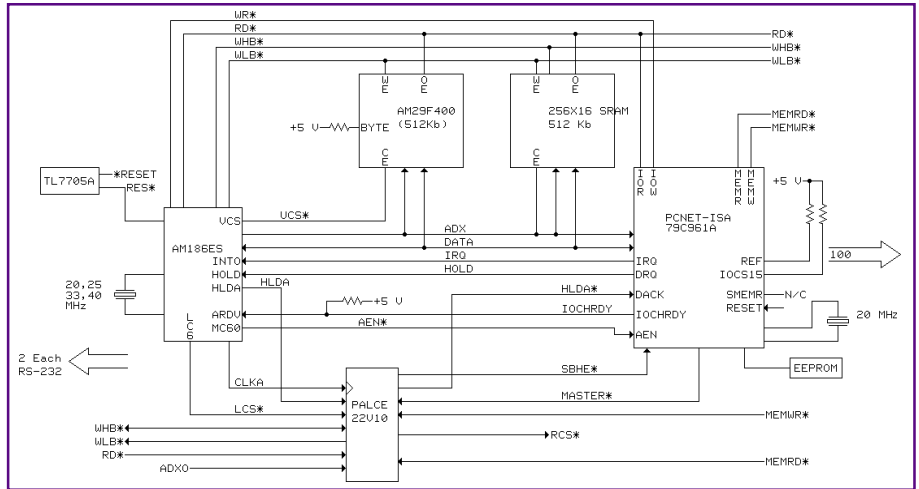


Figure 2—A processor, Ethernet controller, memory, and a modest amount of glue logic is all that's required to implement an embedded web server.

embedded design may someday need to communicate over the Internet, you'd do well to select TCP/IP as your protocol stack.

DESIGN EXAMPLE

To see how you might create an embedded web server, let's consider AMD's Net186. It's a useful starting point for designs in which the embedded device is dealing with information over Ethernet.

Two criteria were used when selecting the microprocessor for this board. First, there needed to be enough performance to run a protocol stack and still have sufficient bandwidth to perform the other tasks required by the system.

Second, most Ethernet controllers have a PC-style bus interface (e.g., PCI, VL, or ISA), so selecting a processor or a system architecture that included a PC-style bus reduced the glue logic required. The Am186ES offers both adequate performance headroom and an ISA-like bus interface.

Because 10-Mbps Ethernet is more than sufficient for most embedded designs, it was used here. To lower wiring costs, twisted-pair wire is supported. The PCnet-ISA II Ethernet controller, which supports 10Base-T Ethernet and offers an ISA bus interface, seemed to be the best fit.

The most interesting aspect of the design was the human interface. We wanted to let users surf the information on the Net-186 board with a standard web browser.

Most designers develop their embedded web applications on top of an RTOS. Here, there must be two components to the software running on top of the RTOS.

The application is responsible for the primary functions of the system, and there

must be some web-serving software to generate the HTML that makes up a web page. The web-serving software also contains a protocol stack (e.g., TCP/IP).

Here's how it all works together. When you sit down at a PC that has a web browser running as its OSI layer 7 application software, the software builds a request asking for information from the embedded web server. This request goes down the client's protocol stack, which begins placing the message in a form ready for transmission on the network. The Ethernet controller puts the message on the network.

When the Ethernet packet gets to the embedded web server, the opposite process happens. The packet arrives at the embedded system's Ethernet controller and is passed up through the OSI layers to the application software.

HERE ARE THE KEYS

Whether it's high bandwidth, long-distance communication, or interoperability with a variety of devices, Ethernet is up to the task. The key to success is selecting a processor or system architecture that provides straightforward connectivity to existing Ethernet controllers as well as sufficient software support to perform the task. [EPC](#)

Aaron Feen handles product marketing in Advanced Micro Devices' embedded processor division. He also worked for five years as an AMD field sales engineer. You may reach him at aaron.feen@amd.com.

SOURCE
Net186, Am186ES, PCnet-ISA II
 Advanced Micro Devices, Inc.
 (408) 732-2400
 Fax: (408) 732-7216
www.amd.com

Astronomical Issues

Part 3: Filters and Undersampling

DSP applications often incorporate FIR filters. This month, Ingo discusses digital filters and FIR techniques, as well as implementing these filters in FPGAs and undersampling—a process used in digital radio design.

As we move into looking at digital filters, I want to focus on one technique you can use to design finite impulse response (FIR) filters, which are used in all kinds of DSP applications. Besides designing FIR filters, I also discuss some aspects of implementing these filters in FPGAs and talk about undersampling, a technique used in digital radio design.

The effects of filters can be best described by looking at what they do in the frequency domain. In other words, look at what kind of frequency response the filter has.

A low-pass filter passes frequencies below a cutoff frequency and blocks higher frequencies. A high-pass filter is the dual of low-pass filters. It blocks frequencies below the cutoff frequency and passes higher frequencies.

Bandpass filters pass frequencies that are higher than a low cutoff frequency (f_1) and lower than a high cutoff frequency (f_2).

You can think of a bandpass filter as a low-pass filter having a cutoff frequency at f_2 followed by a high-pass filter with a cutoff frequency of f_1 .

A band-block filter is a filter that passes all frequencies outside the two cutoff frequencies of f_1 and f_2 . Figure 1 shows the idealized frequency spectrum of all of these filters.

When the difference of f_1 and f_2 is small compared to the absolute frequency, band-pass filters are also called resonant filters. Narrow band-block is sometimes called a notch filter.

Idealized filters are filters with rectangular frequency spectrums. Of course, there's no such thing as an ideal filter in which the filter response drops off vertically at the cutoff frequency and the phase delay is constant for all of the frequencies in its pass band. But, thinking of idealized filters will help you understand digital filters.

DOMAINS

If the frequency-domain representation of a perfect filter response is a rectangle, surely you can describe what this filter looks like in the time domain. One way to do this is to take the inverse Fourier transform of a rectangle or pulse.

Although we typically consider spectra to only cover positive frequencies, they also contain a negative dual. The negative spectrum is a mirror of the positive spectrum. Mathematically, a low-pass filter (0 – F_c) has to be considered a pulse from $-F_c$ to F_c .

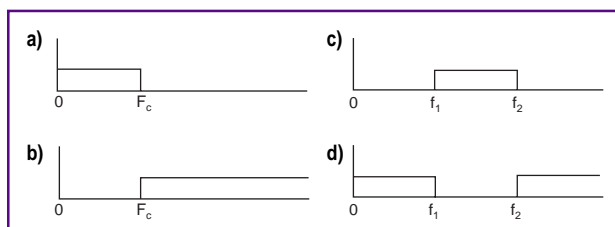


Figure 1—The ideal frequency responses of a low-pass filter (a), high-pass filter (b), bandpass filter (c), and the bandstop filter (d) are rectangular in shape and feature sharp cutoff points.

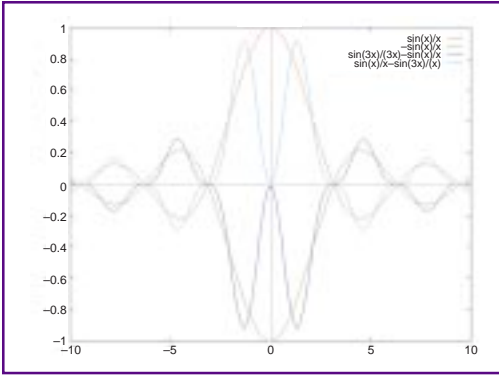


Figure 2—Here's the time-domain impulse response for various filters. The bandpass and bandblock filters have an f_2 equal to $3 \times f_1$.

To find out what this frequency domain pulse represents in the time domain, take the inverse Fourier transform of the pulse. Luckily, the pulse function is one that's popular and the transform can be found in a table.

The Fourier transform for a pulse in the frequency domain of $2F$ width (where F is the cutoff frequency) is the function $2F \times \text{sinc}(2\pi Ft)$. The sinc function $\text{sinc}(x)$ is equivalent to $\sin(x)/x$. Similarly, the transform for the high-pass filter, where the pulse wraps around positive infinity, is

$-2F \times \text{sinc}(2\pi Ft)$. Figure 2 shows what these functions look like.

Great, you might think, what do I do with this? Obviously, it's hard to visualize implementing a circuit that implements the sinc function in analog. At least it is for me. But then, we're discussing digital filters so we don't really care how it might be done in analog. The question is, how do we implement it digitally?

The time-domain representation we found is the transfer function of a circuit that implements such a filter. In the analog world, the transfer function is the impulse response of a circuit. To find out what a transfer function in the time domain does to a signal, we use a technique called convolution.

In the frequency domain, convolution is simply multiplying the functions. But, in the time domain, we're not that lucky. The basic algorithm is:

$$y(t) = \sum_{0 \rightarrow n} x(n\Delta t) \times h(n\Delta t)$$

where $h(t)$ is the transfer function of the filter, $x(t)$ is the input signal, and $y(t)$ is the output signal.

Basically, you start at one end and multiply each discrete time step of the signal with that of the function at the other end. But in our case, the transfer function is symmetric so it doesn't matter which end you start at.

For real signals, let Δ approach zero and then the whole thing becomes an integral over time. For digital signals, you can use the above form. With this information, you might be able to tell that if the input signal is an impulse, the output signal $y(t)$ will be equal to the transfer function $x(t)$. I wonder if they did that on purpose. ;-)

A digital filter can be implemented almost directly from the above form. Take a look at the FIR filter in Figure 3. The coefficient h_0-h_n and the value of the transfer function at discrete time steps are multiplied by the input values as they are delayed and summed to provide the output $y(t)$ at a specific time instance. To implement a real convolution, you'd have to go back in time to the beginning.

It should be almost obvious that if you make the coefficients in the FIR filter match the shape of the $\text{sinc}()$ or $-\text{sinc}()$ function, you can implement both the idealized low-pass and high-pass filter. In fact, with a FIR filter, you can implement all transfer functions as long as they are symmetric.

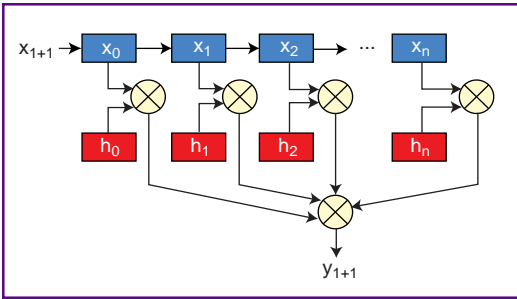


Figure 3—A FIR filter uses delayed input values ($x_1 \dots x_n$) and coefficients ($h_1 \dots h_n$) to compute the output value $y(t)$. The coefficients are derived from the impulse response of the filter.

There is one gotcha with this method. As I mentioned, the input signal $x(t)$ and the transfer function $h(t)$ go on forever in time (i.e., the n in the equation above would be infinity). This situation presents a problem because it's only practical to consider building FIR filters where the number of n steps is finite.

The effect of this time-limited function is that you essentially have a pulse-function response in time overlaid on top of the $h(t)$ function, which causes some ripples in the frequency spectrum.

The pulse has a sinc frequency response, and the resulting filter response is affected by the ripples. For our "ideal" filter, this means that the passband and stop band won't be flat in the pass and block band.

To reduce the effects of the edges, round off the edges of the $h(t)$ function by using a windowing function. The window function makes sure that the edges of the sinc function are zero (or close to it), and minimizes the effects on the rest of the filter.

Different windowing functions have different effects and vary by complexity. A simple windowing function is the Hanning window function.

The coefficients you compute for the filter are adjusted by a function that looks like a half circle with diameter of the number of steps (N), or taps, in the filter:

$$0.5 + 0.5 \times \cos\left(\frac{2\pi n}{N}\right)$$

Other windowing functions are a variation of the Hanning functions. The Hamming window function is quite popular and gives a little flatter frequency response than the Hanning function:

$$0.54 + 0.46 \times \cos\left(\frac{2\pi n}{N}\right)$$

FILTER EXAMPLE

So, let's put this all together by doing an example. Say you want to design a low-pass digital filter to cut off at 1 kHz when the sampling rate is 8 kHz. In DSP, it's convenient to express the frequencies as a ratio of the sampling rate, so 1 kHz at 8 kHz is $\frac{1}{8}$, or 0.125. Let's find the coefficients for a 15-tap FIR filter ($N = 15$) and compare the response with a different windowing function.

First, calculate the constants you need to compute the sinc function for this low-pass filter. Remember, the function is defined as:

$$2F \times \text{sinc}(2\pi Ft)$$

Scale the frequency by 0.1 and plug it into the equation and you get:

$$\frac{0.2 \times \sin(0.628t)}{0.628t}$$

The coefficients are then computed at for the 15 taps. The first column in Table 1 shows the coefficients. The window function is computed for the coefficient and

multiplied together with the coefficient.

Columns 2 and 3 in Table 1 show the modified coefficients for the Hanning and Hamming windows, respectively. Figure 4 plots what the coefficients look like graphically. You see the windowed functions have coefficients that are nearly zero at both edges.

I computed the response to these filters by running an impulse into the input and looking at the output signal using a digital Fourier transform program. Figure 5 shows the frequency responses for the three filters. As you see, all three filters implement a low-pass filter that has a 3-dB (half magnitude) point at 10% of the total band.

As predicted, the nonwindowed filter has a bunch of ripples, whereas the Hanning filter is much smoother. The Hamming filter is only a little smoother than the Hanning filter and a little steeper in the transition band.

Now that I've demonstrated how to compute coefficients for a FIR filter, let's talk about some issues. The method I used is only one approach for designing a FIR filter. See the references for other methods.

How do you determine the number of time steps (or taps) for a FIR filter?

In general, the number of steps determines the steepness of the filter. In the example, I showed a 15-tap filter, which roughly gives you a transition band of $\frac{1}{15}$ the sampling rate. A 2-tap filter gives you a transition rate of half the sampling rate. Figure 6 shows the response for different number of taps for our filter.

The half-band filter is a special kind of filter that's used to filter either the top or bottom half of the spectrum. It's special because every other coefficient is zero (which leads to efficient implementation) and only half of the computations have to be performed, compared to a filter which has the same number of taps but isn't exactly a half-band filter.

I've given you a brief background in designing and computing the coefficients for digital filters. These filters can be implemented with any kind of digital computer. The computational complexity is related to the number of taps in the filter as well as if any special conditions like half-band are implemented.

General-purpose computers have fixed word-length instructions (i.e., a 32-bit general-purpose computer can efficiently perform 32-bit integer operations like adding and multiplying). These machines usually have separate multiply and add instructions.

Computers that are optimized for signal processing usually have a multiply-and-add instruction that accumulates the result into a register. These instructions can implement operations like FIR filters efficiently (with one instruction per tap), but they are still linearly executed.

Some DSPs use fixed-point numbers (integers) and others use floating-point numbers. Obviously, using floating-point numbers is nice because you don't have to worry about truncation or overflow as much as with fixed-point numbers. Also, floating-point numbers have much better dynamic range. But floating-point DSP processing is expensive, and DSPs that implement floating point are expensive compared to fixed point.

For simple algorithms such as FIR filters, the truncation and overflow is predictable and, with careful design of the software or hardware, isn't an issue—especially where cost and speed concerns are important.

The sequential nature of computers and even DSPs are a drawback when trying to implement high-speed signal-processing algorithms (e.g., FIR filters). Some processors have multiple data paths.

For example, the MMX instructions in a Pentium CPU can operate on four pieces of 16-bit data simultaneously, but this is only a 4x improvement over purely sequential computers. For these algorithms, the inner loop that executes the MAC instructions over and over is the bottleneck.

When DSPs and processors aren't fast enough and using arrays of DSPs isn't cost effective, we turn to hardware to implement these functions. Although neither solution is the save-all, systems composed of a general-purpose processor to control low-speed signal processing, DSPs to handle medium or floating-point signal processing, and hardware to perform high-speed processing are usually found in many applications.

T	None	Hanning	Hamming
-7	-0.000473	-0.043247	-0.003895
-6	-0.031183	-0.002978	-0.005234
-5	0.000000	0.000000	0.000000
-4	0.046774	0.020943	0.023009
-3	0.100910	0.066047	0.068836
-2	0.151365	0.126324	0.128328
-1	0.187098	0.179010	0.179657
0	0.200000	0.200000	0.200000
1	0.187098	0.179010	0.179657
2	0.151365	0.126324	0.128328
3	0.100910	0.066047	0.068836
4	0.046774	0.020943	0.023009
5	0.000000	0.000000	0.000000
6	-0.031183	-0.002978	-0.005234
7	-0.043247	-0.000473	-0.003895

Table 1—This table shows the coefficients for the 15-tap FIR filter with and without the windowing functions applied.

FPGAs AND FILTERS

Field-programmable gate arrays are a good fit for implementing high-speed algorithms. They offer many architectural features like look-up tables and fast carry-chains. Also, many FPGAs are in-circuit reprogrammable so it's almost as easy to change the algorithm implemented on them as it is on a processor (one reason FPGAs are so popular in the signal processing and telecommunication industry). Once you have an FPGA-based signal-processing chain, it's usually easy to change or adjust the parameters or algorithm, even after the product is already in production.

There are some special techniques that you can use when implementing signal-processing functions like the FIR filter in FPGAs. Let me show you some of the methods that can be used to implement FIR filters.

The most obvious improvements in hardware are that each tap can be computed in parallel and summed with wide adders (i.e., adders with more than two inputs). By adding pipeline registers, it's possible to design FIR filters that can compute a new $y(t)$ sample for each clock cycle. A processor needs at least N cycles (one for each tap) to compute one sample.

When implementing the type of FIR filters I've talked about, the coefficients are symmetrical. Here, $h_0 = h_{14}$, $h_1 = h_{13}$, and so on. You can use this to your advantage and even change the order somewhat.

Figure 7 shows that adding the $x(n)$ values for the symmetric taps before you multiply saves you about half the multiplier. Till now, I've only designed odd-numbered taps, but it's possible to design FIR filters with even-numbered taps.

