CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#106 MAY 1999

MEASUREMENT AND SENSORS

Accurate Measurement in Harsh Environments

Graphing Weather Monitor

USB Essentials

Embedded Internet: What's Available?

TASK MANAGER

Sensing the Obvious

Editing is a hazardous profession. With this magazine, fortunately, we don't encounter life-critical situations (or at least that's what the disclaimer is for). But, my job has its perils nevertheless.

Over a year ago, Janice Hughes wrote in a Guest Task Manager that editing is a profession that encourages perfectionism and obsessive-compulsive behavior. To do it well, you have to care a *lot* about a *lot* of little things. Too true. We do care, and we try our hardest to make every bit of editorial true and accurate.

Unfortunately, we don't succeed 100% of the time. For example, last month, we added Test Your EQ, a new section that invites you to quiz yourself on how well you know engineering basics. As I write this, that issue is just hitting the newsstands and already my e-mailbox is full of messages from readers pointing out that there were—gasp!—errors in Problems 3 and 4.

Indeed, the schematic in Problem 3 was incorrectly drawn, and Problem 4 had some capitalization errors. Those are the kinds of mistakes that threaten credibility, and to an editor, that's the worst. Although I despise them—to the extent that I make a career out of purging their existence—I can live with the occasional (preferably, extremely rare) typo. But in my opinion, drawing a diode backwards in a quiz like Test Your EQ is a whopper of an error for an established engineering publication like *Circuit Cellar*.

Fortunately, along with the correction, many of the readers who wrote in also included compliments and said how much they like the magazine. One person wrote that the errors "couldn't just be a coincidence. After all, this is the April issue, and it's been so long since we've had even the slightest hint of tomfoolery, any errors *must* be glaringly wrong on purpose. And you thought you really had me, or vice versa!"

Well, we weren't pulling tricks on you, but here's something else to consider. If you're willing to move to Connecticut and you have a burning desire to join the nit-picking editors of *Circuit Cellar*, send us your resume. We are expanding our editorial staff and are looking for another technical editor.

Looking for an editor among our readers may seem a bit odd to you, but it makes perfect sense to me. Indeed, it came as no surprise that our readers picked up on the mistakes immediately. In fact, they generated more letters to the editor than I've seen in years.

Fortunately, I can see the positive sides to all this: One, we don't usually flub up in such an obvious manner. And two, you all really know your basics, or as Jeff said, "Well, that'll test 'em!"

But just in case you didn't notice that backwards diode in Problem 3, I've got a WeatherROCK I'd like to sell you.

Eli

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR® INK

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

MANAGING EDITOR
Elizabeth Laurençot

TECHNICAL EDITORS
Michael Palumbo
Rob Walker

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Ingo Cyliax
Ken Davidson
Fred Eady

NEW PRODUCTS EDITOR
Harv Weiner

ASSOCIATE PUBLISHER
Sue Skolnick

CIRCULATION MANAGER
Rose Mansella

CHIEF FINANCIAL OFFICER
Jeannette Ciarcia

ART DIRECTOR
KC Zienka

ENGINEERING STAFF
Jeff Bachiochi

PRODUCTION STAFF
Phil Champagne
John Gorsky
James Soussounis

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

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 responsiblity 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.

INSIDE ISSUE 106

READER I/O

TIMING IS EVERYTHING

I read "A Minimalist Multitasking Executive" (*Circuit Cellar* 101) with some skepticism as to the benefit of multitasking in embedded systems. Now, I'm even more skeptical and concerned about the errors in the article.

The authors "chose a value of 2000 cycles per tick, or about 25 µs for a 'HC11 running with an 8-MHz clock" (p. 21). An 'HC11 with an 8-MHz clock has a 2-MHz internal clock, so 2000 cycles equals 1 ms, not 25 µs.

No 'HC11 with an 8-MHz internal clock capability is listed on any Motorola short form. Even if there was, the tick would be 125 ns × 2000, or 250 µs, which makes such an OS impractical for most embedded systems (usually heavily interrupt dependant).

The 'HC11 has a possible interrupt latency of 40 (IDIV/FDIV) + 27 + 12 cycles for the RTI, for a total of 79 cycles or 39.5 µs, plus any time required to process the interrupt (e.g., turning off the interrupt flag). These tasks consume ~13 more cycles, so even the simple timer ISR suggested requires at least 50 µs to process—unless the programmer couldn't use the divide instructions, which reduces it to 35 µs but makes the 25-µs tick impractical.

Other than the timer interrupt handling task scheduling, no interrupts can be processed as ISRs. They have to be polled. This fact precludes this OS from almost all embedded designs, even some of the simplest. If the authors wrote the C compiler, they should have had a better knowledge of the 'HC11's timing.

Alan Cook
South Australia

Our purpose wasn't to extol the virtues of a multitasking executive, but to demonstrate two of its fundamental concepts—timer interrupt preemption and the state-saving mechanism. Any preemptive executive has to deal with these. The article shows that to get a minimal system going, you doesn't need a lot of code, even in C.

You are correct regarding the timing error. The important parameter isn't so much the amount of time, but the numbers of cycles per tick.

This system is meant to be reasonably portable. Does 2000 cycles give a reasonable amount of tradeoff between interrupt and scheduling overhead and system responsiveness? If your system needs to schedule more often, change the tick cycle time and the default number of ticks. Decreasing the tick cycle time increases scheduling activity and introduces more overhead.

This system doesn't preclude you from using other interrupts. It lets you hook to the timer interrupt so if you want to poll at some resources, you can do so at the timer interrupt rate. It masks interrupts off during the periods that it accesses the global data structures, which may not be acceptable in some cases.

A system using nothing but interrupt handlers and an idle main loop is a multitasking system. In this scenario, depending on the hardware, scheduling is done by the interrupt priorities (preemptive) or interrupt handlers giving up control (cooperative). But, interrupt handlers are not the right place to perform lots of calculations.

Richard Man

Circuit Cellar ONLINE

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 May Design Forum password is:
Forecast

New!

- Now there's an easier way to search past articles for certain topics. The **searchable CD-ROMs** of the *Circuit Cellar* back issues are ready to be shipped, so stop by our homepage and find out how to get yours!
- Don't forget that your Design99 entries are due June 1, 1999, so check the **Design99 Rules Update** section for the latest updates on contest guidelines.

Design Forum

Silicon Update Online: Flexible Flash '51—Tom Cantrell
Lessons from the Trenches: Logging in the '90s—George Martin
Design Hint: Working with USB: Checklist, Glossary, and USB Sources—Mike Zerkus, John Lusher, and Jonathan Ward

NEW PRODUCT NEWS

Edited by Harv Weiner

HUMIDITY/TEMPERATURE TO PC CONVERTER

The **RH-02** is a compact PC-based instrument that accurately measures both temperature and humidity. It plugs into the serial port of a PC and uses the supplied PicoLog software (DOS and Windows 3.1/95/98/NT) to take measurements.

The software provides simultaneous views of temperature and humidity graphs, exports data to a spreadsheet, prints and saves data, and more. Alarm limits can be set to sound if the temperature or humidity go out of a specified range. The unit is also supplied with software drivers so users can write their own application software. Examples for LabVIEW, Excel, Visual Basic, Delphi, and C are included.

Applications include monitoring cleanrooms, laboratories, and computer facilities; checking heating and ventilation installations; and monitoring artifacts and documents in libraries and museums. The unit requires no power supply, so it's ideal for use in the field with a laptop computer.

RH-02 measures temperatures from –40 to +70°C with an accuracy of ±0.2°C (0–70°C). The device measures humidity from 0 to 100% with an accuracy of ±2% (5–95%).

RH-02 sells for **$245** and comes complete with manual and PicoLog software.

Saelig Co.
(716) 425-3753
Fax: (716) 425-3835
www.saelig.com

PROGRAMMABLE CONTROLLER

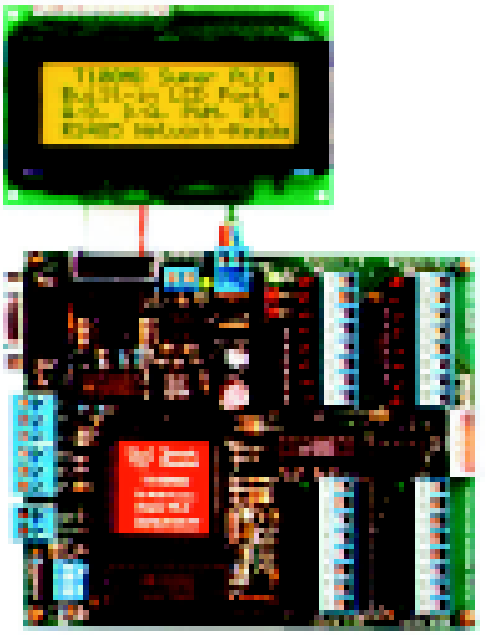
Triangle Research International has introduced a programmable logic controller that features a revolutionary integrated ladder logic + BASIC programming language. The **T100MD-1616** is easy to program using TRiLOGI V.4 software, which includes a programming editor, compiler, and simulator.

The software is enhanced by custom-programmable functions, which the user can create using a suite of BASIC commands. These commands, based on the popular BASIC language, are much more flexible and powerful for handling computationally intensive tasks such as data processing, string handling, and data communications. A free evaluation copy of the software can be downloaded from the company's web site.

The T100MD-1616 includes a 14-pin LCD module port, four analog inputs (0–1-V and 0–5-V ranges), one analog output (0–20-mA current loop), two PWM outputs, two stepper-motor controllers, four interrupts, and two high-speed quadrature-encoder inputs.

Also included are a real-time clock, PID computation function, one RS-232C and one RS-485 port (networkable: master-slave or peer-to-peer), 16 digital inputs (24 VDC, NPN type), and 16 solid-state outputs (24 VDC at 1 A per output).

Triangle Research Intl.
(877) 689-3245
Fax: (877) 689-3245
www.tri-plc.com



NEW PRODUCT NEWS

C-PROGRAMMABLE CONTROLLER

The **PK2500** is a small, rugged C-programmable controller that provides a low-cost high-performance alternative to micro and nano programmable logic controllers (PLCs). The PK2500 features quick-release screw terminals and easy field maintenance. It mounts on a DIN rail for fast field installation. Applications include environmental monitoring, process and batch control, temperature, water and waste-water control, and general machine control.