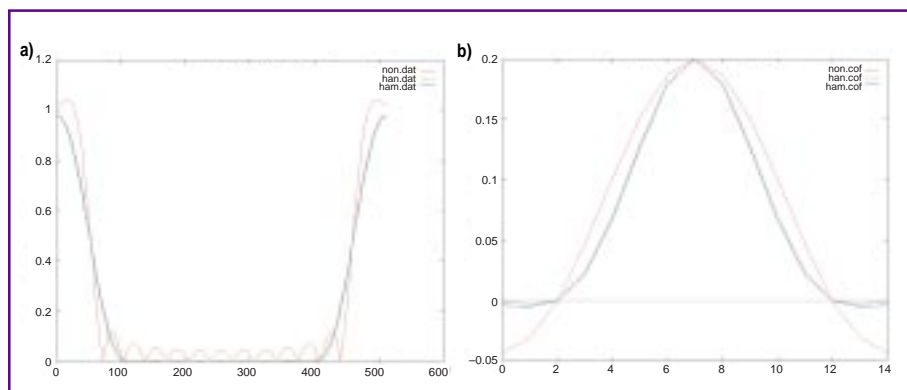


Figure 5—The effects of windowing on a 15-tap FIR filter. All of these implementations have a cutoff frequency of 0.125 Fs but vary in their steepness and how smooth they are in the pass and block band. Figure 4—This graph shows the time impulse response for the 15-tap FIR filter with various windowing functions. Notice that for the two-window function, the edges are close to zero.

Also, because SRAM-based FPGAs are reloadable, you can implement the multipliers as constant multipliers. Instead of reloading registers or arrays that might hold the constant values and using general multipliers, you just reload the whole FPGA with a new filter.

For example, in a digital radio, you might have a 6-kHz filter for AM, a 3-kHz filter for single-side band, and a selection of really narrow filters (300 or 500 Hz) for Morse code. You'd change the filter in the FPGA with a general-purpose CPU whenever you switch receiving modes. Reloading an FPGA takes tens of milliseconds. This is a good example of the system approach of using a CPU for control and an FPGA to accelerate processing.

The next optimization isn't quite as obvious at first. If you implement the adders and multipliers as bit-serial components, you can sometimes gain speed. Each tap is still done parallel to each other, but the whole chip is clocked at the bit rate instead of the sample rate, which means the bit rate for the chip might be 8x the sample rate (if the word width is 8 bits).

Serial adders that run at close to the maximum clock rate of the FPGA can be implemented faster in many FPGAs because all of the signals needed to compute the next bit of a word are local and routing is minimized. With parallel adders, the speed is limited by the routing of the carry chain. Obviously, the tradeoff between bit-serial adders and parallel adders has to be evaluated for each design and FPGA architecture to find out which is faster.

These are just a few of the optimizations that can be done in hardware. Others,

Band		Name	Frequency sense
Low	High		
0	1/2 Fs	1st Nyquist band	low → high
1/2Fs	Fs	2nd Nyquist band	high → low
Fs	3/2Fs	3rd Nyquist band	low → high
3/2Fs	2Fs	4th Nyquist band	high → low
2Fs	5/2Fs	5th Nyquist band	low → high

Table 2—Here are a few different Nyquist bands that can be used by undersampling.

such as distributed arithmetic, would take a whole article to describe.

To find out more, the web sites of the FPGA vendors listed under Sources contain application notes that describe how to squeeze even more speed from the parts for signal processing. Some FPGA vendors even have downloadable software that generates these constructs for you.

OTHER FILTERS

As I mentioned last month, my approach is to use a multi-FPGA-based high-speed signal processor married to a generic CPU for control and low-speed processing. I also looked at the CORDIC algorithm that was implemented in hardware to synthesize sine waves and used to mix the input signal down.

Another element I talked about was the decimation filter, which filters part of the frequency spectrum before reducing the sampling rate. The half-band filter I mentioned is often used as a 2:1 decimation filter before reducing the sampling rate by half.

However, at the input side of my radio, the sampling rate is high. So, building multi-tapped FIR filters isn't easy, even with most of the obscure design techniques available. Here, I used a Comb filter.

You can think of a Comb filter as a FIR filter where all of the taps use the same coefficient. A 4:1 decimation filter based on this architecture would just add the last four x values without multiplying by a coefficient.

This type of filter computes the running average of the input signal. Because the time-domain representation is a pulse shape, the frequency response now looks like a sinc function. The low-pass portion looks like an old-fashioned barn door, hence the nickname "barn-door filter."

Figure 8 shows a 4:1 Comb filter spectrum. It's a crude low-

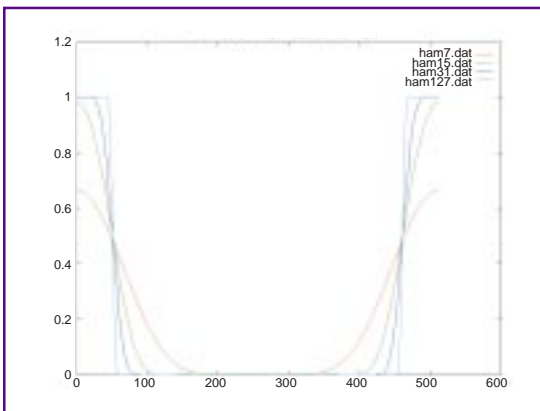
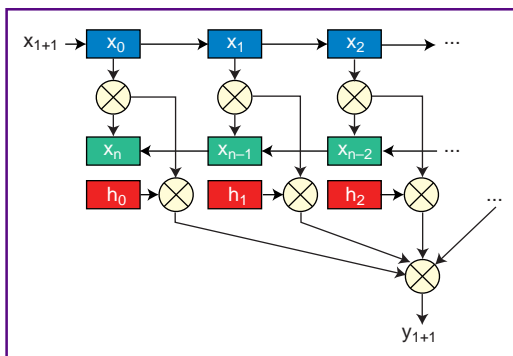


Figure 6—By increasing the number of taps in the FIR filter, you can make the transition band steeper. The cost, of course, is the higher computation rates necessary to implement the filter.

Figure 7—Because coefficients are symmetric in this FIR filter implementation, you can add the corresponding x values and then scale them only once, which reduces the number of multiplications required by up to half.



pass filter with a sloppy cutoff and ripples in the block band. It's the digital equivalent of a RC low-pass filter.

But because the filter is in the front-end of the radio, you can compensate for the sloppy drop-off in the high-end off this filter by designing a FIR filter that compensates for this droop. You can compute the frequency spectrum of this FIR filter and do a Fourier transform to the time domain. In the time domain, this operation is likely to be a modified sinc function from which you pick the coefficients used in a FIR filter.

Computing this transfer function is beyond the scope of this article and is best done with CAD tools rather than working through the transform by hand. Check out some of the DSP textbooks available for techniques and software.

UNDERSAMPLING

When learning about data acquisition, you were told to use an antialias low-pass filter with a cutoff frequency of half the sampling rate. Even though the Nyquist sampling theorem says you need to have a sampling rate of twice the bandwidth you're interested in to reproduce the original signal, it doesn't say that the band of

interest has to be the baseband (i.e., from DC to $\frac{1}{2} F_{\text{sampling}}$). In fact, it can be any multiple of this band. Table 2 shows examples of these bands.

This whole process is just another form of mixing, except this time you use a digital signal as the local oscillator. This causes many harmonics to be present in the product (besides the sum and difference), which makes it possible to sample several bands. This mixing is, in essence, aliasing.

All you need now is to bandpass-filter the band of interest, assuming your ADC has sufficient analog bandwidth. The ADCs I used (Analog Devices AD6640 and Burr-Brown AD807E) are specially designed for this application and have analog bandwidth of well over 200 MHz.

You can also sample more than one Nyquist band at one time if the signals you're interested in don't end up on the same frequency. For example, if the sampling rate is 10 MHz, then a signal in the second Nyquist band at 18 MHz will look like a signal at 2 MHz. But, you can look at a 3-MHz and an 18-MHz signal at the same time.

By using several analog bandpass filters that can be switched in and out with relays before the ADC, I can extend the basic receiver spectrum of my receiver from 0–10 MHz to 0–50 MHz, keeping in mind that I can only look at one 10-MHz band at a time.

The only gotcha is that I can't listen to a signal that's exactly a multiple of my sampling rate if the phase is exactly aligned (coherent). It will look like a DC signal. But this is very unlikely—especially in radio astronomy, where we look at a wide spectrum of signals anyway.

Next time, I'll cover the software and interface on the CPU that powers this system. The CPU is a PC/104-

based SBC that runs RT-Linux. It configures the FPGAs with the signal-processing cores, selects the frequency in the CORDIC processor in one of the FPGAs, and reads out the downconverted data from the FPGA-based signal (co)processor. RPC.EPC

Ingo Cyliax has written for Circuit Cellar on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego-based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. You may reach him at cyliax@derivation.com.

REFERENCES

- K. Chapman, *Constant Coefficient Multipliers for the XC4000E*, Application note, Xilinx, 1996.
- L.W. Couch II, *Digital and Analog Communication Systems*, Macmillan, New York, NY, 1983.
- G. Goslin and B. Newgard, *16-Tap, 8-Bit FIR Filter Application Guide*, Application note, Xilinx, 1995.
- E.C. Ifeachor and B.W. Jervis, *Digital Signal Processing*, Addison-Wesley, Reading, MA, 1993.
- S.K. Knapp, *Using Programmable Logic to Accelerate DSP Functions*, Application note, Xilinx, 1995.
- C.B. Rorabauch, *Digital Filter Designer's Handbook*, McGraw-Hill, New York, NY, 1997.
- Xilinx Corp., *The Fastest Filter in the West*, Application note, 1996.
- Xilinx Corp., *The Programmable Logic Data Book*, 1998.

SOURCES

Floating and fixed-point DSPs

Texas Instruments, Inc.
(508) 236-3800
www.ti.com

Motorola
(512) 895-2649
Fax: (512) 895-1902
www.mot.com/sps/general

ADCs

Analog Devices, Inc.
(800) 262-5643
(781) 937-1428
Fax: (718) 821-4273
www.analog.com

Burr-Brown Corp.
(520) 746-1111
Fax: (520) 889-1510
www.burr-brown.com

FPGAs

Xilinx Corp.
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com

Altera Corp.
(408)544-7000
Fax: (408) 544-7755
www.altera.com

Actel Corp.
(888) 992-2835
(408) 739-1010
www.actel.com

Lattice Semiconductor Corp.
(503) 681-0118
Fax: (503) 681-3037
www.lattice.com

PC/104 FPGA board

Derivation Systems, Inc.
(760) 431-1400
Fax: (760) 431-1484
www.derivation.com

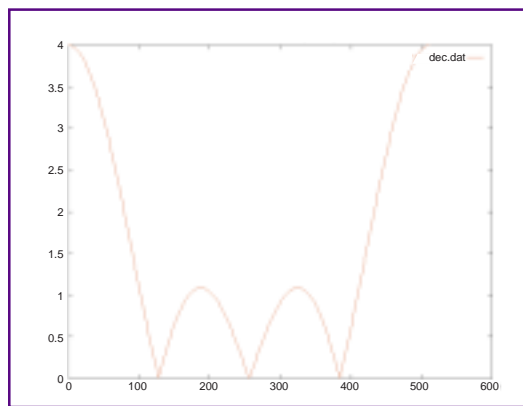


Figure 8—In this frequency response of a 4:1 Comb filter, you can see the magnitude of the sinc function (i.e., the negative components are positive). With imagination, the low-pass portion looks like a barn door.

Embedded Internet

Part 2: TCP/IP and a 16-bit Compiler

Fred continues his journey into embedded 'Net technology by implementing the Netsock/100, an embedded PC with a built-in TCP/IP stack. Regardless of your coding skills, WinSock programming can be yours!

People with bad habits check into clinics to clean up their acts. I'm wondering if I should find a clinic for embedded junkies. I can't get enough of this stuff!

Last month, I discussed TCP/IP with a hint of embedded and promised some additional dialog. In the meantime, I've been on the phone and on the web.

As you might imagine, a box recently arrived on my porch. I already know what's inside and I think you'll find it very interesting. With that, let's get to work!

SOMETHING OLD, SOMETHING NEW

In the past, I've extolled the virtues of Phar Lap's Embedded ToolSuite. It's everything one could ask for in an embedded application tool. Just last month, I used it to promote the embedded TCP/IP paradigm we're discussing right now. Thus, I am a believer in the Phar Lap product. I've used it. I've written about it. It works.

I said all of that to say this: although that product is very good, I have an

opportunity to tell you about something new. What if I could show you how to write TCP/IP-enabled programs with a simple 16-bit C compiler? And, what if I could show you how to write Ethernet-based client/server TCP/IP enabled applications using Visual Basic and that little 16-bit C compiler?

Well, I can. The gravy is that the ideas I'll present here can be done with the Phar Lap tools, too. After all, this is about TCP/IP, a rock-solid standard in the networking world. TCP/IP is a suite of protocols based on the concept of a logical stack.

Remember the good times playing ball in an open field with a bunch of friends? There was always the kid who had the equipment. You know, the one with the ball, bat, or whatever the sport required. The game couldn't be played without him and his gear.

Doing TCP/IP stuff is similar to that backyard ballgame. Someone has to bring the ball. To implement TCP/IP, someone has to bring a stack.

I've been teasing you along with my "what ifs" about software and TCP/IP, but what if I could show you an embedded PC that was designed from day zero to network using the 16-bit-compiler TCP/IP-stack principle?

Well, I can. I brought the ball and I'm ready to play. The "ball" is the Micro/Sys SBC1190 embeddable computer board in Photo 1. The game is TCP/IP over Ethernet.

NETSOCK/100

Deep thoughts by Fred. "If a windsock indicates the direction of the wind, does a Netsock indicate the direction of networks?" Hmmm....

I don't know about you, but I'm sick and tired of writing code that has to depend on a serial port to communicate with the outside world. Don't get me wrong. I realize that serial ports are universal to a certain degree and are good ways to put embedded and not-so-embedded computers in touch. On the

other hand, Ethernet is so elegant and so fast.

But Ethernet is as hometown and apple pie as RS-232, so why not use it? The Micro/Sys Netsock/100 (a.k.a. SBC-1190) embeddable PC does.

The Netsock/100 is a dainty little embedded PC that packs a powerful punch. The SBC1190 is based on the Intel 80C188EB and thus is an XT-class machine. The Netsock/xxx series includes '386, '486, and Pentium processor power-plants as well.

As you know, the 80C188EB is a 16-bit CMOS CPU that executes 8086 instructions. The selling point of the 80C-188EB is the integration of common PC peripherals and a single +5-V power requirement.

By design, there's only eight bits at the data bus. The 80C188EB was built to reduce chip count in its domain. On-chip peripherals include an interrupt controller, three 16-bit timer/counters, a wait-state generator, and two UARTs. Figure 1 blocks out the SBC1190.

When I receive new embedded platforms, I like to just sit and gander at the work of the board layout engineer. The SBC1190 layout is extremely efficient. ICs are packed in tight and populate both sides of the board. I/O connectors are gold-plated male headers and there's even a PC/104 connector for the 104 heads. Take a peek at Photo 1 while I give you a play by play of who's what.

SBC1190 memory consists of a 32-pin JEDEC memory socket and 512 KB of surface-mounted SRAM on the top side of

the board. Although there are numerous ICs on the back side, the main part that stands out is the 512 KB of 5-V flash memory. RUN . EXE firmware is resident in part of the flash memory.

As an industrial-grade BIOS, a DOS emulator, and an XMODEM download system, RUN . EXE initializes the 80C188EB registers and sets up the environment to mimic a PC running DOS. This setup enables the programmer to be free from worrying about things like memory sizing and initialization and system timing.

The RUN . EXE app is small and resides with the power-up vector at the top of memory, permitting the flash memory to carry downloaded applications as well. If the program you write can be compiled to run as a PC-compatible 16-bit executable, the SBC1190 (with RUN . EXE) can run it.

If your application is coded in C, RUN . EXE redirects all console I/O to COM A. RUN . EXE isn't required if your needs are particular. The Micro/Sys SBC-1190 reference manual (all 0.5825" of it) includes all the data for all of the peripheral components on the card. In so many words, you can roll your own RUN . EXE.

An onboard flash programmer enables the downloading of an application program into flash. The flash programmer is implemented as one bit of the 80C188EB port 1 that enables writes to the flash memory.

At powerup or reset, the SBC1190 can be instructed to execute the program residing in the remainder of lower flash

memory. A minimum of 100,000 download cycles would be necessary to "break" the flash memory. Unless you're from Mars and write Plutonian code using a Venusian 1-bit compiler, you won't have to change the flash out in your lifetime.

Need extra RAM? No problem. There's a socket for 32 or 128 KB of SRAM or EPROM onboard.

The two asynchronous serial ports are integrated into the 80C188EB. RUN . EXE covers for DOS, but the serial ports aren't PC



Photo 1—The Ethernet circuitry is between the battery and the Ethernet connector. ADC and DAC capability can also be added under and around the battery.

compatible. That's a drawback as far as writing for compatibility, but the ports make up for it by operating in 9-bit mode and incorporating high transfer rates that can reach above 200 kbps.

The 80C188EB serial port 0 is COM B, and serial port 1 terminates at the COM A connector pins. The COM B port is used for debugging because it's interruptible.

COM A's utility lies in its duality. With the movement of a jumper, COM A can be either RS-232 or RS-485. Micro/Sys offers the CommBLOK software library for those that need to be compatible.

I noticed that the Micro/Sys reference manual spent a bunch of words on serial ports. So, if you plan to work the SBC1190 serial ports like cheap labor and haven't done any 80C188 coding in a while, read the manual carefully in this area.

Digital I/O is supplied via the ever-present 82C55. I don't have to say anymore about that.