Its 22 I/O lines support up to 16 protected digital inputs or up to 12 high-current outputs. Eight of the lines are configurable. Two inputs can be alternately configured as an RS-485 port, and six of the output lines can be configured as inputs. Also included are four analog inputs with 12-bit resolution, two SPST relay outputs, and two serial ports that allow RS-232 or RS-485 communications. Each analog channel has a conditioning op-amp and changeable bias and gain resistors to match the desired analog input range.

The Dynamic C software (complete with integrated editor, compiler, and debugger) offers third-generation language support and execution speed that is unmatched by ladder-logic–based PLCs. The software libraries make this PLC alternative designer-friendly, minimizing project development time.

The PK2500 starts at **$245** per unit in 100-piece quantities.

Z-World, Inc.
(530) 757-3737 • Fax: (530) 753-5141
www.zworld.com

DIGITAL THERMOMETER/THERMOSTAT

The **DS1721** digital thermometer/thermostat provides 12-bit temperature readings that indicate the temperature of the device. Thermostat settings and temperature readings are all communicated to or from the DS1721 over a simple two-wire serial interface.

The device is truly a temperature-to-digital converter, as no additional components are required. Applications for the DS1721 include personal computers, office equipment, and any microprocessor-based thermally sensitive system.

The DS1721 reports the temperature of a device or environment with an accuracy of ±1°C over a temperature range of –10°C to +85°C. The device can also be used as a thermostat by setting the high and low temperature limits. When these limits are reached, the device generates a response.

The temperature is reported in a 9- to 12-bit word, with increments as small as 0.0625°C. Three address bits enable a user to multidrop up to eight sensors along the two-wire bus, which simplifies the bussing of distributed temperature sensing networks.

The DS1721 sells for **$0.99** in quantity.

Dallas Semiconductor
(972) 371-4448
Fax: (972) 371-3715
www.dalsemi.com

NEW PRODUCT NEWS

PIC DEVELOPMENT MODULE

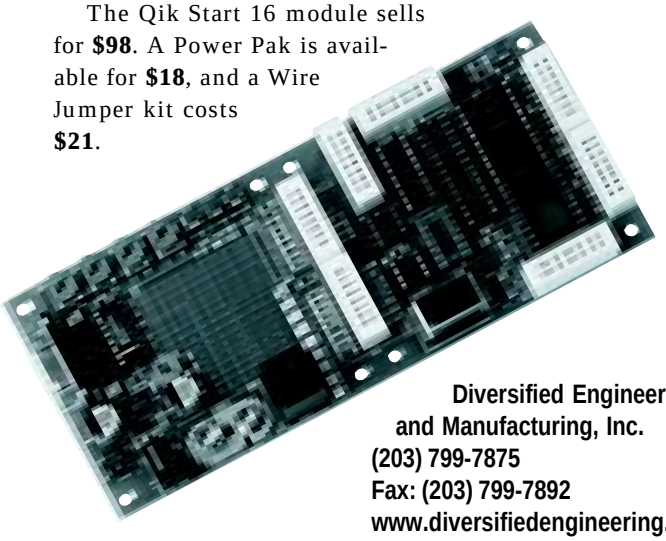
The **Qik Start 16** development module is an integrated stand-alone development and training tool to support students and engineers involved in the design of Microchip PIC microcontroller-based electronic circuits. It enables easy prototyping and testing of software and hardware components in designs using embedded controllers.

A major feature of the module is the background debugging module connector that interfaces to a PC and supports in-circuit emulation of the new Microchip 16F8*xx* series of microcontroller products. The Qik Start 16 module enables any 8-, 12-, 14-, 28-, or 40-pin 12C*xx* or 16C*xx* PIC series microcontroller to be used.

All V_{DD}, V_{CC}, OSC and MCLR lines are prewired. The OSC signal is generated from an onboard plug-in clock oscillator or crystal. The 2–5-V adjustable regulated power supply can be sourced from a 9-V battery or AC power pack.

Terminals are also available for an external lab supply or the company's Design Center. The MCLR can be initiated from the onboard momentary reset button or via an external reset source. There is a dedicated socket for EEPROM (I^2C or SPI). USART-based microcontrollers have Tx and Rx lines tied to a MAX232 interface chip, which connects to a DE-9. All port pins are brought out to connecting terminal blocks for easy access.

The Qik Start 16 module sells for **$98**. A Power Pak is available for **$18**, and a Wire Jumper kit costs **$21**.



Diversified Engineering and Manufacturing, Inc.
(203) 799-7875
Fax: (203) 799-7892
www.diversifiedengineering.net

FEATURES

Graphing Weather Monitor

FEATURE ARTICLE

Mark Bauman

Watch out, weather forecasters! Mark is checking up on you, and now all of his relatives are, too. For Christmas, they received PIC-based graphing weather monitors that analyze current data for accurate local weather prediction.

"It's snowing. We'd better get going!" I awoke to my wife's announcement and ran to the window. Indeed it was snowing and the temperature was 34°.

Wondering what the weather guessers were saying, I turned on the radio as the forecaster proclaimed that the snow was expected to change to rain within the hour. I checked the Internet to review the forecaster's discussion, written moments earlier. It confidently read "Sorry kids—no white Christmas."

Although I love snow (and we don't get it often in the Banana Belt), the forecast was fine with me because we had a couple hours to drive to reach my parents' house for Christmas Eve celebrations. But as an amateur weather watcher, I've learned to second-guess the weather service and to verify their predictions (although I must admit, they are getting better).

My heat unit calculator (a temperature monitoring and storage device that my wife and I designed several years ago) indicated the temperature had dropped 3° in the last hour, even though it was well past daybreak. This clearly didn't match the forecast. I looked at my old-fashioned dial barometer—the pressure had fallen (unfortunately, I couldn't remember when the pointer was last adjusted).

We might be in for an unexpected snowstorm, so we frantically piled the gifts in the car and headed to Grandma's house hoping to get a start before the roads got too bad. Sure enough, it snowed 8″ that day.

The week before Christmas had been a frenzy of activity. Shopping is my least favorite Christmas activity and in a moment of frustration I vowed not to buy gifts next year—I'd build them.

Now, driving down the snow-slicked highway, I got the idea to build an instrument that would show not just what the weather is doing but also how it got there, so I could understand where it was going.

Such an instrument would have clearly shown the contradiction between the weather predicted for that morning and what actually happened (and it would be a great gift for several difficult-to-shop-for relatives). The graphing weather monitor was born.

This project combined my interest in weather, embedded systems, packaging, analog design, and avoiding Christmas shopping. After all, if everything worked out I'd never have to shop again. I could just provide free firmware upgrades!

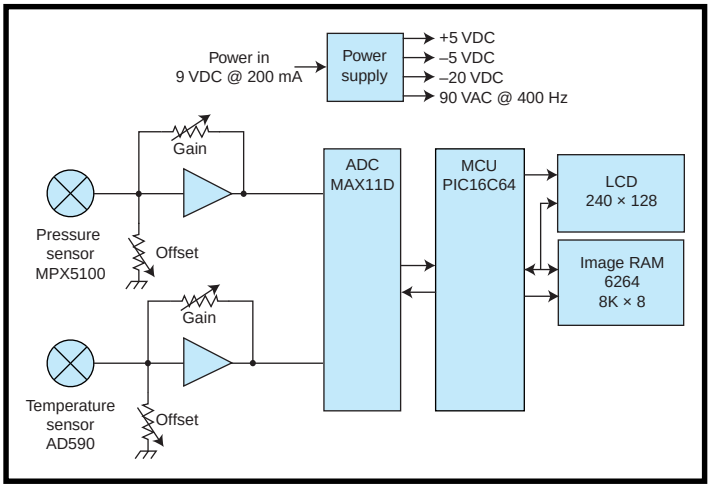


Figure 1—*This diagram of the graphing weather monitor illustrates the circuit architecture and shows the novel PIC-LCD interface and the more conventional analog front end.*

As you may have guessed, one year later I was feverishly fixing that last line of code (which is still more fun than shopping). Fortunately, everything came together and I delivered the units on schedule and within budget.

I've often thought that a weather instrument that clearly depicted trends in the temperature and barometric pressure would be valuable for anticipating the weather during the next several hours. I could track the steep pressure gradient that often precedes high winds or follow the passage of a front. I could watch the life cycle of a local thunderstorm as it builds and decays or even anticipate the change from rain to snow that accompanies an unexpected winter storm.

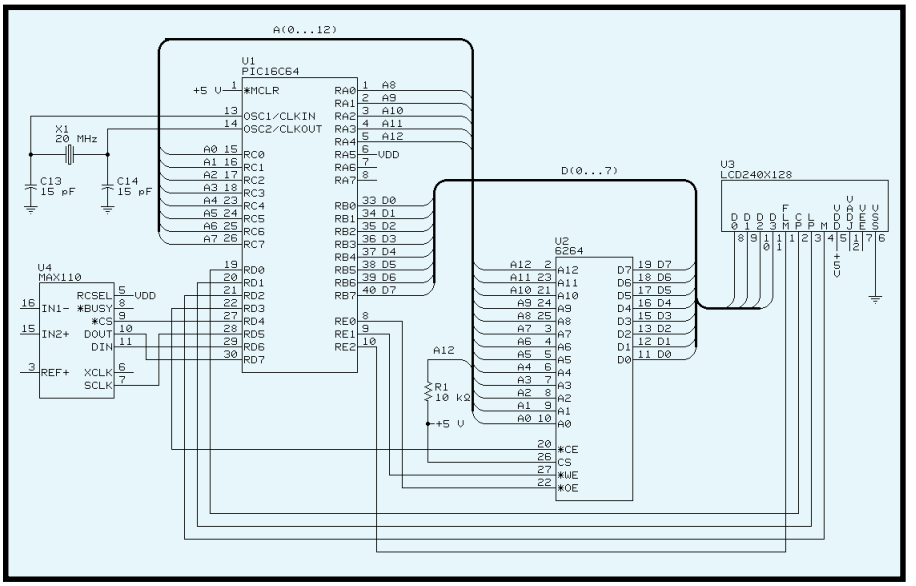


Figure 2—*This schematic shows the LCD control circuit. Note how the data bus is shared between the PIC, RAM, and LCD.*

DESCRIPTION

Packaged in a 2″-deep picture frame (4″ × 6″) with a laminated overlay, the graphing weather monitor fits neatly on a desk or shelf (see Photo 1). The graphic is formed on a 128 × 240-pixel LCD with EL backlighting, and the monitor is powered by a 12-V wall transformer (200 mA).