As well, there's an integrated interrupt controller with seven maskable and one nonmaskable interrupt. Two interrupts internal to the 80C188EB are sourced by the timers and the COM A serial port.

There's also a 1.6-s onboard watchdog timer and three timer/counters. Dallas Semiconductor makes the SBC1190 timely with a DS1302 clock chip, and SMC provides the LAN connectivity with a LAN91C94 Ethernet controller IC. The SBC1190 becomes a Netsock/100 with the addition of the Ethernet controller and Embedded Netsock system.

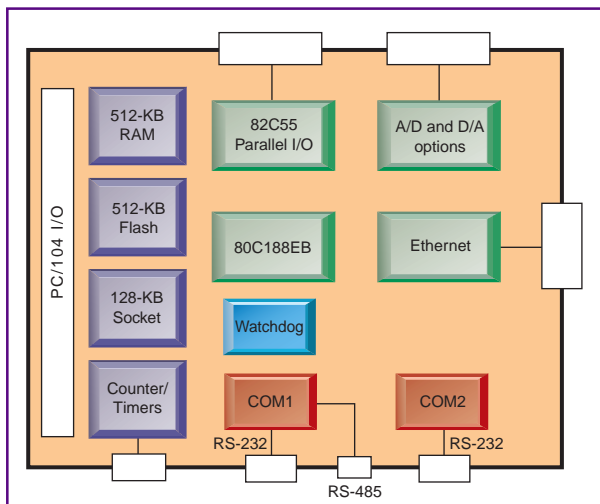


Figure 1—Just think. Years ago, a not-as-well-equipped XT computer like this was heavy and cost thousands of dollars!

Listing 1—*netsock.h* is all you need to move from the serial world to the socket world.

```
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define NETSOCK_MASTER
#include "netsock.h"

#define DACQ_PORT 5001
//the UDP port this server will listen on
```

Listing 2—With the exception of the socket stuff, this is standard C, too.

```
// Global Variables
WSAData SocketData;
unsigned long messagesize;
int fromlen,numbytes,debug,commanddone,err;
char datagram[80];
char parameters[11];

SOCKET msgsock;
struct sockaddr_in local, from;
```

Listing 3—I tried running this on a desktop without Embedded Netsock support, and sure enough, I received the message from the ENE_LDEER_BIOS error trap.

```
err = WSASStartup(0x101, &SocketData);
switch (EmbeddedNetsockLoadError)
{
case 0:
    printf("Embedded Netsock started..\n\r");
    break;
case ENE_LDERR_BIOS:
    printf("System BIOS does not support Embedded
        Netsock!\n\r");
    break;
case ENE_LDERR_ADAPTER:
    printf("No network adapter found!\n\r");
    break;
case ENE_LDERR_MEM:
    printf("Error allocating memory!\n\r");
    break;
case ENE_LDERR_NETSOCK:
    printf("Netsock not available!\n\r");
    break;
}
if (err)
{
    printf("WSASStartup failed with error
        %d\n\r",err);
    WSACleanup();
    exit(1);
}
```

That pretty much describes the SBC1190, but there's one more important item on the board. Data bit D6 of the 80C188EB port 1 is connected to an LED. Setting and clearing this bit illuminates and extinguishes the LED. Das blinkin lites—a feature built in just for me!

ETHERNET, EMBEDDED STYLE

Building on your knowledge of TCP/IP and your newly acquired knowledge of the Netsock/100, we can move the kind of bits across Ethernet that we embedded-control types are used to moving.

Some embedded apps are control-oriented only. These include machine control, fluid-level measurement, and general data acquisition. Although I know that many of you push embedded boxes to their logical limits, we'll concentrate on control this time around.

So, we have a Netsock/100 embedded PC with a TCP/IP stack onboard. The Netsock/100 will be called the "server" in our configuration. The client is a standard desktop PC running Bill's Windows 98.

Mark this in your PDA: Fred is wobbling off the beaten path. No Bill C compiler for this application. For the first time in this series, a Borland product (Borland C++ 5.02) will be the compiler of choice on the server side. Visual Basic 5 and its WinSock control will support the client. Here's the plan.

THE OLD UNRELIABLE

To make this complicated, all I'd have to do is follow the Internet TCP/IP-to-WinSock paradigm. I'd write code that includes all of the "historical" ways to do TCP/IP coding and rigidly follow the WinSock API standards.

For traditional TCP/IP implementations, each layer (by definition) must be totally independent of the surrounding layers, which implies that any datalink protocol should be able to work with any physical protocol and any transport protocol should be able to work with any network protocol. All of this adds up to passing information from layer to layer because layers don't know anything about other layers.

This information-passing trap is also true for data entering the stack. Micro/Sys and I have the same motto: it doesn't have to be complicated. So, here's how the

WinSock API subset

```
socket()
closesocket()
setsockopt()
getsockopt()
recvfrom()
sendto()
bind()
ioctlsocket()
inet_addr()
inet_ntoa()
WSAStartup()
WSACleanup()
WSAGetLastError()
```

Table 1—Put an "EN" before some of these and you have the names of the easier-to-use alternate API subset.

Embedded Netsock and I will deploy the Netsock/100.

First of all, control applications rarely venture out of the local Ethernet network. This eliminates routers, phone lines, and all the other traditional Internet stuff. And, because Ethernet is proven and reliable, we can employ the "unreliable" UDP.

Using UDP reduces the coding complexity because it is connectionless and doesn't require the data-integrity overhead that TCP does. You just have to know the address of the receiver and send the message.

The procedure is like doing an RS-232 serial datalink (i.e., you know where the

data is going, the other side knows where it's coming from, and acknowledgements are optional). On the RS-485 side, UDP over Ethernet adds multidrop capability and additional speed.

Second, Windows products on standard desktop or embedded computers are a good choice for embedded network management. The NT products provide a relatively easy means of monitoring and administering a gaggle of embedded computers. The Windows 95/98/NT computers also act as buffers to outside networks, just in case your data must act like Elvis and leave the building.

Finally, Embedded Netsock is preinstalled in flash memory on the Netsock/100. All you need is a compiler that can create 16-bit DOS .EXE files. Most of Bill's and Borland's compilers meet this requirement.

Borland's Turbo Debugger works a little better with the Netsock/100 than Bill's. Add NETSOCK.H as include in your application code, and the features of Embedded Netsock are at your disposal.

If you're not a TCP/IP coder, Micro/Sys includes an Embedded Netsock API that takes some of the particulars and gotchas out of the implementation of the standard API. The Micro/Sys API is similar in function and simpler to use. The standard WinSock API subset is shown in Table 1.

Listing 4—I admire thinkers, but the Embedded Netsock Alternate API was designed to help the inexperienced TCP/IP programmer eliminate gibberish like the *in_addr* structure.

```
msgsock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (msgsock == SOCKET_ERROR)
printf("Error %d from socket()\r\n",msgsock);

local.sin_family = AF_INET;
local.sin_port = htons(DACQ_PORT);

typedef int SOCKET; // From NETSOCK.H

// standard 32-bit IP address structure (Berkeley) is confusing because
// of access by bytes, words, or long
struct in_addr
{
union
{
struct { unsigned char s_b1, s_b2, s_b3, s_b4; } S_un_b; // byte access
struct { unsigned short s_w1, s_w2; } S_un_w; // word access
unsigned long S_addr; // long access
}
S_un;
};

#define s_addr S_un.S_addr // nickname for accessing in_addr as long

// standard structure for specifying socket address (Berkeley)
struct sockaddr_in {
short sin_family; // address family (i.e. AF_INET)
unsigned short sin_port; // port for transport layer
struct in_addr sin_addr; // 32-bit IP address structure
char sin_zero[8]; // unused padding
};
```

SERVING THE WIN98 MASTER

I keep reminding myself to get a program that does screenshots with NT. Again, I'm forced to move to an OS that supports HiJaak 95. I've heard awful things about Windows 98, and for a time, I refused to even load a Windows 98 testbed in the *Circuit Cellar* Florida Room. I also recall colleagues putting Windows 95's face into the dirt, too.

Personally, I've never had a problem with Windows 95 that I couldn't solve. So, I'm going to give Windows 98 the same treatment. If I can't tame it, I won't use it. I recently put up three machines running Windows 98 in my lab, and other than my ignorance, I've experienced zero problems. Am I doing something wrong?

Anyway, if any of you have suggestions for a new screenshot program, drop me a line. For now, Windows 98 is the client OS for our Netsock/100.

If we can move a simple command across the client/server bridge, then we can move anything else we want across the same path. In this example, I'll target the Netsock/100's 82C55 with simple commands issued by the Windows 98 client.

I don't have room to show the complete listing, but as the code goes, there's not much difference in the logic that you'd normally employ to do this operation with a serial port. Instead of COM ports, I'll use sockets. With that similarity in mind, let's take a command through the hoops.

Just like any other embedded program, the first thing to do is initialize the environment. Listing 1 is typical of the first few lines of any C program with the exception of the netsock lines.

NETSOCK_MASTER is a common definition for both the standard and alternate API. Because a define for ALTERNATE_API isn't present, this program uses the standard WinSock API functions.

To complete the address, the port must be specified, which is done in the DACQ_PORT line. Note in the comments of the port line that UDP is implied. You'll see later where this is defined.

Listing 2 is a view of the global variables that have to do with WinSock. Of course, all of the variables that the program would use are included here, too. I'm not going to explain these because you'll see them out in the open as we continue.

Now that all of the initial and necessary C things are done, it's time to start the



Photo 2—All of this data is transferred to a structure called *NetsockConfig* for later use by the application calls.

WinSock engine. Calling `WSAStartup` (shown in Listing 3) gets things going.

`WSAStartup` initiates the use of the Embedded Netsock TCP/IP stack and starts the underlying network layers. Note the reference to the `SocketData` structure I defined in the `Global Variables` area.

The `SocketData` structure receives details of the available Embedded Netsock support. The `0x101` denotes WinSock API V.1.1. If `err` is returned as 0, then all is well and the program can continue. My program is designed to post the error and perform a `WSACleanup` before exiting.

`WSACleanup` terminates Embedded Netsock functions and releases any memory allocated by `WSAStartup`. Once the Embedded Netsock is up and running, you can interrogate the `SocketData` structure for any information you want to pass along to the user.

The idea is to establish communications between two endpoints or sockets. So, let's socket to it at the server end. Listing 4 shows `msgsock`, which was defined at type `SOCKET` in the `Global Variables` area, as the receiver of the results following the creation of our socket.

`AF_INET` is the TCP/IP address family parameter and the only one Embedded Netsock supports. `SOCK_DGRAM` tells us that datagrams will be used, which implies that the link will be connectionless and fixed in size. `SOCK_DGRAM` also implies that UDP is the protocol of choice. Looking at the next parameter (`IPPROTO_UDP`), this becomes clear.

Using datagrams and UDP enables you to send and receive from arbitrary hosts using the `sendto()` and `recvfrom()` calls. Good old "unreliable." The final lines, beginning with `local`, stuff values into the `sockaddr_in` structure called `local`. The `sockaddr_in` structure is commonly referred to as the name of the socket.

Note that there is a `local` and `from` structure. The host-to-TCP/IP-network (htons) byte order call puts the port address in the proper orientation. In other words, our host order `0x5001` port address in network order is `0x0150`. I included the `netsock.h` definitions in the lower frame of Listing 4.

To effect an end-to-end link, you need:

- local host IP address (192.168.1.50)
- local host port number (5001)
- remote host IP address (unknown now)
- remote host port number (unknown now)

Photo 2 shows the Netsock/100 network setup screen, which is where the local host IP address is set. Ports and applications go together in the TCP/IP world and that's where we set the local-host port number (see Listing 1). I don't know about the remote host yet, and if I did, there's nothing I could do about it.

There is a local-host port address and a local-host socket that know nothing about each other. You have to bind the socket and local-host address (IP address + port address) together to form an endpoint at the server side.

Essentially, the socket is nameless before the bind. The name in this instance is:

- TCP/IP address family—`AF_INET`
- host IP address—192.168.1.50
- application port number—5001

The TCP/IP address family and application port number parameters are manually entered via code. The host IP address is set in the Netsock/100 flash and retrieved during the initialization of Embedded Netsock (see Photo 3). The bind is shown in Listing 5.

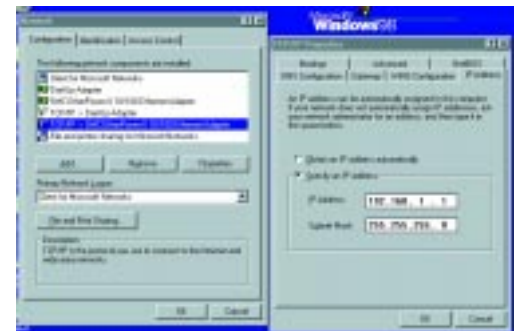


Photo 3—If you've ever wondered why, this is some of the stuff sent with the request. In this case, the IP information is like a return address on an envelope.

The next thing we need to do after the bind is successful is to wait for requests from clients and respond to them. In our case, the requests are really commands that work against the 82C55.

Listing 6 uses the `ioctlsocket` call to determine if any bytes are waiting to be processed from the socket we just bound.

If there are bytes in the buffer, the `recvfrom` call is made to retrieve them.

Assume you've retrieved your command bytes from the datagram and stored them in the command array as shown. At this point, you can execute the command and do nothing in response or execute the command and respond to the requestor or client. But how do you know where the request or command came from?

Listing 5—A successful `WSAStartup` must be completed to "fill in the blanks" that aren't entered manually.

```
// The "local" lines are from Listing 4, inserted here for clarity
local.sin_family = AF_INET;
local.sin_port = htons(DACQ_PORT);

err = bind(msgsock, &local, sizeof(local));
if (err == SOCKET_ERROR)
    printf("Error binding socket: Error %d\n\r", WSAGetLastError());
```

Listing 6—The `FIONREAD` is a must-do. All of the other parameters should be easy to identify now.

```
err = ioctlsocket(msgsock, FIONREAD, &messageSize);
if (err == SOCKET_ERROR)
    printf("ioctlsocket failed with error %d\n\r", WSAGetLastError());
if (messageSize)
{
    fromlen = sizeof(from);
    numbytes = recvfrom(msgsock, datagram, sizeof(datagram), FLAGS_ZERO,
        &from, (int far *) &fromlen);
    if (numbytes == SOCKET_ERROR)
        printf("Failed to receive datagram: Error %d\n\r", WSAGetLastError());
    memcpy(command, datagram, 1);
    command[1] = 0; // null terminate string
    commanddone = 0;
}
```

Listing 7—This is almost automatic. The only parameter you need to add is your datagram.

```
err = sendto(msgsock, datagram, strlen(datagram), FLAGS_ZERO, &from,
    sizeof(from));
if (err == SOCKET_ERROR)
    printf("Failed to send datagram: Error %d\n\r", WSAGetLastError());
```

Listing 8—Visual Basic goes a long way to make it easy to do TCP/IP.

```
Public Sub Form_Load()
    With Winsock
        .Protocol = sckUDPProtocol 'set protocol to udp
        .RemoteHost = "192.168.1.50" 'initialize remote host ip address
        .RemotePort = 5001 'initialize remote host port number
        .Bind 5001
    End With
End Sub

Private Sub Winsock_DataArrival(ByVal bytesTotal As Long)
    Winsock.GetData gstrData 'your data handling code goes here
End Sub

Public Sub SendMsg(ByVal gStrSnd As String)
    ' pass string to Winsock control to be sent
    Winsock.SendData gStrSnd
End Sub
```

The `recvfrom` call not only garners the data but also stores the source IP address and port of the requestor. Here, the `from` structure I discussed earlier is used. By adding a return datagram of your choice, you can send that datagram to the requestor using the `sendto` call and the `from` structure as shown in Listing 7.

It's a bit easier on the Visual Basic side. Listing 8 shows how the VB client gets addresses and protocols for its socket along with the methods and events to send and retrieve data.

The VB-client IP address is set in the Windows 98 TCP/IP networking area. The VB WinSock object dynamically sets the VB-client local port for this application and can be a fixed value, if you so desire.

GETTING SMARTER

If you're a C coder, a VB coder, or a noncoder, WinSock programming is within reach. If you're new to C and C++, get the "Teach Yourself C++ in 21 Days" package with the Borland 3.1 C++ compiler on CD. The Micro/Sys folks put a copy in my box because it works well with the Netsock/100.

The Netsock/100 documentation was useful, with datasheets for everything on-board. I suggest you get your own datasheets though, because some of the deeper stuff may not be in the Micro/Sys sheets.

So, apply what you know about serial applications and put the fundamentals of WinSock programming to work. You'll see, it isn't complicated to be embedded with TCP/IP. [APC.EPC](http://www.apc.epc.com)

Fred Eady has over 20 years' 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.

SOURCES

Netsock/100
Micro/Sys
(818) 244-4600
Fax: (818) 244-4246
www.embeddeddedsys.com

Visual Basic
Microsoft Corp.
(206) 882-8080
Fax: (206) 936-7329
www.microsoft.com

Borland C++
Inprise Corp.
(408) 431-1000
(800) 457-9527
Fax: (831) 431-4142
www.borland.com

FEATURE ARTICLE

David Brooks

Embedded OSs for Internet Appliances

When you're looking at embedded OSs for your Internet-capable product, you've got a lot of pros and cons to think about. David goes through all the points here so when the time comes, you'll be ready to figure it out on your own!



Internet appliances are dedicated devices that use the Internet or another IP network as a communication backbone. They are built around the open standards of the Internet and are generally targeted at doing one or two things incredibly well.

Internet appliances are not general-purpose devices. Sure, you can use a full-size oven to toast your bread, but for many years now, people have understood the benefits of a special-purpose device with two slots in the top, heating elements on both sides, and a timer.

Likewise, you can use a PC to do anything on the Internet. But just like using a full-sized oven for your toast, a PC is sometimes overkill. Industry analysts are predicting that by 2002 the unit shipments of Internet appliances will exceed those of PCs.

That's not to say that they will replace PCs. They will either be adopted by users who don't already have PCs or as companion devices for people who already own one. Just as families have more than one telephone or more than one television, people will buy more than one device to access the Internet.

These devices are already showing up in the marketplace today. Web TV is probably the best-known Internet appliance, but there are many more. Applio sells an Internet phone appliance that uses your regular phone line to bypass long-distance phone charges, and Diamond Multimedia sells a portable music device that plays digital music downloaded from the 'Net.

Appliances are not only personal devices. Dedicated web servers, Internet voicemail systems, and a new device from HP called a Digital Sender hope to replace general-purpose servers, expensive corporate voicemail systems, and fax machines. Handheld computers and cell phones have even started adding Internet capabilities.

WHICH OS?

Everyone is familiar with Microsoft Windows and various versions of Unix, but in the Internet-appliance market, it's still a close race. In the handheld device category, 3Com's PalmOS has out-shipped every other device in the U.S. and Psion/Symbian's Epoch has a strong hold in Europe and the Internet-enabled cell-phone market. Microsoft is investing heavily in this area and, in some respects, provides clear advantages over the established players.

The market leader in the classic embedded-systems race is Wind River Systems. Their VxWorks OS not only powers most of the HP LaserJet printers but is also embedded in many industrial devices that require rock-solid reliability. Table 1 contains a selection of products and vendors.

Now that you've decided to design and build an Internet appliance, how do you choose the OS? The decision largely depends on the intended deployment of the device. If you're trying to produce a proof-of-concept lab prototype that will never be produced in high volume, then your decision will be different from a team that's developing the next great consumer product.

COST CONCERNS

In the real world, cost can be one of the largest concerns. Few of us have the luxury of working for companies with unlimited research and development budgets.

Unlike buying a shrink-wrapped software package, the real costs of selecting an OS can be fraught with potential pitfalls that will come back to haunt you if you don't make a good choice. I learned some of these lessons the hard way, after spending millions of dollars in a now defunct startup company.

The up-front costs of the development environment can vary from free to hundreds of thousands of dollars. Although most desktop software engineers are accustomed to buying a visual integrated development environment (IDE) for a few hundred dollars, embedded-systems developers have to spend tens of thousands of dollars for remote debuggers.

Be careful to select a development environment that provides the compiler tool chain appropriate for your target platform and a full-featured debugging environment. The IDE should also run on the same development systems that your engineers use to write their code.

If they're using Unix workstations for their development but your IDE only runs under Windows, then factor in the cost of buying new hardware. If your engineers work with ICEs, make sure the IDE software is compatible with those as well.

Source-code license costs can be a huge swing factor in the up-front costs. Some companies offer source code for free; some provide it only for the drivers and boot code.

Most of the embedded OS companies will sell the source but at a fairly steep cost. If your application requires you to modify the IP stacks or kernel OS, you'll most likely need source code for at least some portion of the OS, so budget accordingly.

The other up-front cost that is most often overlooked is the cost impact to your development partners. If a portion of the product is being created by a third party and they don't already have the development tools and expertise, you end up paying for their expense as well.

The per-unit royalty is where most OS vendors like to make their money. Microsoft, for example, provides the IDE software for a low up-front cost, but it counts on large unit volumes to make their return. Other vendors offer

a reasonable onetime buyout that enables you to eliminate the per-unit charge.

The embedded RTOS vendors have royalty schedules that have sharp volume-based pricing structures. These vendors also like to have these volumes committed to up front, requiring you to put out tens of thousands of dollars to get the per-unit rate down into the tens-of-dollars range.

If you're producing a consumer product that sells for about \$200, your per-unit software budget may not bear the burden of a \$20 royalty for the OS. But, you may have to commit to a hundred thousand units to drive down the cost.

On-going maintenance costs of the software are certainly not important for proof-of-concept prototypes, but for most real volume products, companies can spend as much on yearly maintenance fees as they do on the up-front purchase of the IDE.

When it comes to time-to-market concerns, keep in mind that Internet appliances are produced on incredibly tight schedules. Engineers working in this timeframe like to talk about "Internet time," where things happen three times as fast as other product categories. Specifications and product needs change quickly, and sometimes getting a product out fast is more important than its cost.

It may sound like an obvious point, but embedded OSs have a lot of components. The bottom line: the more features provided as part of the integrated platform, the faster you are likely to hit the market with a real product.

The ideal situation is that the OS provider has an à la carte selection of pieces and parts that provides almost everything you need to build the final product. You have a much better chance that everything will work together when that's the case. However, that's not always true.

Some vendors provide components for multiple-target CPUs, and not all components are available for all CPU choices at the same point in time. If you're working with a relatively new CPU family or a new chip of an existing family, check with the OS provider and explicitly ask if each of the com-

OS	Company
Windows CE	Microsoft
VxWorks	WindRiver Systems
pSOS	Integrated Systems
VRTX	Mentor Graphics
Epoch32	Symbian
Linux	Linux
Free BSD	FreeBSD
Nucleus	Accelerated Technology
SMX	MicroDigital

Table 1—These operating systems drive many of the embedded devices sold in the world today. See the Sources for additional contact information.

ponents you need is available now for your chosen CPU.

For example, Windows CE runs on some members of the ARM family but not all. So don't just ask if it runs on ARM-family CPUs. Even if the OS and components are available on your target CPU, they may be newly released on that platform. Unless you have the stomach for it, don't be the lead customer for a new port of the OS.

Even if your OS vendor doesn't provide all the components you need, you may still be able to acquire them from other vendors or open-source providers. All of the caveats discussed for the OS vendor-supplied components are even more important when dealing with third-party software.

Most third-party vendors focus on a core competency such as IrDA stacks or audio CODEC software and try to support multiple OS platforms to increase their potential business. Verify that they have done the software on your version of the OS with your target CPU before making your commitment.

Open source is a dream come true for some appliance products. "No up-front costs and no per-unit royalty" has a nice ring to it. Be aware that "free" means no support and no guarantees. Many shareware packages require porting to other platforms that, in many cases, will cost more than the purchase price from a vendor who's willing to support it.

If your team is already familiar with the OS and tools, consider yourself lucky. If you are building a team, you might want to check with your local recruiting resources to find out how easy it will be to hire engineers familiar with your OS. You may suffer a significant delay getting to mar-

Protocol	Function
TCP	Reliable transport
UDP	Lightweight transport
DNS	Name to IP address
SNMP	Network management
DHCP	Dynamic IP address allocation
FTP	File transfer
BOOTP	Network boot protocol
SNTP	Time of day

Table 2—A handful of protocols make up the core of the network stacks of Internet-connected devices. Make sure that the embedded OS you choose provides these capabilities as a minimum set.

ket if you can't find qualified people to do the work.

If your team is in place but the OS is new to your organization, invest in the training. It may be expensive, but it will save time in the long run.

Debugging tools play a major role in time-to-market considerations. Make sure they provide your engineers with all the tools needed to debug at the highest level of source code possible.

If you're dealing with a real-time communications system, make sure that real-time profiling software is available. Inserting `printf` statements can change the characteristics of the program significantly and even make the problem go away. The Heisenberg uncertainty principle applies to software as much as nuclear physics.

FUNCTIONALITY

The exact set of components that make up an OS is being debated both in the industry journals as well as the courts. For the purposes of this article, there are three major areas that must be considered—network and communications stacks, memory and task management, and GUIs.

All Internet appliances share two major portions of an OS—namely, the communication stacks and memory and task management. There are classes of devices that have no display or limited display needs that don't require a complete GUI.

COMMUNICATIONS STACKS

There is a whole host of acronyms that make up a typical Internet appliance, but the foundation of all these protocols is simply the Internet Protocol (IP). With the number of devices attached to the Internet increasing, a

new version of IP (called IPv6) was needed to provide for the larger address space required. For products that will be deployed in the next year, supporting IPv6 isn't necessary, but quickly thereafter it may be required for many types of products.

An explanation of all the IP stack protocols required could fill an entire book. The core set is listed in Table 2 with a brief explanation of their functions. Although all of these aren't required for every Internet appliance, if you choose an OS that supports them all, you should be in good shape.

The API used to interface with these stacks is an important feature. For example, in the PC world, WinSock is the API that most developers expect to program to. On Unix OS variations, you'll find a different API—most likely, Berkeley sockets.

Applications that need IP communications capabilities are written for a specific API. If you choose an OS that doesn't support the API expected by your application, you can spend a lot of time and money to port the application. Communications stacks can be purchased separately, but given the level of importance, it's better to choose an OS that has support already integrated.

MEMORY AND TASK MANAGEMENT

The heart of the OS is the memory and task management function. This is the kernel of the OS and will greatly impact the application capabilities of the end product.

One critical decision concerns whether the OS must be hard real time or not. There's much debate in the industry, driven mostly by vendors whose products are not hard real time, about what hard real time means.

The commonly accepted measure of this capability is called determinism. If your OS can guarantee that an interrupt will be serviced and a critical task launched with all of the resources it needs in a fixed length of time, then it is deterministic.

There aren't many Internet appliances that strictly need deterministic performance. Even full-motion video devices can get by with fast but not deterministic performance. Be sure to understand the requirement of your

device and factor in this choice before any other decisions are made.

The process and memory models supported may severely impact your ability to quickly port applications written on other platforms. Open-source software written for the PC may not port to embedded RTOS platforms as easily as you expect.

Some RTOS choices don't support virtual memory concepts, reentrancy of applications, or multiple name spaces. This setup can impose minor difficulties such as not being able to simultaneously run multiple instances of the same application or making your programmers reserve variable names because all of them must be unique. The name-space issue further impedes your ability to port open-source software.

GUI

The GUI is the part of the OS that may not be needed in some Internet appliances. If it is needed, however, it quickly drives the choice of OS providers to a small group. The two major GUI standards in the world are MS Windows and X Windows.

Many RTOS vendors support a version of X Windows for their systems, and many open-source applications targeted for Linux support this API. Note that this software can be quite large and can require large graphics libraries to work as expected. Also, even though the X Windows API set is supported, the underlying process and memory architecture may make it painful to port an open-source application to the RTOS environment.

The MS Windows graphical API set is supported by Windows CE and provides the single most compelling reason to use CE in an Internet appliance. Open-source packages exist that provide a bridge from MS Windows to other GUI API sets. The most notable is Twin from Willows. Like most open-source packages, support is spotty at best.

Two other emerging GUI APIs may work for your product. If the GUI can be implemented in an HTML browser, then the development team can concentrate on buying or porting a single application (the browser) and implement everything else as web pages.

Simple interactive capabilities can be provided if ECMA Script (JavaScript) is supported. This arrangement is no substitute for a full API, but embedded browsers are available for many platforms.

Java is the other standard API that's widely supported. Providing a Java virtual machine on your Internet appliance carries additional risks and resource requirements, but it also opens up the portability aspect of Internet computing that may be appropriate for your market segment.

LONG-TERM ISSUES

If the product is developed on time and under budget, and if the sales group is able to find homes for the products, then you eventually have to worry about supporting the product for several years.

Your chances of supporting the product improve if the OS vendor is committed to the product and will be in business for awhile. You mitigate the risk by having all the source code, but you're better off letting the experts take care of it for you.

Most Internet-appliance companies won't be able to keep up with the ever-expanding data formats and communications protocols being invented every day on the 'Net. Even if the first product is simply the OS and your magic sauce, eventually you'll need software from a third party. Selecting a widely supported platform enables you to keep up with your competitors when they add a protocol to their product.

The last long-term impact of the OS choice is your ability to retain the staff that developed your product. Engineers are in high demand. If they feel they aren't being challenged or learning a marketable skill, they'll go elsewhere. Choosing a popular OS will make them feel that the skills they learn at your company will keep them marketable.

THE CHOICE IS YOURS

Choosing the proper OS for your Internet-appliance project can be tough. Taking the time to consider all the issues may take a bit longer, but in the end it can save you a lot of money and many headaches. ☹

David Brooks is the general manager of the handheld products division of inViso. Prior to joining inViso, he was the president and CEO of WebSonic. You may reach David at dbrooks@inviso.com.

SOURCES

WindowsCE

Microsoft Corp.
(206) 882-8080
Fax: (206) 936-7329
www.microsoft.com

VxWorks

Wind River Systems
(510) 748-4100
Fax: (510) 749-2010
www.wrs.com

pSOS

Integrated Systems, Inc.
(408) 542-1500
Fax: (408) 542-1956
www.isi.com

VRTX

Mentor Graphics
(408) 436-1500
Fax: (408) 436-1501
www.mentor.com

Epoch32

Symbian, Inc.
(650) 598-4747
Fax: (650) 598-0231
www.symbian.com

Linux

www.linux.org

FreeBSD

www.freebsd.org

Nucleus

Accelerated Technology, Inc.
(334) 661-5770
Fax: (334) 661-5788
www.atinucleus.com

PalmOS

3Com
(408) 326-5000
Fax: (408) 326-5001
www.palm.com

SMX

MicroDigital, Inc.
(714) 373-6862
Fax: (714) 891-2363
www.smxinfo.com

FEATURE ARTICLE

John Luo

Compact Optical Image Scanner

Mobile communications systems are big among people who take the office on the road. These systems have to be lightweight and low power. That rules out scanners, right? Wrong. The details of John's compact scanner are right here.



Mobile communication systems are becoming more popular and the choice of many people who need to take their offices with them when they travel. With the advances in technology and lower price tag, mobile communication systems such as the PDA, mini-notebook, and palm-top computers are replacing many traditional notebooks and calculators in the hands of students, executives, and professionals.

Because of the limited size, such mobile devices are still inadequate in many applications. There's a growing demand for accessories and peripherals for the mobile communication systems.

What kind of peripherals are ideal input devices for such small computers? Even if you're a good typist, you'd feel the inconvenience of the small key-

board on a handheld PC, and hand-writing input on a PDA is even slower.

Try typing business-card information into a PDA and you may start thinking about a better solution for the input device. Perhaps, a scanner with optical character recognition (OCR) software?

Today's desktop scanners have good performance and low prices, but they're big, heavy, and need a 110-V power source. There are some handheld scanners, but most have power supplies bigger than palm PCs and come with a parallel port or USB, which palm devices don't have.

Even worse, you still need to get power from an outlet. You might be able to find a battery-powered scanner, but they are still big, heavy, expensive, and have no interface to the PDAs.

Is there any way to make a small, low-power consumption, and low-cost scanner? Yes.

SYSTEM DESIGN

From my system-design considerations, a lightweight and low-power consumption device must be the top priority. If it's going to be lightweight, any external power source such as battery pack is out of the question.

Because the device is intended as a companion for a mobile system, a hookup to a power outlet isn't practical. The most ideal power source is the host system, and it must provide enough power to drive this device.

Most mobile computers have a serial port for data exchange. This is mainly because of its limited space and weight consideration. The output pins of this port can be used as a power supply (a typical example of this technique is the mouse).

Because the output pins can only provide several milliamps of current, you need to carefully select the low-power parts. For a scanner, the basic components include:

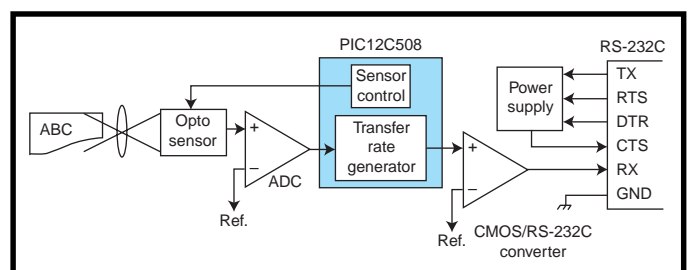


Figure 1—The captured image is digitized by the ADC and packed into the RS-232 serial datastream.

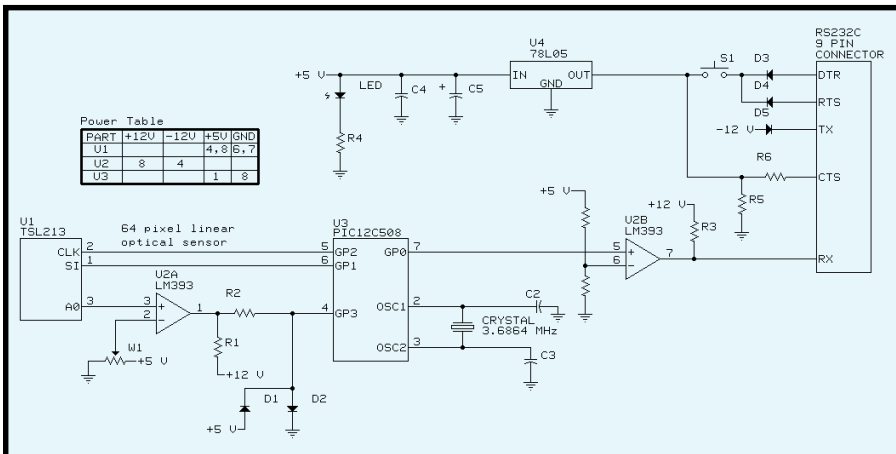


Figure 2—The system consists of three chips and some passive components. The DTR and RTS pins on the host are set to logic 0 to provide +12-V power to the circuit.

- optical linear sensor array as the image input device
- ADC to convert the analog signals to digital data
- system control unit to trigger the sensor and generate serial data
- COM/RS-232C adapter to interface to the PC
- voltage regulator to provide +5-V power

Most communication systems need a transfer buffer before sending out the data. I selected a high transmission bit rate to eliminate the buffer.

For an OCR system, monochrome image is good enough. When a scanner moves on a piece of paper, an array of optical sensors picks up the image signals and saves them into the registers. Usually, analog signals are converted into 8-bit digital data (for 256 grayscale).