The barometric pressure and temperature are displayed simultaneously along with their trend lines for the last 48 h. There are also short-term trend lines showing how things have changed during the last 10 min.

The center horizontal line indicates the point where the pressure or temperature was equal to the current reading. Points above this line indicate that the pressure or temperature was higher than the current reading (by the amount indicated on the left and right axis). Points below this line indicate that the pressure or temperature was lower than the current reading. The pressure line is represented using a thick line, and the temperature line is shown as a thin line.

Each point on the long-term trend is updated every 12 min., which means that there are 5 pixels/h. The short-term trend is updated approximately every 20 s and is more sensitive than the long-term trend with 30 pixels in the graph (giving 10 min. of history). This arrangement is useful for tracking events such as the passage of a front.

DESIGN CONSIDERATIONS

As a full-time engineering manager, I'm familiar with design constraints and tradeoffs. I've learned the importance of listening to the voice of the customer as part of the initial design process and developing a dialogue during the design phases. I've also learned the importance of providing proven tools to increase staff efficiency and still stay within budget and schedule constraints.

For this project, I placed some severe constraints on the engineering staff

(me). I was working off a tight schedule and budget. Finding the voice of the customer was tricky because I wanted the project to be a (hopefully pleasant) surprise.

I had a general concept of what the graphing weather monitor would be, but it's always useful to differentiate between design musts and "it would be nice if...." The design *must* be:

- a stand-alone unit because not all recipients use computers
- simple to operate, with minimal setup
- reliable and stable
- able to illustrate trends. Graphs are powerful (a picture is worth a thousand words).
- ready for delivery (four units) on Christmas Eve, 1997

The design would be *nice* if:

- components cost less than $120 in single quantities
- new tools and setup (including PCB tooling) cost less than $200
- total design and build time was limited to 10 h per month (120 h total)
- through-hole design could be used rather than SMT because the unit would be built at home

CIRCUIT DESIGN

Now the fun begins. The graphing weather monitor circuit consists of a power supply block, sensor signal conditioning, ADC, MCU, external RAM, and an LCD, as shown in Figure 1.

The first design challenge came early. How should I display the information? Because I needed a stand-alone unit, I ruled out a PC interface. Another possibility was to provide video modulation for direct connect to a television. I chose a more conventional method—an LCD module.

Like you, I receive catalogs from surplus houses touting cheap matrix LCDs. Rummaging through back catalogs, I located some great displays for under $30. I used AutoCad to sketch the display matrix and determined the minimum resolution required. I was pleasantly surprised to find a 240 × 128 display that met my needs for only $20.

When you spend $20 for a 240 × 128 display, you should expect to have to provide a controller for the display. I've used standard LCD character modules with built-in controllers for a number of projects and found them easy to use, but now I was in uncharted territory.

I've learned the hard way that if something is available off-the-shelf, it's a good idea to use or modify it. So, I researched the available controllers for matrix LCDs (e.g., Epson's SED-1335). These controllers offered great promise but had major disadvantages in my application.

For example, most of them require external video RAM, which is not directly accessible by the MCU (without additional circuitry). I needed a large area of external data RAM, so an extra RAM chip was required.

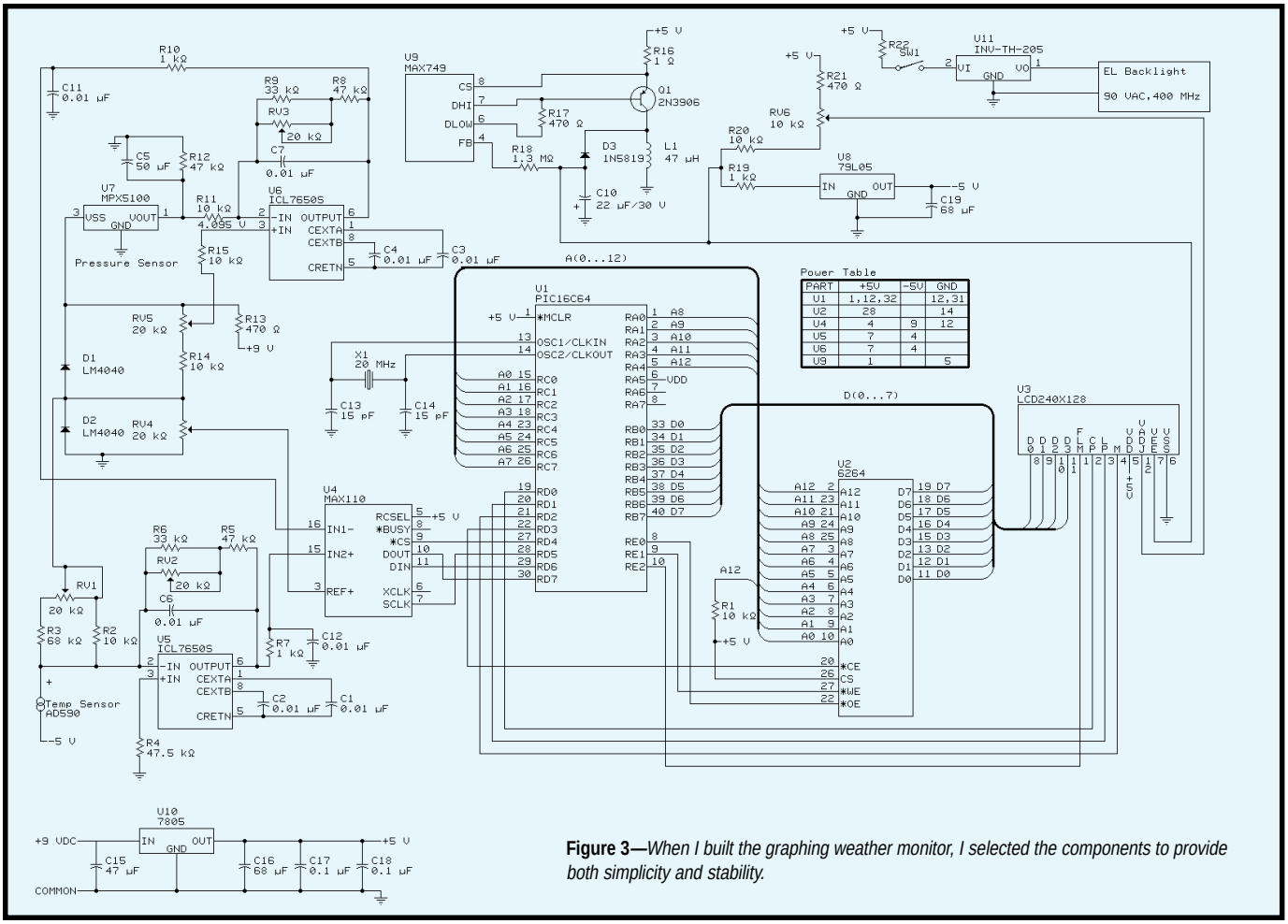


Figure 3—*When I built the graphing weather monitor, I selected the components to provide both simplicity and stability.*

Also, these controllers are only available in SMT packages (which didn't meet one of my design criteria) and they cost about $10 in single-piece quantities. The controllers are capable and offer many windowing features. They simplify character generation and display multiplexing but offer little help in graphing and line algorithms. I decided to design my own LCD controller—in software.

Pressure	Range: 27–32″ of Mercury (Hg)
Accuracy:	±0.05″ of Mercury (Hg)
Resolution:	0.01″ of Mercury (Hg)
Temperature	Range: –50.0–140°F
Accuracy:	±1°F
Resolution:	0.1°F
Power input	120 VAC, 50/60 Hz converted to 9 VDC, 240 mA
Operating temperature	50–90°F
Display size	240 × 128 pixels
Microprocessor	PIC16C64A

Table 1—*As you can see from these graphing weather monitor specifications, the unit is more than capable for casual weather observation.*

Of course, this forced some memory, I/O, and speed requirements on the microcontroller. The memory requirements for the display are 240 × 128 (or 3840 × 8 of RAM). Besides that, at least 512 × 8 was needed for the history data and an additional ~120 × 8 was needed for scratchpad memory, which gave me a total of over 4 KB of RAM. No small microcontrollers (that I had tools for) had this amount of onboard memory, so external RAM was required.

Using external RAM dramatically increases the I/O requirements for the microcontroller to a minimum of 29 points (unless the address/data are multiplexed lines, which would reduce to 22 points but require an external octal latch chip).

Speed was a major consideration. The LCD must be scanned nearly 100 times per second to prevent flicker. To accomplish this, the MCU creates an image stored in external RAM and then it "replays" this image sequentially by reading the external RAM into the onboard display drivers.

Because the MCU shares its data lines with both the external RAM and the display, the data read out of the external RAM also appears at the input to the display. While this occurs, the MCU frames the data by toggling the LCD control lines. As you may expect, the MCU spends the majority of time in this refresh loop. The required bandwidth for this link is ~800 kbps.

The Microchip PIC series (utilizing a Harvard architecture) provided a controller that met the memory, I/O, and speed requirements. A PIC16C64 with a 20-MHz crystal proved worthy in the circuit shown in Figure 2. Lower crystal frequencies caused some display flicker, so I stayed with the 20-MHz crystal in the final design.

I knew I'd need some sort of ADC. I was hoping to use the newly introduced (at that time) PIC14000, which has an onboard high-resolution ADC. Unfortunately, the I/O count wasn't sufficient (even using an eight-bit latch).

I selected the Maxim MAX110 12-bit ADC because of its resolution and simple interface to the MCU. As a side note, the PIC16C774 is now available with sufficient I/O and onboard ADC to do the job.

The display (U3) is an Optrex DMF660N-EW LCD with 240 (wide) × 128 (high) pixels. The onboard drivers are Hitachi HD-61105 and HD61104. The LCD uses a four-bit parallel data transfer. These data lines are shared between the PIC, RAM, and LCD.

There are four LCD control signals—first line marker (FLM), data latch signal (CP), clock signal for shifting data (LP), and alternate signal for LCD drive (M). These signals are provided by the MCU and used for data framing.

The LCD requires a negative contrast voltage of ~20 V for proper operation. There are many chips available for this task, but I chose Maxim's MAX749.

The MAX749 (U9) is connected in a flyback configuration to generate the contrast voltage. Its high efficiency (80%) and high switching frequency (500 kHz) allow the use of small components. The output is digitally adjustable, but I didn't use that feature in this design. The MAX749 also provides the voltage to drive the 79L05 regulator (U8) to power the –5-VDC analog requirements.