Before the image is translated to characters, the microcontroller or PC system converts the 256 gray levels into black and white levels (one bit per pixel). Here, I directly converted the analog signal to one-bit data to minimize the micro's calculations.

A single line of text scanning takes 1–2 s. The image is about ¼" tall and 7" wide. For the resolution of 200–300 dpi (high enough for the OCR system), the image is about 50 × 1400 to 75 × 2100 pixels.

Because I used one binary digit to present each pixel, I need to send 7000–15,750 bytes of data to the serial port for a line of text in 1–2 s. For asynchronous serial transmission, each byte is accompanied by a start bit and a stop

bit; one line of text generates 70,000–157,500 bits. I selected the 57.6 kbps transfer rate for an acceptable scanning speed and an affordable cost of the microcontroller.

HARDWARE DESIGN

Parts selection is critical because of the limited power supply (see Figure 1). Initially, I chose a TSL1401 128-pixel linear sensor array, which costs \$4.43 each and needs a 4-mA supply current (too much), as the image-input device. An alternative is a TSL1401 128-pixel sensor array. It costs \$4.43 and needs only 2.5 mA, but the microcontroller needs more time to acquire the data.

Because the peak responsivity of the sensor is from 600 to 880 nm, I used a red LED (660 nm) to illuminate the intended target (the document).

For the one-bit ADC, I chose an LM393 dual-voltage comparator (\$0.39 each) which needs less than 1-mA supply current and 1.3-μs response time. I used half of the LM393 as the ADC.

The reference voltage that corresponds to the luminance is adjusted by a voltmeter. The other half of the LM393 is used as a CMOS-to-RS-232C driver that eliminates another standard interface chip.

I chose a PIC12C508/509 because it has an eight-pin package, has six I/O pins, and needs 1.8-mA supply current. The 4-MHz clock and four clock cycles per instruction give me enough time to drive the sensor and generate 57.6 kbps of serial data. The PIC costs \$9.65 for the EPROM version and \$1.88 for the OTP version.

DESIGN OUTLINES

The efficiency of the input method is key to the speed of the image process and the performance of the device. The most ideal medium for image input would be a black object with a white or light-color background. This arrangement provides the biggest degree of contrast in the intended object and the background.

But in real life, we live in a colorful world, which introduces some difficulties into this project. It would cost more and the design would have to be greatly enhanced to accommodate all the possibilities of different image environments and media. To simplify this project, I'll introduce a design with simple dark object and light color background as the image input medium.

Let's assume a simple document with black (or dark) text printed on top of a white (or light) background paper. To drive the optical sensor, you need two signals—a serial input (SI) and a clock (CLK). The system logic schematic is shown in Figure 2.

SI defines the start of the data-out sequence and CLK controls the charge transfer and pixel output. The LED illuminates the text document. After the sensor gets an SI signal from the microcontroller, it captures an image frame of 128 pixels. It sends 1 pixel out through the analog output (AO) pin each time it gets a CLK pulse.

The sequence of analog output voltage is arranged from 0 to 2 V and corresponds with the sensor's 128 optical diodes. After the pixel values are sent out, a new SI pulse resets the sensor.

All the senses we encounter in life are basically analog. The sounds we hear are in terms of pitch. The images we see are colors in different depths of darkness and mixtures. It's the same way when you first scan the image into

Pin name	Function
GP0	Serial data output
GP1	SI output
GP2	CLK output
GP3	Captured data input
GP4/OSC2	Crystal
GP5/OSC1	Crystal

Table 1—Here you see the I/O pin assignment of the '12C508. No extra pins are required, and none are wanted.

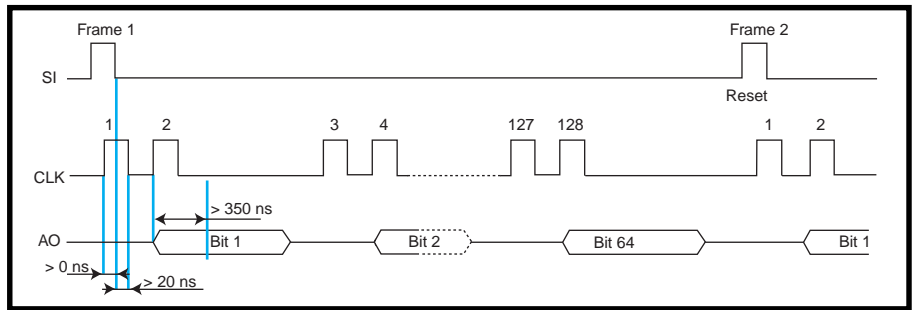


Figure 3—The waveform shows the relationship among the sensor trigger signals (SI and CLK) and the analog output (AO). For correct data acquisition, the 1.3- μ s delay of A/D conversion must be considered.

the device. The information must be converted into digital values (i.e., 1s and 0s) before it can be processed.

To translate the signal, you must first identify the input value. One of the comparators in the LM393 compares the value it gets from the sensor with a reference voltage. If the sensor output is higher than the reference, the comparator sends out a high-level voltage that presents a one-digit number of 1, and vice versa.

The voltmeter can adjust the reference voltage to control the image luminance. Because the comparator uses high-voltage power supply (+12 V and -12 V), two diodes are used to force the ADC output into the range of 0 to +5 V.

Another comparator in the LM393 is used as a CMOS-to-RS-232C converter. High-voltage power supply is required for generating an RS-232C standard signal level (-3 to -5 V for low level and +3 to +15 V for high level). Keeping in mind that the '12C508/509's I/O maximum output low voltage is 0.6 V, the reference voltage of the comparator is set to be at about 0.8 V.

The I/O pin configurations are shown Table 1. For a 57.6-kbps data transmission, I need a crystal-stabilized oscillator as a timing reference. The frequency of the crystal should be divisible by 57,600. I selected the highest frequency possible (3.6864 MHz) so I could fit the maximum number of instruction cycles into a single-bit data transfer period.

POWER SUPPLY

The power source and the power consumption were major concerns from the start. This device is meant to be a convenient companion for portable computers. Therefore, its power source must be integrated into the overall system design.

I explored the possibility of drawing the power from the host device. When DTR and RTS signals are tied to logic 0 on a standard RS-232C socket, the serial port of a portable computer can generate +9 to +12 V for external use. The power supply for the compact optical image scan-

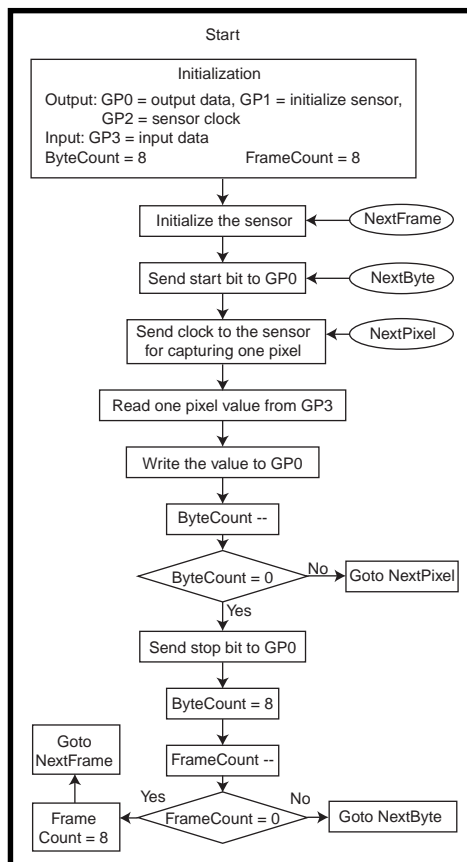


Figure 4—The program in the '12C508/509 manages the sensor triggering, data collection, and serial data generation. Every flow of commands should be finished within 16 '12C508 instruction cycles.

ner comes from the RS-232C serial port of the host device.

The DTR and RTS signals are always set as 0, which generates +9 to +12 V on the RS-232C socket. I used them as the positive power source. Because there is no data output from the TxD pin of the host, its output is -9 to -12 V and I used it as the negative power source.

The LM393 comparator uses positive and negative power sources to generate an RS-232C signal level. A 78L05 regulator converts the +9 to +12 V to +5 V to drive the rest of the components.

A switch turns the +12 V and +5 V on and off. Once the user presses the push button down, the scanner starts to capture the image. The CTS pin of the RS-232C is pulled up to +12 V to tell the host the scanner is working.

When the user finishes scanning a line of text and releases the button, the scanner shuts off and the CTS pin drops to -12 V to tell the host the scanning is finished. Now, the RxD pin of the RS-232C (output of CMOS-to-RS-232C interface) drops down to -12 V to present the rest bits for the last outgoing image byte and a stop bit (logic 1).

SOFTWARE DESIGN

The TSL1401 sensor contains 128 optical diodes. My design has a 300-dpi resolution, which is better than an OCR-recognizable image. The height of a character in a document is usually less than 1/5", so the pixels for the character height are:

$$\frac{300}{5} = 60 \text{ pixels}$$

I selected 64 pixels (half the resolution of the sensor). In the program, I skipped every other pixel to get the corresponding 64 bits data (I call it a frame).

A frame is separated into eight groups (bytes) for serial transfer. I declared a byte counter and a frame counter to control the transfer. The default values are eight for both counters.

The microcontroller continuously sends SI and CLK pulses to the sensor

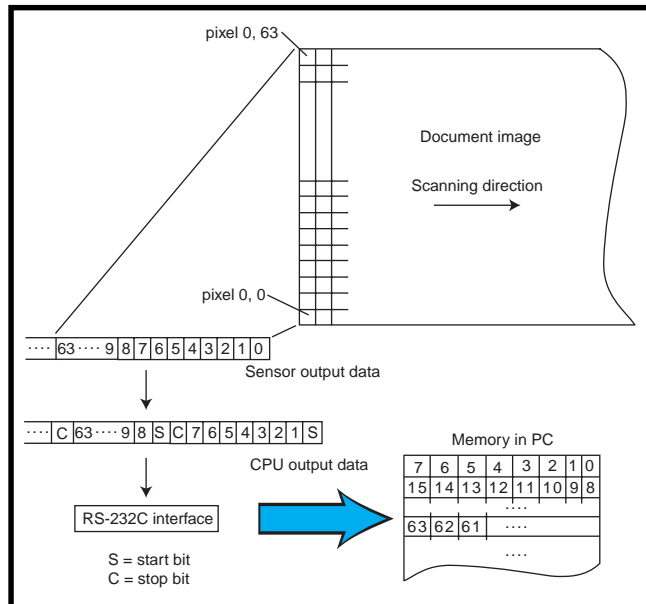


Figure 5—The pixels are converted into a single-digit bitstream. A start and a stop bit are added for each group of 8 bits. Each group of 8 bits is then saved into one byte of PC memory.

and gets the image data from the AO pin of the sensor.

Each CLK signal corresponds to one pixel, and each SI signal corresponds to one frame of the pixels. As you can see in Figure 3, the microcontroller gets the 64 even values and ignores the other 64 odd values.

In each frame of pixels, the signal for the first pixel is treated a little differently than the other pixels. The CLK's rising edge for the first pixel must happen after SI's rising edge and before SI's falling edge. Otherwise, you won't get the right sensor output values. The timing relationship between SI and CLK is also shown in Figure 3.

For asynchronous serial data transfer, I added one start byte (logic 0) at the beginning of each byte and one stop bit (logic 1) at the end of each byte. Because the CMOS-to-RS-232C converter isn't an invert comparator (not the same as a standard RS-232C driver), all of the bits are inverted in the CPU before being sent out.

In the 57.6-kbps transmission process, each bit takes exactly 16 CPU instruction cycles (under 3.6864-MHz clock frequency). Within the 16-cycle period, I need to trigger the sensor, get the data corresponding to one pixel, and send the data to the serial port. Even though a longer time for stop bits doesn't hurt, I still finished the counters and sensor reset in 16 cycles

to ignore the gaps between bytes and frames (see Figure 4).

Figure 5 shows that pixel 0 is the first bit sent to the PC and is stored in the least significant bit of one byte in PC memory. There are two ways the PC can test the end of the picture—by checking the CTS status or by checking the gaps between the received bytes. A gap indicates the end of the receiving image.

READY TO GROW

The trend in the peripheral design world is moving toward a higher integration of multi-functions. The functions on the compact optical scanner are limited now, but ideas for future upgrades include a

USB interface to help with higher data transfer and power supply, a variable scan-size control to provide a more efficient scan of different font sizes, a multilingual scan capability, and an independent dictionary function. ☐

John Luo is a senior engineer for ESS Technology. He specializes in the area of video-compression and communication technologies. You may reach him at john.luo@esstech.com.

SOFTWARE

Source code may be downloaded via the Circuit Cellar web site.

SOURCES

PIC12C508/509
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

TSL1401
Texas Instruments
(972) 644-5580
Fax: (972) 780-7800
www.ti.com

LM393, 78L05
National Semiconductor
(800) 272-9959
(408) 721-5000
Fax: (408) 739-9803
www.national.com

MICRO SERIES

Jim Lyle

USB Primer Classes and Drivers



One of USB's earliest and most important goals was to make it easy to use. It

has to be easy because the computer marketplace is rapidly expanding to include increasingly less-technical users.

These users don't know what an interrupt or DMA channel is, let alone how to finesse them into a working configuration. Nor should they have to. Even highly technical users are tiring of the difficulties involved in configuring or upgrading their computers.

From my perspective, it's not that difficult to install an ISA or PCI card. I've been doing this for years and I know how to set the jumpers (plug-and-play usually takes care of it anyway). I rarely get the cables on backwards anymore or offset by one row of pins, either.

But, one part of the process still strikes fear into my heart. One part of the installation never goes quite the way the instructions claim (when I finally do get around to reading them). There's one element that rarely fails to "blue screen" the machine repeatedly and strangely:

THE DRIVER

I've spent days trying to install the drivers for a seemingly simple device. Sometimes, it's incompatibilities with other drivers or software. Sometimes, the driver wasn't

tested well or has a bug and needs a patch or upgrade. Sometimes I never do find the problem.

Wouldn't it be nice if all the drivers you ever needed came with the OS? You'd just plug something in and it would work. No more installation headaches; no problems moving from one machine to the next or even from one type of machine to another (e.g., from PC to Mac to Linux to workstation). There would be reduced disk and memory requirements, too, and one-stop shopping for upgrades. Overall, compatibility and reliability would improve dramatically.

Developers would find tremendous advantages as well, bringing more products to more platforms in less time and with less effort. Adding USB would no longer require the expertise (and the time, often in the critical path) needed to write drivers. Testing and support requirements would be reduced, and so would the overall project risk.

There are thousands of different kinds of devices already, and more are on the way. An OS can't possibly provide all the drivers for all of these types of devices. Or can it?

Although lots of different products are or will be available, many of them have more similarities than differences. In some cases, identical devices are produced by different manufacturers. In other cases, the products are different but the functions are similar.

Consider mice, track balls, and touchpads. They are physically different (and there's variation even within those broad categories), but the overall function is the same—moving a cursor on the screen. They all provide an *x* and *y* displacement and two or more buttons (or the equivalent).

What about full-page scanners, hand scanners, digital still cameras,

Part
of
2
of
4

Now that we have some of the USB

basics from Part 1, we're raring to go with USB! Jim wonders if an OS can provide all the drivers for the many devices there are today. With USB classes, he explains, it's entirely possible.

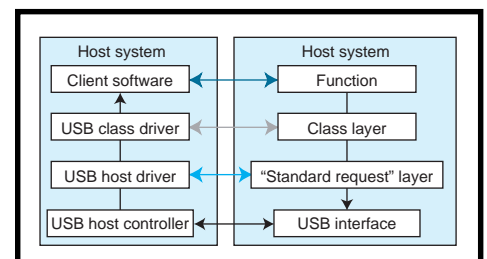


Figure 1—USB uses well-defined protocol layers to reduce complexity and improve standardization.

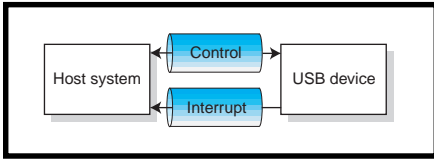


Figure 2—USB devices use logical “pipes” to transfer information. This device uses two.

and slow-scan video cameras? All produce an image of some form. Or printers? Color or black-and-white, laser or inkjet, Postscript or not—all put an image on paper.

With a little insight and forethought, most devices can be grouped into fewer categories, each with a common purpose and set of requirements. Then, it’s possible to define a common API for each category and therefore the requirements of a single, generic driver suitable for use with any of the devices in that group. USB is trying to accomplish exactly this by defining a variety of device classes.

The USB specification defines the mechanical and electrical requirements for all USB devices as well as the fundamental protocols and mechanisms used to configure the device and transport data. The class definitions are add-on documents that refine the basic mechanisms and use them to establish the class-specific blueprint for both the device and the generic driver.

There will always be unique devices as well as manufacturers that choose to differentiate their product from the competition within the driver. For these cases, vendor-specific drivers will always be necessary.

But for most products, it’ll be possible to use generic drivers that are part of (or included with) the OS. That’s one of the most important advantages of USB.

Comm, Printer, Image, Mass Storage, Audio, and HID (human interface device) are a few of the defined USB classes. Some devices may fit into more than one category.

For example, there are combination printers/scanners. Although physically this is one device, logically it is two. Part of the device fits into the Printer class and uses that generic driver. Part of it fits into the Image class and uses that driver. Devices in more than one class are called compound devices.

DEVICE CLASSES

Windows 98 includes many but not all of the USB class drivers. This situation is unfortunate, but it couldn’t be helped because some of the class definitions weren’t finished in time (some still aren’t complete).

Future releases and service packs will add additional class drivers until most or all of them are available and supported. Apple and Sun Microsystems also have class driver implementations available or underway for their respective platforms.

As the name implies, HIDs are designed for some kind of human input or output. The most common examples are keyboards, pointer devices like mice, and game controller devices such as joysticks and gamepads.