The display is illuminated using an EL backlight powered by an inverter generating 90 VAC at 400 Hz. Initially, I left the backlight on all the time, but the intensity degraded rapidly. To prevent this, I added an external switch to control the backlight.

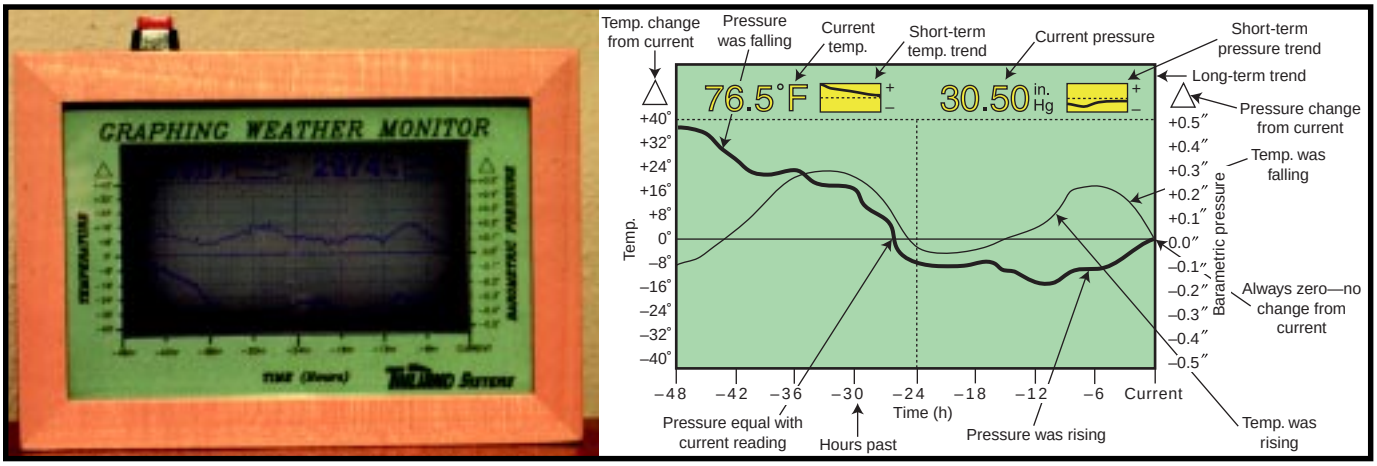


Photo 1—*The graphing weather monitor is housed in an attractive picture frame. The graphic on the right illustrates the information that is displayed by the unit.*

The external RAM (U2) is simply an 8K × 8 industry-standard (ancient) 6264. The only requirement here is memory size and speed. The overall schematic is shown in Figure 3. During image writing, the LCD is turned off (~100 ms) while the MCU stores the display image in external RAM.

I borrowed the temperature circuit from the heat unit calculator design that I mentioned earlier. This circuit has been rock solid in a number of units and environments.

The temperature sensor (U12) is an AD590. This laser-trimmed sensor provides a calibrated current proportional to the absolute temperature. The signal conditioning circuit uses a conventional 7650S chopper-stabilized op-amp (U5) with offset trim (RV1) and gain (RV2) potentiometers.

The chopper-stabilized amps offer good long-term stability by zeroing out offset currents on a regular basis, and they are virtually immune to DC drift. A potential design improvement could be made by replacing the potentiometers with a software-intensive calibration and scaling routine (e.g., storing setup parameters in an EEPROM).

	+000h	+001h	+002h	+003h	+004h	+005h	+006h	+007h	+008h	+009h	+00Ah	+00Bh
+000h	0	0	0	0	0	0	0	0	0	0	0	0
+03Ch	0	0	0	1	1	1	1	1	1	1	0	0
+078h	0	0	0	1	1	1	1	1	1	1	0	0
+0B4h	0	0	0	1	1	1	1	1	1	1	0	0
+100h	0	0	1	1	0	0	0	0	0	0	0	0
+13Ch	0	0	1	1	0	0	0	0	0	0	0	0
+178h	0	0	1	1	1	1	1	1	1	0	0	0
+1B4h	0	0	1	1	1	1	1	1	1	1	0	0
+200h	0	0	1	1	0	0	0	0	1	1	1	0
+23Ch	0	0	0	0	0	0	0	0	1	1	1	0
+278h	0	0	0	0	0	0	0	0	1	1	1	0
+2B4h	0	1	1	1	0	0	0	0	1	1	1	0
+300h	0	1	1	1	0	0	0	0	1	1	0	0
+33Ch	0	0	1	1	1	1	1	1	1	1	0	0
+378h	0	0	0	1	1	1	1	1	1	0	0	0
+3B4h	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4—*Here you see an example of a font block for the number five. The column and row headings are memory offsets, whereas the zeros and ones indicate the bit state.*

The pressure sensor provided a special challenge. Initially, I tried an uncompensated pressure sensor and found it to be a much better temperature sensor than a pressure sensor. I went back to the drawing board after examining the compensation curves, running through the math, and realizing that I didn't have a suitable temperature chamber to design the circuit.

Fortunately, Motorola came to the rescue with the MPX5100. This sensor is factory calibrated to provide a temperature-compensated pressure output and it has proven to be stable and reliable. Its output (U7) is conditioned by another 7650S chopper-stabilized op-amp (U6). Offset and gain are adjusted using RV5 and RV3, respectively.

For the ADC, I used a MAX110, which is a highly capable two-channel ±14-bit serial chip. The resolution is more than sufficient for this application, and its reference voltage is derived from an LM4040 (D2) precision reference adjusted to 2 V by a tap off RV4.

I ran into some difficulty with the power-up condition on the MAX110. As it turns out, V_{SS} must come up before V_{DD} or the analog portion of the chip won't function. Because V_{SS} is generated by the flyback converter, it takes some time to come up. I fixed this by delaying the onset of V_{DD}.

FIRMWARE DESIGN

With the hardware in place, I set out to craft the firmware. I used assembly language for all of the code. The code is deterministic (no interrupts) and spends most of its time transferring through a software state machine kernel that dispatches various routines at appropriate times.

As you probably figured out, the firmware is responsible for just about everything. Here are the major tasks that are embedded in this application:

- time slicing between refresh and data acquisition/image generation
- timekeeping
- data scaling
- data storage management
- graphing calculations
- display image mapping
- control and addressing of external image and data RAM
- display refresh (~100 Hz)
- font generation
- trend-line generation (both short term and long term)

It's beyond the scope of this article to discuss the details of each of these facets, so I'll expand on:

- border, axis, and line generation
- data to image bitmap conversion
- trend line data conversion to image graphic

The border, axis, and line generation are simply math functions implemented in loops to create the lines for the scale and axis. To determine where the line should go, I made an AutoCad drawing depicting the display (with corresponding RAM addresses) and inserted the lines where appropriate. This made it easier to compose the code because I could see exactly which addresses needed to be included in the code.

Because the LCD responds only to pixel data, any information presented must be converted to a bitmap. To map the display, I sketched a 12 × 16 font block because this ratio best fit my needs for the main data displays (see Figure 4).

The 0s and 1s represent the individual bits that are coded into memory. Each number on the left represents an offset row address, and each number on top represents an offset column location. The width of the font is larger than one byte (12 vs. 8 bits) so I used two bytes to represent each row.

Another graphic function is the transformation of the time-ordered data from the sensors to a trend line and graphic that depicts how the parameter changed over a period of time. I chose to store relative history data rather than absolute data to simplify calculations and reduce the size of the database.

Each value is adjusted relative to the current reading and is displayed accordingly. The data is stored in the external RAM from 00f0–0fff hex and 10f0–1fff hex. The external RAM is written in an efficient manner so it can be quickly retrieved for the LCD during the refresh process.

The trend routine contains a dot-connecting routine so the data is displayed as a solid line rather than a series of dots (significantly improving visibility). There's also a utility to draw a thin line and a thicker line to differentiate two parameters.

PULLING IT ALL TOGETHER

Having everything working on a breadboard is one thing, but having four working units that are packaged and relatively reliable is quite another. I quickly concluded that this job was not a wire-wrap job. Printed circuit boards were the only way to go.

I began searching for an inexpensive layout tool. I was glad to find a light version (that cost only $50 to register) of the ARES layout program. A copy of the ISIS companion schematic capture was also included. Unfortunately, the net list connection was deactivated but it was easy to work with the ARES layout tool and its autorouter did an outstanding job, given the constraints.

To keep other tooling costs down, I chose to go with a single-layer board using jumpers (0-Ω resistors) on the top layer. I was able to obtain four boards (including tooling and drilling) for $50!

The PCB is the same size as the LCD PCB, with matching mounting holes. This allowed the two boards to sandwich together and made a tight package that fit into the picture frame with a slight modification by a Dremel tool.

The overlay is a laminated laser-printed sheet containing the axis legend and other information that is affixed to the front of the LCD housing. The final specifications for the graphing weather monitor are shown in Table 1.

In *Seven Habits of Highly Successful People*, Stephen Covey recommends starting projects with the end in mind. That thought kept me going on this project. Fortunately, the project was on time and within budget, and it generated happy customers. Special thanks to my wife for her encouragement.

Mark Bauman is a registered professional engineer and the manager of the electrical engineering group at Key Technology in Walla Walla, WA. Mark has held an amateur radio license for 21 years. You may reach Mark at bauman@bmi.net.

SOFTWARE

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

SOURCES

PIC16C64
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

MAX110
Maxim
(408) 737-7600
Fax: (408) 737-7194
www.maxim-ic.com

DMF660N-EW
Timeline, Inc.
(800) 872-8878
(310) 784-5488
Fax: (310) 784-7590

HD61105, **HD61104**
Hitachi
(415) 589-4207
Fax: (415) 583-4207
www.hitachi.com

MPX5100
Motorola
(512) 328-2268
Fax: (512) 891-4465
www.mot.com

AD590
Analog Devices
(617) 329-4700
Fax: (617) 329-1241
www.analogdevices.com

FEATURE ARTICLE

George Novacek

Accurate Linear Measurement Using LVDTs

Need reliability in a hostile environment? LVDTs may be the answer. These transducers are widely used for converting mechnical displacement into electrical signals. George helps us understand how they work and how we can use them.