This class also includes things like front panels or keypads (e.g., on a telephone or a VCR remote control), display panels or lights, as well as tactile and audible feedback mechanisms—essentially, anything you might press, twist, step on, measure, move, read, feel, or even hear.

Seemingly, this class would include almost anything connected to a computer, but it doesn’t. Its primary purpose is control. Although it’s very flexible, this class definition doesn’t handle large amounts of data well. It doesn’t need to; other device classes can better serve that purpose.

In a USB speaker, for example, the volume, tone, and other controls fall well within the HID class. But, the sound channels are data intensive, so they are better handled by the Audio class. In fact, many products in the other classes are compound devices with HID handling the controls.

Given the diversity of USB applications in general and HID devices in particular, how can any one driver hope to do all the things required by its class? The first part of the answer comes from the physical interface. There’s only one! All USB devices communicate with the host via their USB port.

This sounds self-evident, but the implications are tremendous. The USB port works according to the same basic principles for all devices, in all modes of operation. The class driver never needs to worry or know about

ISA or PCI buses, SCSI, IDE, or ATAPI interfaces, serial ports, parallel ports, keyboard ports, mouse ports, game ports, or anything else for that matter.

The class driver doesn’t even need to know much about USB ports. Even that physical interface is abstracted and managed by the USB Host driver. This abstraction, or layering, is another key concept that makes class drivers possible.

Each layer has its own responsibilities and it uses APIs provided by the lower levels to accomplish them. It doesn’t need to know how the lower levels work or which ones are present.

Figure 1 shows a simplified view of the various protocol layers that might be present for a USB device. Note that there are connections at all levels, but most of these are logical.

The single physical connection is between the USB host controller and the device’s USB interface and is at the lowest level (shown in black). This layer is the hardware—the cables, connectors, and state machines.

The first layer of software, which is required in all cases (in light blue), is the USB host driver on the computer. On the USB device it is the essential firmware that manages the hardware and provides the standard requests (also called “chapter 9” requests because they are in that chapter of the specification). There’s a logical connection between these layers for configuring and controlling the USB interface.

The device driver layer comes next (shown in grey) and is usually the class driver(s) on the host side and the corresponding firmware on the device side. The logical connection at this level carries class-specific commands and requests, although these often use protocols modeled after those in the layer below.

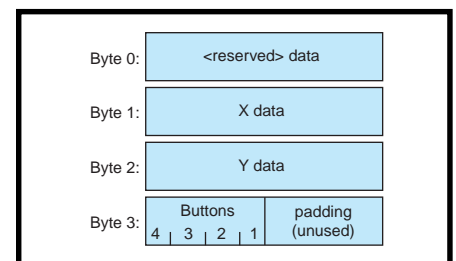


Figure 3—This sample report for a USB joystick shows you one possible data organization.

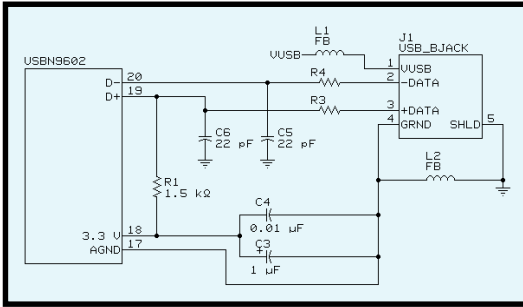


Figure 4—This schematic shows you a typical connection between the USBN9602 and the USB connector (or cable).

The top (dark blue) layer is the one the user sees and cares about. For example, the client software on the host might be a flight simulator and the associated function might be a joystick. At this layer, the only thing the client software (and user) cares about are the joystick inputs. It doesn't care (and doesn't need to know) how those inputs are read, packaged, and transported.

PROTOCOL LAYERS

To communicate with a USB device, the host software opens up a series of pipes, and uses them to transport data. The pipes correspond to hardware endpoints, which are individual channels, usually with dedicated buffers or FIFOs.

Pure HID devices use only two pipes (see Figure 2). The control (default) pipe, required by all USB devices, is used for receiving and responding to specific requests or commands. The standard requests use this pipe, and many of the class definitions (including HID) add class-specific requests.

The interrupt pipe sends asynchronous data to the host. This pipe is poorly named; USB doesn't support true interrupts but rather enables the device to predefine a maximum poll interval. This way, if a key is pressed, the mouse moved, or the joystick steered, the device can report in a timely fashion without a specific request (from the driver) to do so.

The HID class driver starts with the physical/standard request API common to all USB devices and adds the HID standard pipe structure and command superset. The difference from one HID device to another is the data it returns and what the data means.

HID data is packaged into structures called reports. Figure 3 shows a sample

report for a joystick. It's simple and composed of four bytes.

The first byte is unused here but is reserved for a throttle position on another model. The second and third bytes are the x and y coordinates, respectively. The fourth byte contains information about the four buttons (one button per bit, with four unused bits that are zero-filled to pad out the byte).

This is just one example for one joystick. Other HID devices have different report structures. Other joysticks may have other structures, too. Some may order the data differently or have additional functions and capabilities (e.g., force-feedback).

SAMPLE REPORT

Obviously, the HID class driver can't keep report maps for all possible implementations of all possible devices. The device has to be able to describe the report to the class driver. This too

is in keeping with standard USB mechanisms.

USB devices use predefined data structures called descriptors to describe their identification, capabilities, requirements, and protocols. The USB spec defines device and configuration descriptors that must be provided by all devices. The HID class definition adds information to these and goes on to define a report descriptor.

The report descriptor provides the map that the HID class driver needs to understand and interpret the report. The structure of the report descriptor is complex, though flexible. Fortunately, it doesn't complicate the device-side firmware because it is a data structure that can be written and compiled externally and then remain constant.

Listing 1 shows a sample report descriptor. The details are beyond the scope of this article, but note that it defines the type of application, size, maximum and minimum values, and subtypes of the various report fields.

Listing 1—This *ReportDescriptor* function corresponds to Figure 1. USB devices use descriptors to describe themselves to the host PC.

```
unsigned char ReportDescriptor[59] = {
0x05, 0x01,      /* USAGE_PAGE (Generic Desktop) */
0x15, 0x00,      /* LOGICAL_MINIMUM (0) */
0x09, 0x04,      /* USAGE (Joystick) */
0xa1, 0x01,      /* COLLECTION (Application) */
0x15, 0x00,      /* LOGICAL_MINIMUM (0) */
0x26, 0xff, 0x00, /* LOGICAL_MAXIMUM (255) */
0x75, 0x08,      /* REPORT_SIZE (8) */
0x95, 0x01,      /* REPORT_COUNT (1) */
0x81, 0x03,      /* INPUT (Cnst,Var,Abs) */
0x05, 0x01,      /* USAGE_PAGE (Generic Desktop) */
0x09, 0x01,      /* USAGE (Pointer) */
0xa1, 0x00,      /* COLLECTION (Physical) */
0x09, 0x30,      /* USAGE (X) */
0x09, 0x31,      /* USAGE (Y) */
0x95, 0x02,      /* REPORT_COUNT (2) */
0x81, 0x02,      /* INPUT (Data,Var,Abs) */
0xc0,            /* END_COLLECTION */
0x15, 0x00,      /* LOGICAL_MINIMUM (0) */
0x25, 0x01,      /* LOGICAL_MAXIMUM (1) */
0x75, 0x01,      /* REPORT_SIZE (1) */
0x95, 0x04,      /* REPORT_COUNT (4) */
0x81, 0x03,      /* INPUT (Cnst,Var,Abs) */
0x05, 0x09,      /* USAGE_PAGE (Button) */
0x19, 0x01,      /* USAGE_MINIMUM (Button 1) */
0x29, 0x04,      /* USAGE_MAXIMUM (Button 4) */
0x55, 0x00,      /* UNIT_EXPONENT (0) */
0x65, 0x00,      /* UNIT (None) */
0x95, 0x04,      /* REPORT_COUNT (4) */
0x81, 0x02,      /* INPUT (Data,Var,Abs) */
0xc0            /* END_COLLECTION */
};
```

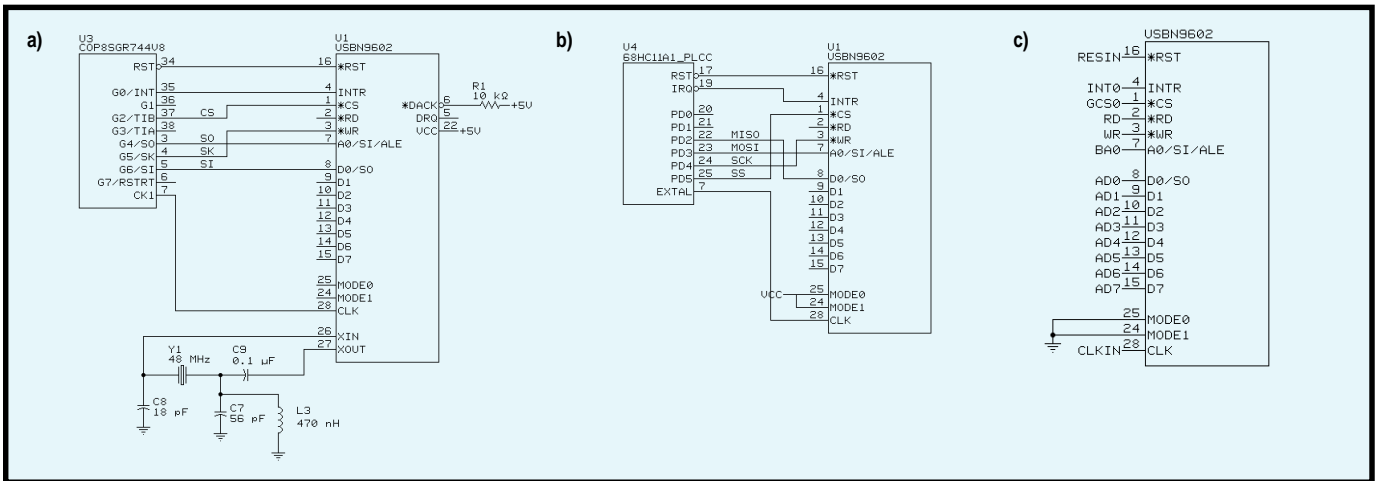


Figure 5—This schematic shows a serial interface between the USBN9602 and a COP8 microcontroller. It also shows the oscillator circuit. **b**—Here's another serial interface. In this case, the microcontroller is a 68HC11. **c**—In this parallel interface to the USBN9602, the microcontroller is an Intel 80C188EB. But, this example would be typical of any case where an 8-bit data bus is available.

So, a class-compliant USB product can entirely specify what it is and how it works in the onboard firmware. This makes the job of building, testing, and modifying a USB interface easier and more modular, and it brings it within the capabilities of most developers.

In the joystick, there are only three essential blocks—the ADC, USB interface, and microcontroller. The micro ties it all together, sampling the joystick at intervals and passing the data up through the USB interface (also managed by the micro).

The only new element is the USB interface. There are many varieties available: some are integrated with the microcontroller and some are separate components. These interfaces contain the state machines and buffers necessary to transmit and receive serial data on the USB. Conceptually, it's a smarter-than-average UART-like function.

National Semiconductor's USBN-9602 is one example of a USB interface. One side is attached to the USB cable or connector with a circuit like the one in Figure 4. (This figure and the ones following are not complete schematics; they merely highlight specific functions and interfaces.)

C3 and C4 bypass the USBN9602's internal voltage regulator (used by the internal USB transceiver). R1 is the required pullup that the device uses to signal its presence (and data rate) on the bus. The other components reduce EMI and transmission line effects to provide a cleaner signaling environment.

TYPICAL CONNECTIONS

The other side of the USBN9602 is the data path to the microcontroller. This data path is flexible and allows easy use with a variety of serial or parallel interfaces (there's even a DMA interface for high data rates).

Figure 5a shows a Microwire interface to a COP8 microcontroller, as well as the requisite dot clock oscillator circuit. Figure 5b shows an SPI interface to a 68HC11, and Figure 5c shows a parallel interface to an 80C188EB.

To make it even easier, several USB device manufacturers provide sample firmware source code. For the USBN-9602, National provides source code in C with compiler options for all of the microcontrollers mentioned here (and readily ported to others). This code is available on the web. Such firmware provides a ready-made solution to the some or all of the necessary device-side protocol layers.

If you want to build a mouse, keyboard, or other HID device, just modify the descriptor tables and a few top-level (function and class layer) firmware routines. Even if you're not building an HID device, the firmware layer that manages the USB interface device and responds to the standard requests provides a solid basis to start with.

PLAIN AND SIMPLE

USB simplifies the lives of developers and experimenters alike. It's possible for OSs like Windows 98 to provide most of the drivers you'll ever need

for USB devices via class drivers, which make USB easier to incorporate into products and embedded systems. ☐

Jim Lyle is a staff applications engineer at National Semiconductor where he has worked with flash memory, microcontrollers, and USB products. Jim has also worked as a development engineer and technical marketing engineer for Tandem Computers, Sun Microsystems, and Troubador Technologies. You may reach him at jim.lyle@nsc.com.

RESOURCES

USB information, www.usb.org
 HID device information, www.usb.org/developers/hidpage.htm and
www.microsoft.com/hwdev/hid
 USBN9602 firmware source code,
www.national.com/sw/USB

SOURCES

USBN9602, COP8
 National Semiconductor
 (408) 721-5000
 Fax: (408) 739-9803
www.national.com

68HC11
 Motorola
 (512) 895-2649
 Fax: (512) 895-1902
www.mcu.motsp.com

80C188EB
 Intel Corp.
 (602) 554-8080
 Fax: (602) 554-7436
www.intel.com

FROM THE BENCH

Jeff Bachiochi

Look Ma, No Hands The Qprox Touchless Sensor



Jeff explores the new charge-coupled

touch sensor—the QT110—from Quantum Research Group. As you'll see, the analog and digital techniques it uses improve the long-term reliability of capacitive sensors.



Franklin's kite had actually taken a direct hit from a bolt of lightning back in the 1750s, it's doubtful that he'd be on the \$100 bill today. There's some doubt that he ever really flew a kite during a thunderstorm, but investigators believe he did it to prove that storms were massively charged.

The kite's string successfully transferred a small portion of that charge from the storm to his Leyden jar. Because glass is a good insulator, the charge remained in the Leyden jar until it was bled off either intentionally or through leakage. The ability of this capacitor to transfer and hold a charge has made it an important element in electronics.

There are three factors that interplay when dealing with capacitance. A voltage potential (neces-

sary to supply current flow), capacitance (the ability to take on a charge), and time (how long the potential is applied).

We can define capacitance as the amount of charge current that a device takes on over time. This value is proportional to plate area, distance between plates, and the insulating material.

Naturally, some materials are better insulators than others. At the low end of the scale is air, with a dielectric constant of 1. One of the best materials is glass, which can have a dielectric constant >8 (e.g., the Leyden jar).

The larger the capacitance value, the slower the rate of charge. The larger the voltage potential, the faster the rate of charge. The reason that time is involved has to do with circuit resistance.

A perfect voltage source would charge a perfect capacitor instantly. Because of resistances in the voltage source, the circuit connections, and the capacitor itself, the charging current is limited. Thus, the more resistance the circuit has, the more time it will take to complete its charge.

A capacitor will charge to 63% of the voltage source in one time constant. This is equal to time (in seconds) equals resistance (in ohms) times capacitance (in farads). So you see, with zero resistance, time is also zero.

Accurately controlling both time and resistance enables the capacitance to be calculated by measuring the amount of charge the capacitor in question has taken on. Increasing the capacitance (putting some capacitance

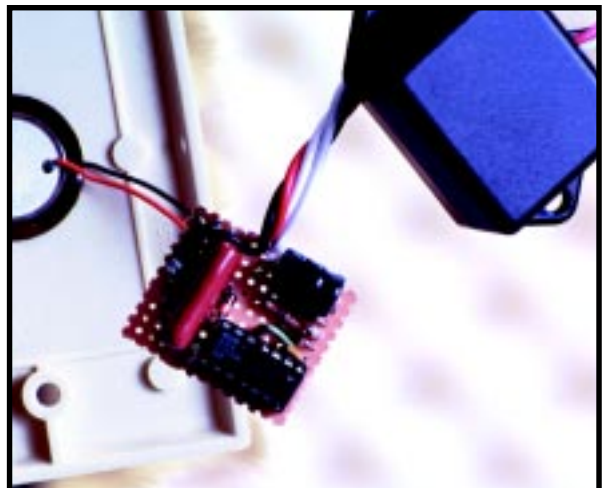


Photo 1—All the parts for Figure 1's circuit mount on less than 2 in.² of PCB. The triac is on the bottom where a heat sink is only needed for higher power devices. Notice the piezo device mounted to the blank switch plate.

in parallel) reduces the amount of charge (for the same potential and resistance).

If the added capacitance is stray capacitance (like from your body), you have just detected your presence in the vicinity of the circuit. Stray capacitance detection is the basis for capacitive sensors.

SAMPLE THIS

Quantum Research Group has released a self-contained charge-coupled touch sensor in an 8-pin package that will make all of your dreams come true. Unless, that is, you don't dream of capacitive sensors (although after reading this, you might start).

The Qprox QT110 uses some sophisticated yet simple digital and analog techniques to create a highly stable contactless sensor. A small external sampling capacitor and sensing electrode are all that's needed.

The charging is controlled by the QT110's internal timing circuitry. This circuitry closes a solid-state switch, which applies a measured charge to the sensing electrode.

When the switch is opened, the charge remains and a second switch enables the charge to be coupled to the external sampling capacitor. This process can take place a number of times, depending on the charging potential and the external capacitor values. Each time, the sampling capacitor's charge is integrated.

The internal 14-bit ADC can then read the accumulated charge on the sampling capacitor. The internal control circuitry discharges the sampling capacitor to ready it for the next cycle.

The ability to change the dynamics of the charging pulse by using multiple bursts improves sensor performance over those involving tuned-circuit, bridge, or RC-based capacitance sensor techniques. This system is highly tolerant of changes in the external sampling capacitor and the electrode (background) capacitance because the threshold level is dynamically compensated for over time.

The external capacitor should be around 0.02 μF with the sensing electrode looking like 20 pF. Because piezo "beepers" are essentially large flat

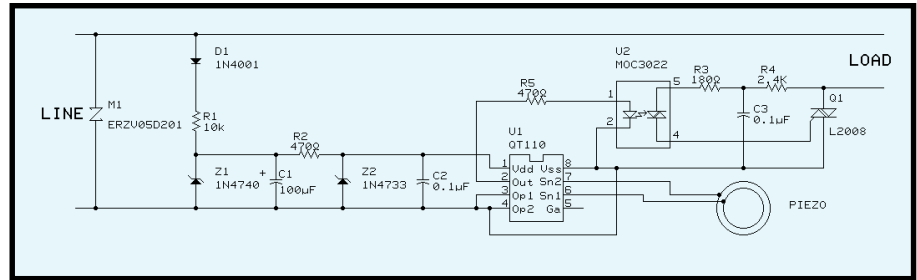


Figure 1—This schematic shows how I used a QT110 to control a triac replacing a wall switch. I noticed that, after a power failure, the switch as shown defaults to on. This can be remedied by using an inverting transistor between the QT110's output and the MOC3022 (because the QT110 can only source 1 mA—not enough for the MOC3022's LED).

ceramic capacitors, they can often be used as the external capacitor with the added benefit of audible feedback.

When the sensor is triggered, an internal H-bridge applies a 4-kHz burst across the two sensor inputs, driving the sounder for 75 ms. Note that the charging and discharging cycles also produce some audible clicking; however, it's generally rather low in energy.

STRAY CAPS

So, here's another question: how does our body look like stray capacitance to a circuit that might not even be grounded? The circuit may be capacitively coupled to earth ground through transformer windings or, if battery powered, through the stray capacitance of the PCB and earth.

A path from the circuit to ground is a must if the electrode is expected to measure the stray capacitance of our body to ground. Besides the hard-wired or transformer coupling approach, virtual capacitive grounding can be created by increasing the ground plane of the circuit with a conductor.

Our bodies have capacitance values in the high-picofarad range. Although the sensing electrode can be just about any size, it need only be as large as the area of touch. Surrounding the electrode with ground plane is one way to create a directional sensing field and prevent the electrode from being influenced by nonintrusive movement.

GET MY DRIFT

In the past, when a capacitive sensor was designed, it was based on a specific capacitor and the sensor was only as good as the stability of the circuit. The QT110 uses signal processing to alter the design dynamically to adjust for drift.

This drift may be based on the capacitor's changing value over time or temperature. But more importantly, it can compensate for changes in local stray capacitance.

Remember how the charging bursts can be controlled to create just the right charge build up on the sampling capacitor? The burst rate helps to compensate for total capacitance drift.

A reference-level charge is based on past measurements of the sampling capacitor. As the measurements rise and fall, the reference level tracks this to eliminate drift.

Actually, a rise in sampling measurements doesn't affect the reference level immediately. If it did, detection could not take place because the reference would rise with the detection signal.

To be detected, a measured charge must increase past an upper threshold level. This upper level is based on the reference level and follows its movements. Two methods of reducing false triggering are employed.

First, to eliminate spurious noise, the measured charge must remain above the upper threshold level for at least four consecutive measuring cycles. Second, the measured charge must drop below a lower threshold before it can again retrigger as a "hit."

When the circuit is first powered, a calibration routine provides a means to establish the reference level and then adjusts the charging bursts based on the sampling capacitor's measurements. The automatic drift compensation is disabled whenever the sensor recognizes a hit and is reenabled once the sampled measurement drops below the lower threshold.

The result of this action is if some stray capacitance is recognized as a hit

and never goes away (e.g., a pen being placed next to a sensor), the signal may never return below the lower threshold and subsequent hits may not be recognized.

However, the QT110 has a time-out feature that automatically performs a recalibration after 10 s (or alternatively, 60 s). The recalibration now sets a new reference level based on what it presently sees, eliminating (or compensating for) the added stray capacitance.

MODAL MANNERISMS

Once detection is recognized, there are choices you can make as to how this detection is presented. The output pin can sink 4 mA or source 1 mA. You can choose to have the output go low while the sensing electrode registers a hit or to generate a 75-ms low for each hit. An alternative toggle mode changes state on each detected hit.

These modes are selected via two option inputs. As an added benefit, the QT110 sets the output into tristate mode for 300 μ s each charging burst. This can be used as a heartbeat in

systems where a microprocessor is monitoring the QT110's output. The heartbeat indicates an active sensor.

Typical operating current for the QT110 is $\sim 20 \mu\text{A}$ (3 V). This voltage enables the QT110 to be powered from a micro's output bit. Because calibration is done automatically on power-up, the micro can be used to route separate sensing electrodes to the QT110's sensor input, presumably enabling it to monitor alternate electrodes.

TOUCHY VS. FEELY

When entering a dark room, my hands immediately search for the light switch. This feely motion not only searches for the switch's control lever but also for the position of the lever (up or down). Once I recognize the switch's position, a quick flick in the appropriate direction fills the room with incandescence. Let's use the QT110 as the basis of a touch-sensitive replacement to the utilitarian governor of darkness.

Figure 1 depicts a design that I used to power the sensor as well as control

the AC to the lighting fixture. The QT110 requires little current to operate, but you must have sufficient current left over to do the controlling.

I was tempted to use an off-the-shelf DC-controlled AC switch like an industry-standard OAC05. Better yet, I could use one of those opto-isolated black blocks with integral screw terminals.

However, I didn't want to wait for delivery of the parts from a supplier. I decided to forge ahead with stuff I had in my parts bins.

My first thought was to control the L2008 triac directly from the QT110. Its sensitive gate has only a small requirement for gate current to control the 8-A output potential. Unfortunately, the small current necessary is about 20 mA, a few times what the QT110 can supply.

The alternative was to use a Motorola MOC3022 opto-triac. Although the output is a triac itself, it was intended to be used as a switch for larger triacs, like that of a Darlington arrangement.

For the sensor electrode, I used a bare piezo element (about the size of a quarter) from Radio Shack. It can be easily glued onto the back of a blank switch plate and act as both the sensing electrode and the audio feedback device. However, seeing the lamp turn on or off is all the feedback you'll need—unless of course the bulb's life has ended, in which case you'll be able to answer the age-old question, "How many engineers does it take to change a light bulb?"

Requiring only a few milliamps to run the QT110 and control the optotriac driver gives us a new dilemma. Where does the power come from?

A few months ago, I spent an entire column talking about low current AC/DC converting. Some of the Harris parts I used are now obsolete.

But, wouldn't you know, these were the last unique parts available for making a simple small current step-down ADC. Now I'm looking for only a few milliamps, so let's try to steal some without having to expend too much heat.

My first thought was to use a 30-V zener on the half-wave rectified signal so it could be followed with a linear regulator (whose maximum input voltage is 35 V). I could get away with poor filtering as long as the resulting voltage was above the regulator's minimum input voltage of 7.5 V.

Quiescent current of a 7805 regulator (~8 mA) was more than I needed for the rest of the circuit, so that would be a waste. Instead, I used a 10-V zener as a preregulator to reduce the working voltage requirement of the filter capacitor and save size.

Then I followed this input section with a 5.1-V zener diode to leave the ripple behind. The supply ripple and noise requirement of the QT110 is 20 mVp-p (see Photo 1).

THE PRICE IS RIGHT

Parts for this project run about \$10, with the QT110 at \$3.50 in small quantities. Where can you have more fun for a 10-spot? If you want to play around with this device without having to lift a soldering iron (shame on

you!), you can get a small EV board for \$15 from Quantum Research Group.

There's a ton of proposed applications for this tough little sensor. You may have already seen a few of them in action without realizing it. For example, some restroom faucets are equipped with them. Just bring your hands in close proximity to the faucet and voilà, running water.

These sensors are even used in elevator buttons (no mechanical parts to wear out). I've touched the future and it's here. ☒

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.

SOURCE

QT110

Quantum Research Group Ltd.
(412) 391-7367
Fax: (412) 291-1015
www.qprox.com

Tom Cantrell

Working with Accelerometers



Welcome to the wonderful world of micro-

machines! In fact, after taking a close look at a new two-axis accelerometer, Tom thinks these devices may even dominate the next price/performance curve.



The march of silicon is truly a wondrous thing. In what other course of human endeavor do we routinely expect to get more for less? Especially in the embedded world, better and cheaper chips continue to drive more applications into more markets.

Many wonder if the pace of Moore's Law can be maintained, fearing there's a wall beyond which even the IC wizards won't be able to leap. It's rather unrealistic to imagine that the pace can be maintained indefinitely, and one can argue that the reality of the chip business dictates some fundamental limits as formidable as those posed by technology.

For instance, consider the recent wave of low-priced MCUs announced by the likes of Microchip, Motorola, and Zilog. Frankly, the fact that one might cost \$0.50 and another \$0.49 isn't compelling, either in terms of which is chosen or of enabling new business. What

you're likely to see tomorrow is a \$0.50 MCU that does a lot more than today's, rather than one that's a lot cheaper.

Unlike commodity ICs, one area where silicon is just hitting its stride is micromachining, which turns the IC manufacturing process into something along the lines of "Honey, I Shrunk the Machine Shop." Researchers have been able to fabricate all manner of microscopic gadgets such as motors, gears, and so on.

G-WHIZ

The ADXL202 two-axis $\pm 2-g$ accelerometer from Analog Devices is a good example of a micromachine that's making waves in the commercial market. More sensitive than earlier airbag designs, it's well suited for novel applications like two-axis tilt sensing and inertial navigation. For instance, Microsoft is using the '202 in their new Freestyle Pro game controller, which senses body motion.

The basic principle of micromachined accelerometers is simple enough. A tethered or "sprung" mass is forced into motion by an applied acceleration. The distance that the mass moves, and thus the acceleration, is determined by differential capacitance, as shown in Figure 1.

The principle may be simple, but the implementation is incredible, given the intricacy of crafting it in silicon. Consider that the smallest detectable capacitance change, 20 zF (yes, that's "z" as in 10^{-21} F), corresponds to a 2-pm deflection! But while it's capable of resolving mere mg's (thousandths of a g), the device can take a 500-1000-g hit and keep on ticking.

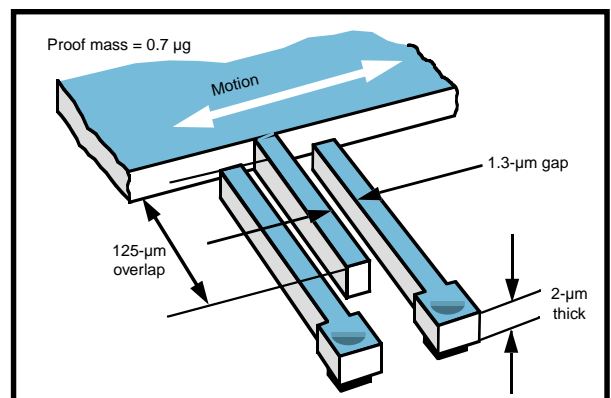


Figure 1—The ADXL202 works by translating movement of the sprung mass into differential capacitance.

The use of a standard IC process means the same die can integrate signal-conditioning and digitizing circuits, dispensing with the design hassles of dealing with low-level analog signals. That makes the ADXL202 real easy to use. Just add power (3–5.25 V, a mere 1 mA at that) and have at it with your favorite MCU or PLD.

As shown in Figure 2, the output of each axis is fed to a modulator (DCM) that translates the analog signal to a duty cycle on the digital x and y output pins. The acceleration is simply interpreted as 12.5% shift in duty cycle ($T1/T2$) per g —that is, full-scale output (-2 to $+2 g$) corresponds to a nominal 25–75% duty cycle (see Photo 1).



Photo 1—Exploiting micromachine technology, the Analog Devices ADXL202 makes precision acceleration measurement easy and affordable.

Digital output offers huge advantages, not only eliminating the need for an ADC but minimizing wiring concerns as well. That's because the digital signals are more immune to wiring-induced noise. The outputs drive nearly rail-to-rail (e.g., 0.2–4.8 V in a 5-V design), so it's nearly impossible for the logic at the receiving end to get confused about whether it's seeing a 1 or a 0.

Nevertheless, traditionalists are placated with analog x and y outputs. External filter capacitors allow adjusting the bandwidth/noise tradeoff.

One advantage of the analog outputs over the digital is that the former offer higher bandwidth (5 kHz vs. 500 Hz). But, analog output sensitivity isn't that great (312 mV/g; i.e., 1.248 Vp-p), likely calling for an op-amp. Fortunately, 500 Hz (i.e., 2-ms sample time) is plenty fast, which makes the digital

interface the proper call for most applications.

Besides the filtering capacitors (and recommended 0.1- μ F power-supply bypass cap), the only other external component is a resistor (RSET) that adjusts the period of $T2$ anywhere between 0.5 and 10 ms.

Generally, the proper approach is to choose the longest $T2$ period possible within constraints imposed by bandwidth requirements and timing logic. For instance, if you need to sample at 200 Hz, $T2$ must clearly be 5 ms or less. Similarly, if you're using an 8-bit counter clocked at 10 μ s, $T2$ should be less than 2.5 ms (i.e., $256 \times 10 \mu$ s) lest the counter overflow.

Though not targeting airbags, the ADXL202 does inherit a handy reliability feature from its high- g (and highly lawsuit prone) cousins. Setting the ST (self test) pin high forces electrostatic deflection of the micro-machined beam sufficient to generate an easily detectable 10% (i.e., almost 1- g equivalent) shift in the duty-cycle output.

LET THE ACCELERATION BE WITH YOU

The best way to understand the design issues and tradeoffs is hands-on experimentation. Fortunately, Analog Devices makes it easy with some low-cost evaluation units.

I got my ADXL202EB off the shelf from Crossbow Technology. Besides some other ADXL-based modules (I see two in the latest Jameco catalog), Crossbow is a source for more esoteric gadgets like six-degree-of-freedom fiber-optic gyros.

Although both Crossbow and Analog Devices offer smart modules including MCU, RS-232 interface, and driver software, the entry-level '202EB I used is little more than an ADXL202 and a few discretes on a tiny board.

The main virtue, besides the \$29 price, is that the board

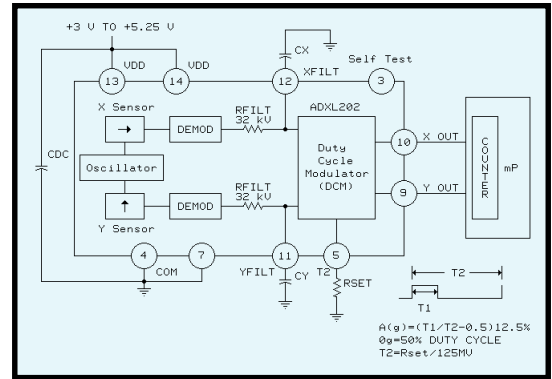


Figure 2—The ability to use a standard IC process enables monolithic integration of the micromachined sensor and all the signal conditioning electronics on a single die.

makes prototyping with the tiny 14-pin surface-mount package a lot easier. A standard 0.1" five-pin header connects power, ground, X, Y, and ST.

The board comes without the two filter caps and resistor installed, leaving the choice of bandwidth and $T2$ period to the user. I decided to go for 50 Hz and 7.4 ms, respectively. Actually, I decided to use the 0.1- μ F caps and 1-M Ω resistor I found in a drawer, and 50 Hz and 7.4 ms is what I ended up with.

This worked out well because I was using a small 8051-based BASIC SBC for the evaluation. It's no speed demon, so I didn't expect 50 Hz to be a speed limit. But, BASIC52 does have floating point, which eliminates concerns about dealing with fractional math.

The first step is some software to listen to the '202, so let's walk through Listing 1. The program sets aside some memory at the top of BASIC's RAM for an assembly-language routine that performs the raw timing.

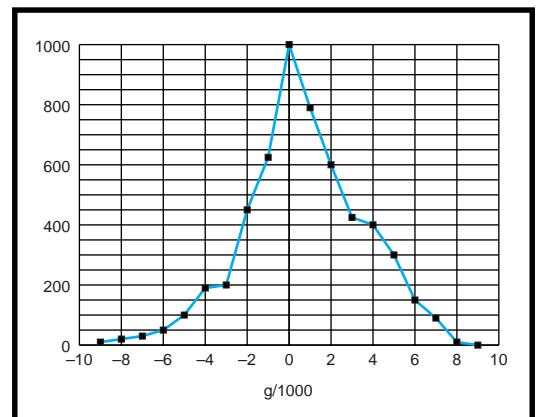


Figure 3—As demonstrated in actual operation, the noise is normally distributed and meets the datasheet specs.

The BASIC program calls the respective entry points to get a T1 or T2 reading. The routines use the 8051 branch-on-bit set/clear (JB and JNB) instructions and the built-in TIMER1 to measure elapsed time between edges. (Note: all my examples reference the *x* axis, but *y* works exactly the same.)

After setting up the ASM routine and specifying offset and scale calibration factors, the test program enters a loop sampling T1 and T2, computing the elapsed time (the 8051 TIMER1 runs at 1.085 μ s per count) and corresponding *g*'s, and then displaying the results as shown in Photo 2a.

Ah, but where did the calibration factors come from? Listing 2 is the calibration routine that determines the offset and scale factors (see Photo 2b) by taking $-1-g$ and $+1-g$ readings. Calibration is a must since absolute sensitivity and offset are rather loosely specified.

For instance, the nominal duty-cycle shift is 12.5%/g, but it can vary by 2.5%/g between units. And the 0-g offset, nominally 50% duty cycle, may be anywhere between 25% and 75%.