Ah, the ubiquitous LVDT! It's the most widely used device for converting mechanical displacement into electrical signal (usually a DC voltage linearly proportional to its core displacement).

In this article, I want to look at the LVDT from a practical standpoint. What is it? How does it work? How can you use it in your projects?

Besides theoretically infinite resolution and excellent linearity, LVDT's major claim to fame is reliability in hostile environments. Such reliability makes LVDT the obvious, and often only, choice for aerospace, chemical process, nuclear industry, or any other precision or safety-critical application under tough environmental conditions.

Of course, using an LVDT as part of a closed-loop control system with an embedded controller doesn't guarantee a failsafe operation. It's just the first step. I'll show you what it takes to develop a failsafe embedded controller (e.g., for aircraft landing gear).

Delving into the details of safety-critical design of embedded controllers opens an engineering can of worms that leads to such esoteric subjects as built-in test (BIT), testability, electromagnetic compatibility (EMC), reliability, and hazard-/failure-mode analyses.

I'll also discuss the most controversial (and sometimes incorrectly labeled) subject of embedded controller design—software reliability. This topic will shatter any remaining illusions that software is fun. And, hopefully, it will cause you to raise your eyebrows even higher every time you perform the three-finger salute.

Let's start by defining an LVDT. Linear variable differential transformer (LVDT) is a transformer-type linear displacement transducer that converts the rectilinear motion of an object mechanically coupled to its movable core into a corresponding electrical signal.

The LVDT has been around for nearly 100 years. It's used wherever precision, repeatability, reliability, and safe operation even in the most hostile environments are required. LVDTs are used in industrial processing, medicine, transportation, and aerospace industries where the environments range from cryogenic temperatures of 4 K (–450°F) all the way up to 1100°F and beyond.

LVDTs fit the aircraft industry well. You'd be hard pressed to find any other type of displacement transducer on aircraft. From the freezing temperatures of flight surfaces in stratosphere, to slush and sleet on landing gears, to the hot gases inside jet engines, LVDTs are unmatched for reliability, precision, and life expectancy.

LVDTs and their cousins (RVDT and DVRT, which I'll look into later) are made by many companies. U.S.

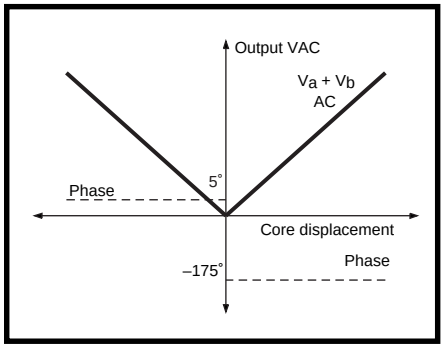


Figure 1—*LVDT secondary output voltage is linear with core displacement with abrupt phase reversal at null. Ideally, the phase is 0° and 180°, but small phase shifts like the 5° shown can be compensated by RC networks.*

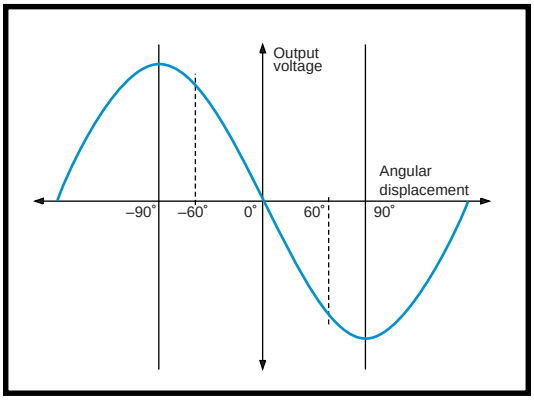


Figure 2—*RVDT output dependence on the angular displacement is close to sinusoidal, with the ±60° range linearized by core design. This graph shows a one-cycle RVDT. The two-cycle RVDT has the peaks 90° apart (as opposed to 180°).*

manufacturers include Macro Sensors (Schaevitz Technologies), Kavlico, G. W. Lisk, MPC, and Microstrain.

WHAT'S IN A NAME

As its name implies, LVDT is a transformer. It has one primary and two secondary windings. Photo 1a shows a cross-section of a small LVDT. The primary winding, located in the middle between two secondary windings, is supplied with a constant-amplitude, constant-frequency excitation voltage (carrier).

With the movable core (Photo 1b) located precisely in the middle such that the magnetic flux between the primary and both secondary windings is identical, the two secondary windings produce equal voltages, V_a and V_b. As the core moves, the ratio between V_a and V_b changes.

If you connect the secondary windings in series but in opposite phase, V_a and V_b subtract from each other. The resulting AM modulated carrier amplitude is linearly proportional to the absolute core displacement from the center, and the phase abruptly reverses 180° at the null (center).

Figure 1 shows the relationship between the core displacement and the output voltage and its excitation versus output voltage phase within linear operating range, which is usually up to 80% of the coil's physical length.

To satisfy just about any conceivable measurement need, LVDTs are routinely produced with linear stroke ranging from 0.005″ to 25″. The challenge in producing a long-stroke LVDT is to maintain uniform distribution of magnetic flux through the length of the coils.

The most serious drawback is price. Not even off-the-shelf devices are cheap, and aerospace-grade LVDTs can cost thousands of dollars each. Why not replace them with less expensive transducers? The answer to that question can be found by looking at the LVDT's list of unique features.

For one, an LVDT permits frictionless measurement. Physical contact between the moving core and the coils is unnecessary. Because there's no friction, there's nothing to wear out, giving an LVDT an essentially indefinite mechanical life. The physical separation between the core and coils means the core can be immersed in hot corrosive pressurized media, such as 3000-psi Skydrol hydraulic fluid, without a dynamic seal.

The inherently infinite resolution is achieved because of the frictionless operation (no hysteresis) combined with the induction principle of operation. The LVDT responds to minute core movements, limited only by the electronics' S/N ratio and display resolution.

The location of the LVDT's intrinsic null point is stable and repeatable. The isolation between the windings and the fast dynamic response are nothing to sneer about, nor is the incredible operating temperature range. After all, it's just a wire; there are no PN junctions to blow.

Last but not least, the LVDT is sensitive to the axial movement of only the core, providing excellent cross-axis rejection when used, for instance, in accelerometers.

RELATED DEVICES

Before discussing the LVDT's electrical characteristics, I should mention some related devices. The LVDT's cousins are the rotary variable differential transformer (RVDT), the linear variable transformer (LVT), the rotary variable transformer (RVT), the half-bridge LVDT, the differential variable reluctance transducer (DVRT), and the AC and DC LVDT, RVDT, and so on.

The RVDT measures angular displacement. Internally, the RVDT can be compared to an electric motor. It has three stator windings analogous to the LVDT windings. The core is an armature rotating inside those coils.

Figure 2 shows the relationship of an RVDT output versus angular displacement. It is near sinusoidal, with approximately ±60° linear range.

RVDT manufacturers can improve the linearity by profiling the rotating armature so you can obtain excellent linearity and a usable range of about ±85°. If good linearity is required beyond ±60°, a gear (often internal) is used to scale down the measured shaft rotation.

Unlike the LVDT, which can be designed in many different lengths to fit the requirement, the RVDT comes only as a one- or two-cycle RVDT. Figure 2 shows the output versus angular displacement characteristic of the one-cycle RVDT.

A two-cycle RVDT has its range cut in half by symmetrical design of the armature, so its linear range is only about ±30°. This range is useful in applications where the larger range is not needed (e.g., the movement sensor for aircraft rudder pedals, typically limited to ±27.5°).

Figure 3 shows the equivalent circuit of the variable differential transformer sensor. LVDT, LVT, RVDT, and RVT are electrically the same. Every-

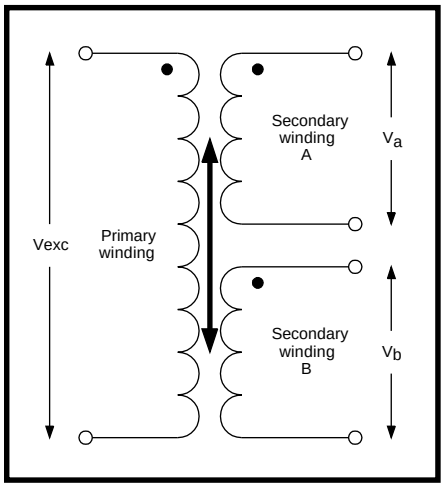


Figure 3—*The LVDT is "just" a transformer, with the dots marking the beginnings of the windings. With the secondary windings' ends connected together and the beginnings serving as outputs, the two secondary signals are added in opposite polarity, providing zero output, when magnetic flux between the primary and each secondary is identical.*

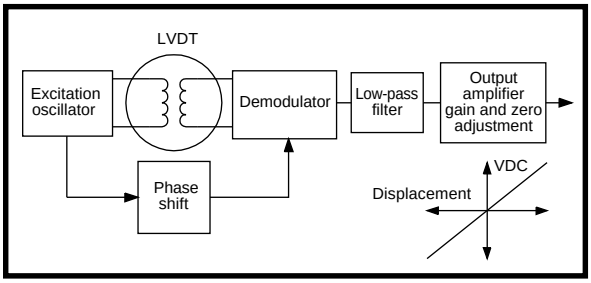


Figure 4—*The synchronous demodulator is gated at the positive zero crossing of the excitation signal to half-wave rectify the secondary. This will be a positive voltage when the phase is 0° and negative when the phase has switched 180°. The phase shifter is needed to compensate nonzero primary to secondary phase.*

thing I say about LVDTs applies to these devices, unless mentioned otherwise.

Remember the secondary windings A and B are connected in series with opposite phase, such that V_a and V_b are subtracted from each other. Originally, when most signal conditioning was performed by synchronous demodulation, access to the individual secondary coils was unnecessary (more on this later). Manufacturers connected the two coils internally and brought out two input (excitation) and two output leads to reduce cost and weight.

With the advent of monolithic ratiometric decoders about a decade ago, the need arose to have access to the connection point between the two secondaries—sometimes called center tap. Not a problem, as long as you understand that the coils are wound in opposite phase.

When the tap was required, European engineers began calling devices with the tap "self-testing" or "self-monitoring." Engineers in North America referred to devices without access to the tap as LVT and RVT, and devices with tap access as LVDT and RVDT.

A rumor developed that LVTs and RVTs were obsolete designs and therefore inferior. It's not true, but it is understandable: ratiometric decoding provides excellent performance, which had been nearly impossible to achieve.