JUST SAY NO TO NOISE

Looking closer at my test program reveals a questionable practice—the T2 and T1 readings aren't taken at the same time. It's conceivable that an error might be introduced by the duty-cycle modulator beyond that inherent in the basic sensor.

DCM error could affect both T1 and T2 absolutely but not their ratio. Under 0-g acceleration (i.e., 50% duty cycle), one T1/T2 reading could be:

$$\frac{x}{2x}$$

and a second:

$$\frac{x+n}{2x+2n}$$

Both are 50%. But, a split T1 and T2 reading might result in:

$$\frac{x}{2x+2n}$$

which could be a problem if *n* is large.

I decided to poke around the issue a bit, starting with the raw noise spec of the sensor, which is proportional to the square root of the bandwidth. Ac-

ording to the datasheet, the peak-to-peak noise (95% probability) at 50 Hz is 17.2 mg.

Before trying to quantify any noise, better make sure you can even see it. Let's start with the 8051 timer resolution. Referring back to Photo 2a, you see the T2 period is about 6800 counts.

But, only 50% of the cycle is used (i.e., 12.5%/g). That turns into 850

counts per *g*—that is, almost 1-mg resolution, much better than the noise spec of 17.2 mg. But, the ASM routine has a few mg of jitter because of the two-clock latency of JB and JNB.

Well, I could study it to death, but it's more fun to try some stuff and see what happens. I modified the test program to take readings and keep track of the results. As Figure 3

Listing 1—This program includes a small ASM routine that measures the duration of T1 and T2 and uses the ratio, along with empirically determined offset and scale factor, to compute *g*'s.

```

PROGRAM T1T2
INTEGER i,j,t1,t2
REAL g,offset,fs

BEGIN
  MTOP = 3fdfh          /* leave 32 bytes for ASM */
  GOSUB init           /* copy ASM routine */
  offset = 0.154       /* offset compensation in g's */
  fs = 1.018           /* full scale compensation divider */
loop:
  TIMER1 = 0: CALL 3fe0h /* get T2 */
  t2 = TIMER1
  TIMER1 = 0: CALL 3fe8h /* get T1 */
  t1 = TIMER1
  g = ((T1/T2)-0.5)/.125 /* compute g's */
  g = g + offset
  g = g / fs
  ? USING(0),"T2 =",t2,
  ? USING(####.##),"[" ,t2 * 1.085,"uS ] ",
  ? USING(0)," T1 =",t1,
  ? USING(####.##),"[" ,t1 * 1.085,"uS ] ",
  ? USING(#####)," g =",g
GOTO loop
init:
  /* copy ASM routine to MTOP */
  FOR i=0 TO 21
    READ j
    XBY (3fe0h+i)=j
  NEXT i
RETURN

/* ; Time ADXL202 T1 & T2 using 8051 Timer1
00008B = TL1: equ 8bh ;Timer1 LSB
00008D = TH1: equ 8dh ;Timer1 MSB
00008E = TR1: equ 8eh ;Timer1 Control
000097 = AX: equ 97h ;Port 1.7
003FE0 ORG 3FE0H
; Enter at T2 to measure ADXL202 T2 (period, i.e., AX=low&high)
; Enter at T1 to measure ADXL201 T1 (high time, i.e., AX=high)
3FE0 3097FD T2: JNB AX,T2 ;Wait until AX high
3FE3 2097FD WAIT1: JB AX,WAIT1 ;Wait until AX low
3FE6 D28E SETB TR1 ;Start Timer1
3FE8 2097FD T1: JB AX,T1 ;Wait until AX low
3FEB 3097FD WAIT2: JNB AX,WAIT2 ;Wait until AX high
3FEE D28E SETB TR1 ;Start Timer1
3FF0 2097FD WAIT3: JB AX,WAIT3 ;Wait until AX low
3FF3 C28E EXIT: CLR TR1 ;Stop Timer1
3FF5 22 RET ;Return to BASIC */

DATA 030h,097h,0fdh,020h,097h,0fdh,0d2h,08eh
DATA 020h,097h,0fdh,030h,097h,0fdh,0d2h,08eh
DATA 020h,097h,0fdh,0c2h,08eh,022h
END

```

```

a)
T2 = 6834 7414.89 uS T1 = 4155 4508.17 uS g = 0.999
T2 = 6834 7414.89 uS T1 = 4153 4506.09 uS g = 0.997
T2 = 6836 7417.06 uS T1 = 4151 4503.83 uS g = 0.993
T2 = 6834 7414.89 uS T1 = 4153 4506.09 uS g = 0.997
T2 = 6834 7414.89 uS T1 = 4155 4508.17 uS g = 0.999
T2 = 6836 7417.06 uS T1 = 4153 4506.09 uS g = 0.995
T2 = 6834 7414.89 uS T1 = 4153 4506.09 uS g = 0.997
T2 = 6836 7417.06 uS T1 = 4159 4512.51 uS g = 1.003
T2 = 6834 7414.89 uS T1 = 4153 4506.09 uS g = 0.997
T2 = 6836 7417.06 uS T1 = 4153 4506.09 uS g = 0.994
T2 = 6836 7417.06 uS T1 = 4153 4506.09 uS g = 0.996
T2 = 6834 7414.89 uS T1 = 4155 4508.17 uS g = 0.999
T2 = 6832 7412.72 uS T1 = 4155 4508.17 uS g = 0.999
T2 = 6834 7414.89 uS T1 = 4153 4506.09 uS g = 0.997
T2 = 6834 7414.89 uS T1 = 4155 4508.17 uS g = 0.999
T2 = 6836 7417.06 uS T1 = 4155 4508.17 uS g = 0.998
T2 = 6834 7414.89 uS T1 = 4153 4508.17 uS g = 0.999
T2 = 6834 7414.89 uS T1 = 4153 4508.17 uS g = 0.999
T2 = 6836 7417.06 uS T1 = 4157 4510.34 uS g = 1.000
T2 = 6836 7417.06 uS T1 = 4157 4510.34 uS g = 1.000
T2 = 6834 7414.89 uS T1 = 4157 4510.34 uS g = 1.002
T2 = 6834 7414.89 uS T1 = 4153 4506.09 uS g = 0.997
T2 = 6832 7412.72 uS T1 = 4159 4512.51 uS g = 1.005

```

Photo 2a—Under static 1-g acceleration, the test program shows that the ADXL202 is quite accurate (i.e., within ± 0.01 g). **b**—Each ADXL202 must be initially calibrated with device specific offset and scale factor compensation. **c**—Presuming there's bandwidth and resolution to spare, simple low-pass filtering (i.e., averaging) can improve accuracy significantly.

```

b)
calibrate ADXL202
Orient AX up and hit any key
Orient AX down and hit any key
g = (g + offset) / fs
g = (g + 0.154) / 1.018

```

```

c)
avg 0.998 sample spread 0.008
avg 0.998 sample spread 0.008
avg 0.999 sample spread 0.009
avg 1.000 sample spread 0.011
avg 0.998 sample spread 0.008
avg 1.001 sample spread 0.009
avg 1.002 sample spread 0.006
avg 1.000 sample spread 0.006
avg 0.999 sample spread 0.007
avg 0.999 sample spread 0.012
avg 0.998 sample spread 0.010
avg 0.999 sample spread 0.006
avg 0.999 sample spread 0.009
avg 0.999 sample spread 0.011
avg 0.999 sample spread 0.008
avg 0.999 sample spread 0.008
avg 1.000 sample spread 0.009
avg 1.000 sample spread 0.009
avg 0.999 sample spread 0.012
avg 1.000 sample spread 0.013
avg 1.000 sample spread 0.012

```

shows, the results well matched the datasheet prediction (i.e., 95% < 17.2 mg).

With timing resolution better than the noise spec, there's a good opportunity to boost accuracy by averaging multiple readings. Of course, this cuts the bandwidth accordingly. Averaging four 50-Hz readings delivers 12.5-Hz bandwidth. Indeed, averaging is just the software equivalent of using bigger filter caps but a lot easier than getting out the solder sucker.

I modified the program to take 16 readings at a time and display the average and the spread. As you see from Photo 2c, the technique worked well. Although the spread in 16 samples routinely exceeded 10 mg, the average result only was off by 2–3 mg. The only downside of 16x oversampling was that BASIC could only manage to eke out about a 1-Hz sample rate.

Nevertheless, a few mg's is incredibly good (0.1% or so accuracy). It's accurate enough for tilt-

sensing applications that rely on trigonometry to convert a g reading to corresponding tilt (see Figure 4).

You can calculate the result directly or use a look-up table. But, do note the nonlinearity of the g -to-tilt function is such that the sensitivity required varies dramatically as the accelerometer rotates through 90° . At one extreme (0° ; i.e., accelerometer oriented parallel to horizon), the delta is 17.5 mg/degree.

But, at 90° (perpendicular to horizon), it's only a fraction of a mg per degree. One solution is to restrict the range to 0 – 45° per axis, exploiting the fact that both x and y axes are available to determine which octant you're in. Within the 0 – 45° range, a simple 16-mg/degree approximation is quite good.

Keep in mind the accelerometer also dutifully captures external accelerations. It can't tell the difference between acceleration due to tilt versus that created by movement of the module. A little motion is probably tolerable or at least filterable.

Don't forget to account for over-range readings (i.e., more than $+1 g$ or

Listing 2—This program computes the device-specific offset and scale factor by measuring the output with the accelerometer oriented up (i.e., $+1 g$) and then down ($-1 g$).

```
PROGRAM calibrate
INTEGER i,j,t1,t2
REAL g,min_g,max_g,fs,offset
BEGIN
  MTOP = 3FDFH
  GOSUB init
  ?"Calibrate ADXL202"
  ?"Orient AX up and hit any key"
  DO
    i = GET
    UNTIL i<>0
  GOSUB adx1
  max_g = g /* +1 g */
  ?"Orient AX down and hit any key"
  DO
    i = GET
    UNTIL i<>0
  GOSUB adx1
  min_g = g /* -1 g */
  fs = (ABS(max_g) + ABS(min_g))/2 /* scale factor */
  offset = (max_g + min_g)/2 /* offset */
  ? "g = (g + offset) / fs"
  ?USING(#.###),"g = (g +",-offset,
  ?USING(#.###),") /",fs
  STOP
adx1:
  g = 0
  FOR j=1 TO 64 /* average 64 readings */
    TIMER1=0: CALL 3fe0h
    t2 = TIMER1
    TIMER1=0: CALL 3fe8h
    t1 = TIMER1
    g = g + ((t1/t2)-0.5)/.125
  NEXT j
  g = g / (j-1)
RETURN
```

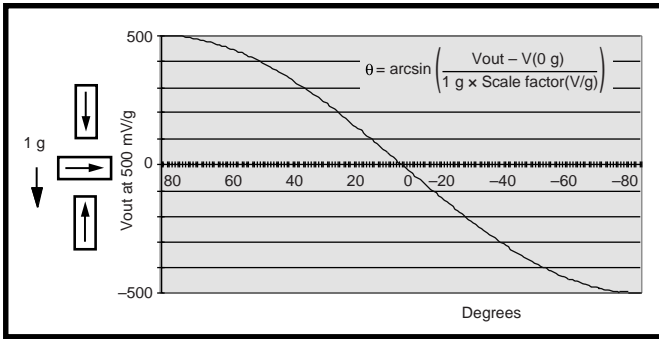


Figure 4—The ADXL202 meets the two main criteria for tilt applications: DC response and high sensitivity.

less than -1 g . If there's a lot of shaking going on, separating out the tilt component probably calls for adding a second (nontilting) accelerometer to act as a motion canceler.

PASS THE BATON

About four years ago, I put an accelerometer through its paces in "A Saab Story" (*Circuit Cellar* 57). Have we come a long way, baby?

That unit, from Silicon Microstructures (since acquired by Exar), had a lot in common with the ADXL202, including a micromachined sensor with good

Although the basic sensor element was micromachined, the IC process used wasn't suitable for integrating the conditioning electronics, making for a bigger, heavier hybrid multichip module.

The earlier single-axis unit retailed for over \$200 in singles with high volume quoted at around \$50. That's almost 10× the current ADXL202 price, not to mention getting the second axis as well. Analog Devices is projecting volume prices as low as \$1.50 per axis!

When it comes to regular chips, silicon may be starting to huff and puff a bit. But, it sure looks like micro-

sensitivity and accuracy across the same $\pm 2\text{-g}$ range.

But, its analog output not only required an ADC on the receiving end but caused me a bit of head-scratching over noise problems that turned out to be wiring related.

machines are picking up the pace and getting ready to blast through the next price/performance curve. 📡

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for more than ten years. You may reach him by e-mail at tom.cantrell@circuitcellar.com, by telephone at (510) 657-0264, or by fax at (510) 657-5441.

SOURCES

ADXL202

Analog Devices
(617) 329-4700
Fax: (617) 329-1241
www.analog.com/imems

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

Jameco
(800) 536-4316
(415) 592-8097
Fax: (415) 592-2503
www.jameco.com

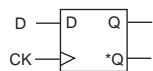
CIRCUIT CELLAR Test Your EQ

Problem 1—Show that $(a \oplus b) \oplus b = a$

Note: \oplus means exclusive OR, also called XOR, denoted by \wedge in the C language. $a \oplus b = a \cdot \bar{b} + \bar{a} \cdot b$

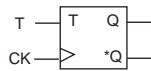
Problem 2—Show how to convert an edge-triggered D flip-flop into an edge-triggered toggle flip-flop. The tables show the intended operation of the D and toggle flip-flops. Assume all signals are active high.

Note: Use as many additional components as necessary.



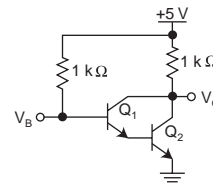
D	CK	Q_{t+1}	Operation
1	⌊	1	SET
0	⌊	0	CLEAR
∅	0	Q_t	Hold
∅	1	Q_t	Hold

∅ = Don't Care



T	CK	Q_{t+1}	Operation
1	⌊	$*Q_t$	Toggle
0	⌊	Q_t	Hold
∅	0	Q_t	Hold
∅	1	Q_t	Hold

Problem 3—Assume $V_{BE} \cong 0.7$ and $V_{CE(SAT)} \cong 0.3$. What is the quiescent state of this circuit? What are the voltages at V_B and V_o ? Why isn't V_o closer to ground?



Problem 4—You have a processor with only two free registers and no RAM (perhaps similar to the smallest Atmel AVR). The processor can perform all typical logic and arithmetic instructions. The registers are general purpose, and any register may operate as either a source or destination (or both) for ALU operations. Write a program to swap the contents of the two registers.

PRIORITY INTERRUPT

Servings Per Issue



The Internet is hard to ignore these days. Three years ago, businesses could say they weren't ready to have a web site. Today it's an absolute must. The reasons aren't all hype. Think about how you search for information today and how you did it a few years ago. There was a time when you were happy to fill out a magazine bingo card and wait 4–6 weeks for datasheets to arrive by regular mail. If you were more impatient than that, you'd call the company immediately and the datasheet would arrive within a week or two.

Although I'm speaking solely for myself, I suspect that many of you do the same as I do these days. When I see an ad for a product or company, I go directly to their web site. The reasons are twofold. First, the likelihood is that there will be some substantive information that immediately satisfies my curiosity. The second (and more important) reason is anonymity. Sometimes I just want to check out a product with no specific intent to buy it. I don't want manufacturer's reps calling the next day. I don't want my name on their mailing list, and I don't want to leave a name and number on a voicemail machine. I just want to rummage a little in their technical drawer.

So, what am I driving at? Well, it seems that we've changed the way we seek information in response to advertisements. Have we also changed the way we seek solutions and guidance from the articles in technical magazines? How much has this rabid desire for immediate gratification affected how you read or appreciate technical magazines today? Should they try to be a full-service information source or should they be an information directory instead?

When I see this metamorphosis in other magazines, it causes me to ask if I'm being too conservative. One of my favorite magazines through the years has been *Popular Science*. They've always been a wealth of technical information and the model for many of the in-depth presentations you've come to expect from *Circuit Cellar*. Certainly, neither magazine expects to take the place of fundamental learning sources, but we both strive to have our detailed technical presentations serve as a catalyst for the real engineering experience.

In the past couple years, I've noticed a change in the focus of *Popular Science*. For as long as I can remember, *Popular Science* has been a combination of technology snippets and a bunch of general feature articles. The snippets always seemed to be secondary to the features. The evolution has been gradual and maybe it's just my perception of things, but when I look at an issue of *Popular Science* now, I get a distinct feeling of being bombarded with a magazine full of one-paragraph technical quickies.

Certainly the continued growth and change of technology has prompted a wide variety of new topics that the editors feel they must cover. Unfortunately, the reality of print-magazine economics dictates that something has to give if they expand coverage of new topics in the same size magazine. In light of the fact that I'm presented with the same technology-coverage dilemma, I find that weighing the needs of the readers with the realities of being a publisher leaves me with the same questions and no answers. It's not that I'm going kicking and screaming into the next millenium. I just need to feel that any editorial reorientation occurs for reasons other than being trendy. The alternative is quite disconcerting.

You get a hint of what I mean when you deal with anyone who has grown up glued to a TV screen. Do we find more short features and highly graphic materials in contemporary consumer magazines these days because they perceive that we all have the attention span of a grape? Are technical magazines like *Circuit Cellar* immune?

Circuit Cellar's editorial direction has always been a matter of gut feeling. It has never been about any commercial game plan. My attention span, albeit attached to an increasingly aged personality, is the same as it always has been. I like to think that our editorial content is cutting edge, but that doesn't mean that it has to be knee-jerk reaction to changing technology either. At the same time, I don't want to think that my conservative nature and general resistance to change is keeping me from recognizing that some topics should be short and glitzy. I know from experience that *Circuit Cellar* readers have opinions, so let me know what you think.



steve.ciarcia@circuitcellar.com