To make interfacing simpler and using the transducer more attractive, manufacturers such as Schaevitz offer both AC and DC varieties. The AC LVDT is the device I talk about here.

The DC LVDT is precisely the same device with the interface (conditioning) electronics integral within the device. The DC LVDT simplifies interfacing by reducing it to two or three wires. However, it's uncommon in aircraft applications primarily because of its lower reliability and potential inability to survive the harsh operating environment.

To complete the family tree, I need to mention the half-bridge LVDT and the DVRT. Both are autotransformers with a center tap and a moving core. The excitation, often as high as 70 kHz or more, is applied across the autotransformer. The output voltage from the center tap and its phase have the same characteristics as the LVDT in Figure 1.

Because the device needs only three wires for interface, it is lighter and cheaper to manufacture than the three-coil LVDT. The signal conditioning required for the interface is similar to the LVDT.

ELECTRICAL CHARACTERISTICS

Before I discuss how to convert the LVDT output to the customary displacement-proportional DC voltage, let's review the transducer's electrical characteristics. After measuring the output voltage and phase, the results are as shown in Figure 1, provided that two crucial requirements are satisfied.

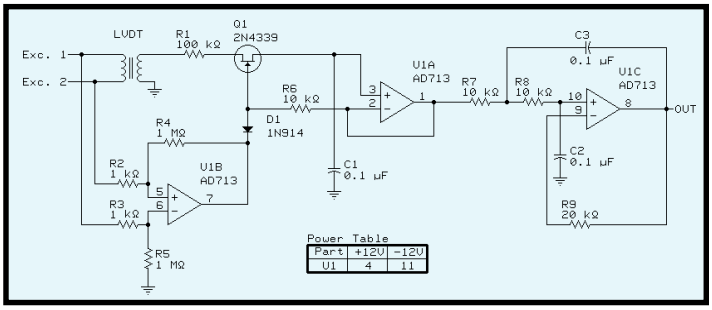


Figure 5—*A simple and reliable synchronous demodulator can be built with a few discrete components. The excitation signal is generated separately and must be very stable for the circuit to deliver acceptable results.*

First, the AC excitation must have constant amplitude. Also, the phase angle between the two secondaries when connected in opposite direction must be constant 180°. Ideally, the phase angle between the primary and the secondaries should be 0°; otherwise you need to adjust it for synchronous detection. At any rate, the phase angle must be constant unless a phase-independent signal conditioning is used.

Selecting the excitation voltage and frequency is fundamental for satisfying these requirements. The transducers come calibrated for a specific excitation signal, guaranteeing gain, linearity, and often 0° phase shift at the calibration voltage and frequency.

Although in theory the excitation frequency can range from a few hertz to megahertz, the lower limit should be at least ten times the maximum frequency of the mechanical movement you want to measure. If your closed-loop controller runs an aircraft nosewheel with ~6-Hz critical frequency, the excitation should be at least 60 Hz.

Older systems use 400 Hz, which is found on many aircraft as the main power-system frequency. But, higher frequencies provide better sensitivity, efficiency, and phase control. Without going into the details, the most common excitation frequencies today are between 2.5 and 4.5 kHz, with 3.2 kHz as a prominent choice.

The excitation signal, usually ~3 V RMS, must not saturate the LVDT core. It has to be a sine wave with low THD, typically less than 0.5%, for several reasons. First, the high harmonic content of a distorted excitation signal may adversely affect the transducer's accuracy and linearity. Second, the higher harmonics can cause unwanted interference with sensitive electronic equipment.

The coils typically exhibit 600-Ω impedance at the excitation frequency. They often have 1:1:1 turn ratios, with the secondary windings working into loads greater than 100 kΩ. The excitation driver needs to be properly rated

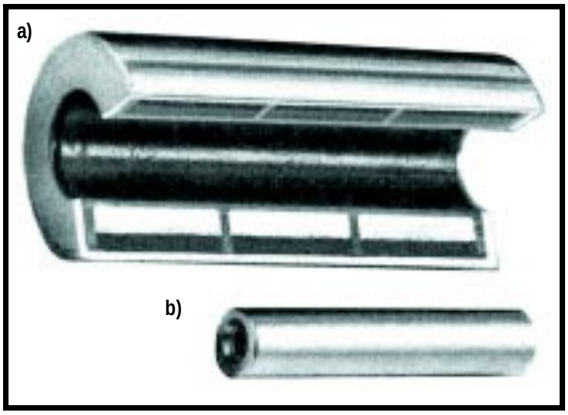


Photo 1—*These photos show the cross section of a typical small LVDT (**a**) and its core (**b**). The primary winding in the center and the two secondary windings flanking it are clearly visible. Long-stroke LVDTs use more sophisticated coil arrangements to ensure linear magnetic flux through the operating range.*

to avoid loading-related linear distortion. And finally, when several LVDTs are used in proximity, it's a good idea to synchronize their excitation frequencies.

Because many aircraft systems operate at 400 Hz, stay away from using harmonic frequencies for excitation (e.g., 3.2 kHz) unless that frequency is locked onto the 400-Hz distribution system. This is to avoid generating beat frequencies by inadvertent crosstalk that could fall within the control loop operating range and cause all kinds of embarrassing problems.

SIGNAL CONDITIONING

LVDT signal decoding (or signal conditioning) represents conversion of the AM modulated excitation frequency (carrier) obtained from the LVDT secondary windings into a DC voltage. The resulting voltage represents the core displacement.

There are several methods for signal conditioning. Here, let's look at the two most widely used ones—carrier synchronous demodulation and passive DC demodulation. Each has its advantages and disadvantages and I'll identify their strengths and weaknesses when I discuss decoding circuits.

No matter which method you choose, your signal conditioning circuit must generate a sinusoidal excitation signal of low THD with good frequency and amplitude stability as well as sufficient power. The circuit must demodulate the AM carrier (excitation) and filter its remnants out, and amplify the demodulated signal as needed for interface with the rest of the system.

In the diagram of the carrier synchronous demodulation in Figure 4, notice the phase shifter, which ensures proper triggering of the synchronous detector. Most LVDTs are calibrated for 0° phase at the calibration frequency and amplitude and therefore don't need the phase shifter unless they're going to be used outside the calibration spec.

Monolithic demodulators are available off the shelf from manufacturers like Analog Devices, so only the brave at heart would want to roll their own. But, it's worth looking at a discrete demodulator circuit if only to understand how it works.

The heart of the design in Figure 5 is an analog switch (Q1) that is synchronized with the positive-going zero crossings of the excitation frequency by voltage comparator U1b. Assuming 0° phase-shift between the primary and the secondaries, no phase control for the synchro detector is necessary.

Remember that through null transition the secondary output voltage phase abruptly changes 180° (depending on the core displacement from null), so positive or negative half-waves are switched to charge through R1 capacitor C1, which develops average voltage across it. Amplifier U1a is a buffer, followed by a Sallen-Key low-pass filter (U1c) to attenuate the remainder of the carrier.

Although the circuit is straightforward, its main weakness is readily apparent. The displacement measurement's accuracy is directly affected by the stability of the excitation amplitude, as well as the frequency. Frequency fluctuations can cause phase shift, affecting the operation and accuracy of the synchronous detector.

If the system needs to work with precision in a wide temperature range

environment, better solutions are needed. For that, I'll jump to the present where monolithic ICs (e.g., the AD698) make the problem simpler to handle.

In addition to synchronous demodulation (as in the discrete example), the AD698 monitors the primary excitation voltage V_a, calculates the secondary versus primary voltage ratio V_b/V_a, and modifies the system gain accordingly. This setup greatly reduces the excitation voltage fluctuations that are responsible for system gain errors.

Along with the complete decoding circuitry, the IC contains a stable precision sine-wave oscillator with power driver and adjustable frequency and amplitude. All operating parameters can be adjusted by a few external components. Figure 6 shows the actual application with AD698.

The design of component values is straightforward and relies on a few basic steps. First, determine the oscillator frequency based on the system requirements and the LVDT specification. The frequency is controlled by:

$$C1 = \frac{35\,\mu FHz}{f_{excitations}}$$

Next, determine the oscillator amplitude. Start with the manufacturer's recommended excitation amplitude, which is typically about 1–3.5 V RMS. Make sure the power dissipation of the AD698 is not exceeded and determine the secondary signal at maximum deflection (should be in the 0.25–3.5-V RMS range).

You can calculate the secondary signal from the turn ratio (if you know it) or the sensitivity, which is usually defined in millivolt output per volt excitation per mil travel (mV/V/mil). You need to optimize the excitation and set it by R1. R1 equal to 10 kΩ provides 3.5 V RMS. The AD698 app note provides a detailed discussion.

C2, C3, and C4 determine the bandwidth of the low-pass filters. They should be of equal values, such as:

$$C2 = C3 = C4 = \frac{10^{-4}\,FHz}{f_{subsytem}[Hz]}$$

The value is determined by the subsystem bandwidth requirements. If the desired system bandwidth is 250 Hz (a good value for many control applications), the resulting capacitance value is C2 = C3 = C4 = 0.4 µF.

The third step is to calculate gain for full-scale output, which is set by R2. R2 is set at:

$$R2 = \frac{V_{out}}{S \times d \times 500\,\mu A}$$

where *S* equals LVDT sensitivity in mV/V/mil and *D* equals full-scale displacement. Where V_{out} is ±10 V, sensitivity is 2.4 mV/V/mil, and displacement is ±0.1″, the equation is:

$$R2 = \frac{20\ V}{2.4 \times 0.2 \times 500\,\mu A} = 83.3\ k\Omega$$

Determining a closed-loop system bandwidth and how it relates to the system components is outside the scope of this article. But, as a practical note, 0.4 µF is just about the largest

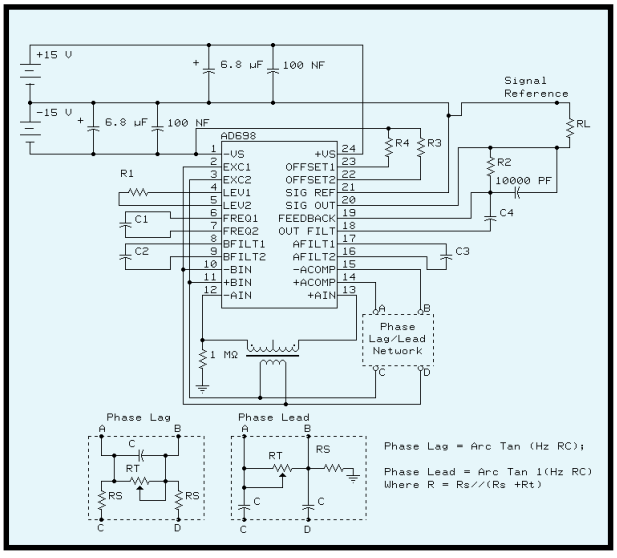


Figure 6—*The AD698 does everything the discrete circuit in Figure 5 does and more. It has a built-in low THD excitation generator and compensates for excitation amplitude variation. Phase-lead and phase-lag circuits are shown to compensate for LVDTs with other than 0° phase shift between the primary and secondary windings.*

capacitor you want to use with the IC and it must be a quality nonpolarized capacitor.

With the excitation frequency usually in the 3200-Hz range, I prefer to do the low-pass control in a separate stage with a Sallen-Key filter. Buffering of the demodulated output is an added benefit.

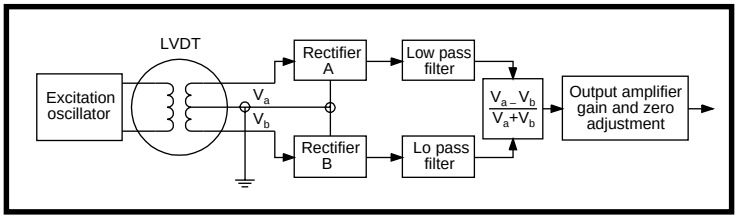


Figure 7—*The ratiometric decoder rectifies each secondary winding separately and compares the resulting ratios, making the precision of the displacement measurement independent of the excitation voltage and phase shift. The only proviso is that the sum of the two secondary voltages does not change with the core position—a condition satisfied in the majority of LVDTs.*

One nice feature of AD698 is that it enables you to work with a four-wire interface to the LVDT. But, the problems of excitation-frequency stability and phase-shift adjustment still remain, which brings me to the AD598.

The AD598 is the workhorse of the industry and performs ratiometric decoding of the LVDT position signal. It is insensitive to temperature, frequency, and phase variation throughout the entire MIL temperature range of –55°C to +125°C. The only minor disadvantage is that the AD598 requires access to the secondary tap.

The principle of ratiometric decoding is fairly simple (see Figure 7). The excitation frequency induces signals in the secondary windings V_a and V_b. The two signals are rectified, so they're insensitive to the phase shift, and the output signal is then calculated as:

$$V_{out} = \frac{V_a - V_b}{V_a + V_b}$$

The '598 performs the functions neatly and (like the AD698) includes the excitation oscillator onboard. The design procedure for the AD598 is rather close to that of the AD698.

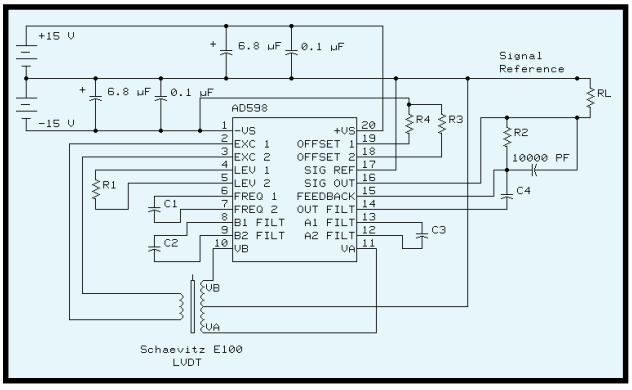


Figure 8—*The AD598 provides an excellent, yet very simple ratiometric interface for LVDTs. Only a handful of discrete components is needed to achieve avionic-grade precision throughout the full military temperature range.*

Although Analog Devices doesn't say so, my guess is that the two devices share some building blocks.

Besides the secondary tap access, the only other limitation ratiometric decoding imposes on the LVDT is that $V_a + V_b$ remains constant throughout the operational range. This requirement is met by the majority of LVDTs. Although I can't assume that all LVDTs automatically satisfy this requirement, I've never seen one that doesn't.

Designing the circuit around the AD598 follows the same steps as for the AD698: determine the excitation frequency, the excitation amplitude, and the bandwidth requirements of the system (see Figure 8).

The excitation frequency determined by the value of C1, the excitation amplitude determined by the value of R1, and the bandwidth controlled by C2 = C3 = C4 are calculated exactly the same as for the AD698. Full-scale output voltage is calculated using:

$$R2 = \frac{V_{out} \times (V_a + V_b)}{S \times V_{exc} \times 500\,\mu A \times d}$$

where V_{out} is the desired full-scale output, $V_a + V_b$ is the sum of the secondary voltages (constant), S is the sensitivity, and d is the maximum displacement. The difference between the devices is the addition of the full secondary to full primary transformation ratio:

$$\frac{V_a + V_b}{V_{exc}}$$

Note that both devices show four additional resistors—R3, R4, R5, and R6. Under normal operation (i.e., dual power supplies and symmetrical output), those resistors aren't used.

R3 and R4 serve to inject output voltage offset all the way up to a unipolar output. R5 and R6, in conjunction with R3 and R4, permit single-supply operation. The app notes provide step by step instructions for determining the resistors' values.

CLOSING THE LOOP

Almost a century ago (January 2, 1906), Porter and Currier obtained U.S. Patent 808,944 for using a variable differential transformer as a contactless AC motor reverser. Since then, we've seen many refinements to the original LVDT, but the principle has remained the same.

When it comes to measuring displacement with precision and with little worry about the environmental effects, the LVDT remains unsurpassed. If only they were a little less expensive....

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

REFERENCES

Analog Devices, LVDT Signal Conditioner, AD598 app note, 1989.

Analog Devices, Universal LVDT Signal Conditioner, AD698 app note, 1995.

E.E. Herceg, *Handbook of Measurement and Control*, Schaevitz Engineering, Pennsauken, NJ, 1986.

SOURCE

AD598, **AD698**
Analog Devices
(617) 329-4700
Fax: (617) 329-1241
www.analog.com

Sensing Water with Multiple Electrodes

FEATURE ARTICLE

Jack O'Neill

When it comes to sensing water via its conductivity, electronic sensors don't enjoy the same widespread implementation as mechanical devices. The reason? Contamination. But Jack doesn't think the electronic sensor is in over its head.

What's the best way to sense the presence of a conducting liquid? When safety or protection of property is important, mechanical is better. Now, those of you in electronics may object, but the real world has already spoken. I'll use a few boating applications to demonstrate what I'm talking about. After all, that's where this project originated.

Only about 1% of bilge pump switches in boats are pure electronic. Sewer lift stations, standard sump pumps, well pumps, and boiler water control are almost all mechanical.

My son is a captain and when he asked me to design an electronic alternative for the mercury-float bilge pump switches in his boat, I confidently replied, "Nothing to it." That was five years ago.

Not long after I made that arrogant statement, I realized that it was impossible to design a reliable water sensor for a boat that operates in saltwater because the switch must also be able to recognize fresh water (e.g., rain).

Some boilers use two-electrode conduction-type water sensors, which is possible because the water is relatively clean and has a known conduction range. With bilge water, there's no guarantee of clean water. A design that recognizes dirty fresh water is best.

A few years after realizing that reliable operation of dirty-water sensors was probably impossible, I remembered a strange (or maybe just nonintuitive) result I'd encountered when making water measurements. In the process of rechecking those results, my method for sensing a conducting liquid with multiple electrodes was born and awaiting delivery to the patent office.

Sensing the presence of water with electrodes is easy enough for a simple ohmmeter-type circuit to handle. And, using an AC current source to prevent electrolysis permits the design of an inexpensive device. Then why are almost all water-sensing devices mechanical, even though (in the case of pump control) they need to provide some sort of on/off hysteresis?

Mercury-float bilge pump switches use a pool of mercury inside a float switch to change the float's center of gravity. Sump pumps and sewage lift stations use mechanical slippage to get up several feet. Why not use simple electronic sensors placed where you want the pumps to start and stop? You can even incorporate any useful delays.

Most water sensing is done mechanically because electronic methods are unreliable. The reason is contamination. The same contaminants that support using conducted electrical current to sense water will conduct as much current when precipitated onto the electrode substrate as when in the water. That doesn't make it impossible to build a reliable sensor, but it does make it more difficult, even if the switch can be cleaned regularly.

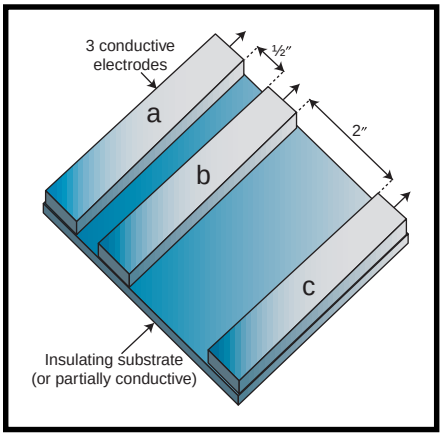


Figure 1—*This basic electrode configuration demonstrates the principle. A practical sensor can have more elements and they may encircle the housing.*

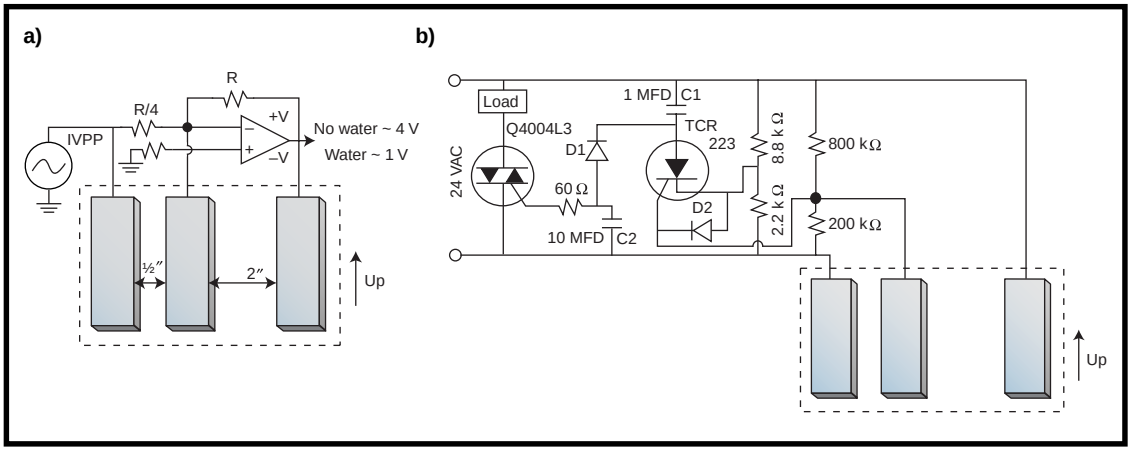


Figure 2a—*Water is sensed by the op-amp output level change. When the electrodes are immersed, the output is 1 V. When they are dry, the output is 4 V.* **b—***This circuit uses a bridge to sense water. The electrodes are spaced in the same ratio as the bridge resistance ratios. If the ratio is not close to 4:1, current flows in the gate of the TCR223 (SCR). This will gate the triac into conduction to operate the AC load (relay, motor, etc.).*

Electronic sensors can be used in water boilers because the sensor is designed with a sensitivity to match the water. Tap water, although variable, doesn't present the wide range of conductivity encountered in bilges, lift stations, and sumps.

But, enough with all that. Let's see what it takes to get the job done.

SENSOR DESIGN

An ohmmeter, with probes spaced ½″ apart in distilled water, gives a reading over 100 kΩ. In tap water, the reading ranges from 10 to 30 kΩ. And in seawater, you get readings below 1 kΩ. The design problem is clearly the need for a wide range of sensitivity.

Because the electrodes must be supported by something, a conducting layer eventually precipitates onto whatever supports the electrodes. A sensor that's sensitive enough to detect rainwater would have a hard time ignoring any buildup left by saltwater. Building a reliable water sensor using only two electrodes is quite impossible.

This example uses probes spaced ½″ apart. If the spacing is increased to 2″, would the resistance be four times greater? No. The concept of resistivity gets in the way here.

You'd think a sample of resistant material four times longer would have four times the resistance. In the case of a volume of conducting liquid, only a small percentage of the conduction is in a direct path between the electrodes. So, over small changes of spacing, there is virtually no change in resistance. This is the key to the patented method. How do we exploit this phenomenon?

Contaminants that precipitate on the electrodes and the supporting substrate create a planar resistor, much the same as a film resistor. The resistance of such a resistor is proportional to its length. When a three-dimensional volume of water surrounds the electrodes, the resulting resistance is nearly the same at ½″ spacing as it is at 2″.

When there's no water present and a layer of contaminants remains, the respective resistances are in a ratio of the respective physical spacing (here, 4:1). Now you can tell for sure when no water is present. This is the basis for the electrode arrangement in Figure 1.

Note that the electrodes span the width of the supporting substrate. This arrangement keeps the resistance (result of surface contamination) linear, with respect to the distance between them. It's important that an intimate connection exist between the substrate and the electrodes to reduce any connection anomalies to a minimum.

A sensor like the one in Figure 1, when immersed in water, presents a resistance from *a* to *b* about the same as from *b* to *c*. Any contaminant resistance appears to the connected electronics in the spacing ratio (4:1).

There are many inexpensive and highly reliable circuits that can be used to exploit this simple relationship. Before exploring such applications, note that no existing device uses this principle and the mechanics of the electrode mounting and substrate material are the key to a successful application.

Mechanics may not be interesting, but they are important. As for the electronics, that's the easy part.

CONSTRUCTION

In Figure 2a, an op-amp is connected so the gain is controlled by fixed resistors R and R/4. The electrode spacing is also in a 4:1 ratio because contamination between electrodes would form planar resistors in parallel with R and R/4.

It's reasonable to expect the contaminants to deposit uniformly enough to maintain a resistance ratio of about 4:1. With 1 V in, the output would be 4 V with no water present and 1 V when immersed in water.

In the case of a two-electrode sensor, a water–to–no-water threshold current must be established. If a deposit

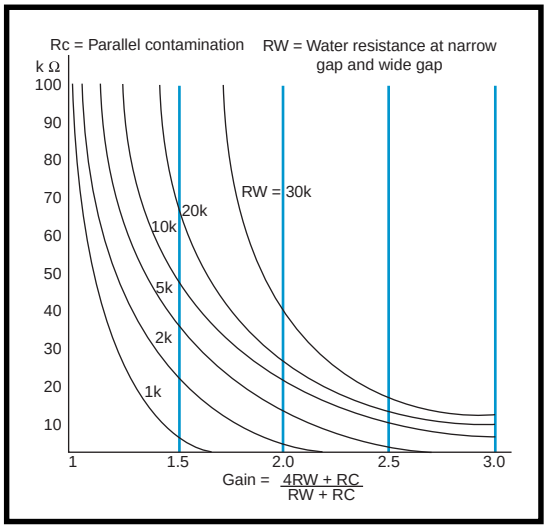


Figure 3—*This graph plots Rc against amp gain. These curves are typical effective water resistances for various types of water and small electrodes. These plots, plus field data about contamination, enable you to design a reliable sensor.*

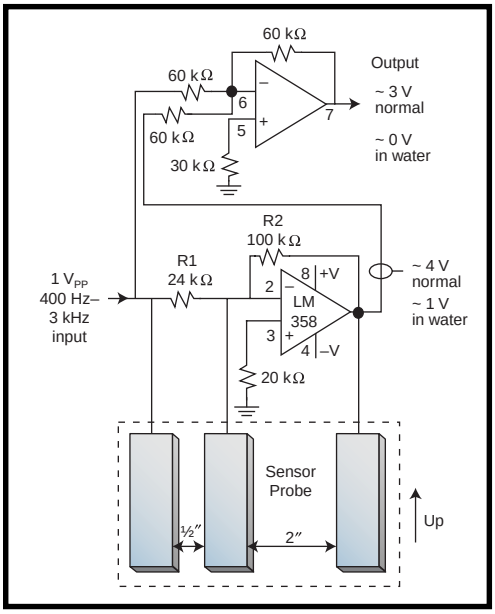


Figure 4—*This circuit adds a summing amp to the circuit in Figure 2. The summing amp compensates for input loading changes and other variable conditions. This extends the ability to differentiate between water or no water at the electrodes.*

of contamination lets the threshold current flow, there's no way the circuit can discriminate between it and water. There don't appear to be any electronic tricks to fix the two-electrode sensor.

My sensor uses a resistance ratio change to alter the gain of an op-amp or unbalance a bridge-type circuit (see Figure 2). The sensitivity is reduced as contamination builds up, but at least it doesn't give false indications of water.

There are several ways to overcome reduced sensitivity. As contamination builds up, it appears in parallel with R and $R/4$. The resulting ratio remains close to 4:1.

The ratio (or amp gain) doesn't reduce to one when immersed in water because the water's resistance (RW) is in parallel with the input ($R/4$) and feedback (R) resistors. RW is about the same across each electrode gap. The effective gain of the op-amp, considering these resistances and the contamination's resistance, is:

$$\text{Amp gain} = \frac{R}{R/4} = 4$$

With contamination, it is:

$$\frac{R_c}{R_c/4}$$

and with water present, it is:

$$\text{Gain} = \frac{R_c RW / R_c + RW}{\left(R_c/4\right)RW / \left(R_c/4\right) + RW}$$

$$= \frac{RWR_c\left(RW + \left(R_c/4\right)\right)}{RWR_c\left(RW + \left(R_c/4\right)\right)}$$

$$= \frac{4RW + R_c}{RW + R_c}$$

Figure 3 presents plots of the gain at several values of water resistance. These plots show the gain of an op-amp when the three electrodes are immersed. The normal op-amp gain when the electrodes are dry is shown in Figure 2a. The circuit discriminates by using an output of around 4 V compared to a selected lower threshold voltage, which the circuit can reliably recognize.

With no contaminants on the substrate, the output is either 1 or 4 V. The contamination changes the gain of the op-amp from 1 V when immersed, to a higher gain when water is not present because the planar resistors,

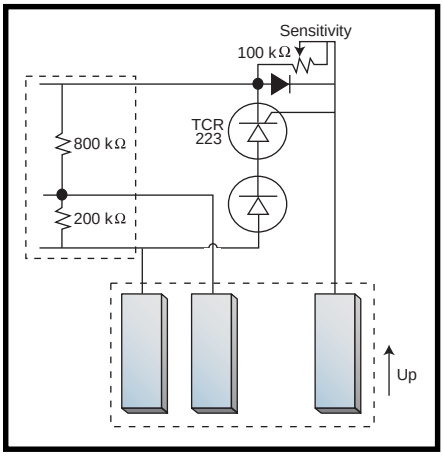


Figure 5—*This circuit can be added to Figure 4 to monitor current through contamination when dry. Current flows into the gate of the SCR. If if exceeds a threshold (e.g., a two-electrode sensor), action can be taken, either automatic or manual.*

although in a 4:1 ratio, end up in parallel with the apparent water resistance. The result is a ratio of less than 4:1.

To set an op-amp's gain, the typical values of fixed resistors are 100 and 25 kΩ. Figure 3 shows the effects of a contamination build-up starting at 100 kΩ (wide gap). The water resistances are also typical for clean tap water to seawater with electrodes spaced at ½″ and 2″.

Suppose you select 3 V as a threshold op-amp out. More than 3 V means no water; less than 3 V means water.

The graph shows where the contamination reduces the sensor's ability to sense water. For example, if the water is 30 kΩ and the wide-gap contamination is 15 kΩ, then any more contamination would defeat the sensor.

But, if the contamination is less than 15 kΩ, any water would be sensed. The response is the reverse of the two-electrode sensor and is also less severe. A two-electrode sensor with a threshold current equivalent to 30 kΩ reports water when 30 kΩ of contamination is present. In most practical applications, the reduction in sensitivity under discussion is not a problem. (Besides, there is a remedy, if necessary.)

Two electrodes of a three- or a separate two-electrode sensor can measure contamination as it is deposited. The three-electrode sensor knows when the water is gone. If, before it loses sensitivity, some preset level of current is exceeded while no water is present, an alarm or shut-off action can be triggered.

The TCR223 in Figure 5 is set to react before the basic three-electrode sensor got close to too low a sensitivity for its application. The circuit in Figure 2b is a more precise version of Figure 2a. A summing amp compensates for varying conditions of the signal by subtracting the effective input signal from the output, leaving only the amount resulting from amplifier gain.

Figure 2b is a bridge-type circuit. Standard 24-VAC control voltage is used to operate a relay or some other AC load, and the gate circuit of the TCR monitors the bridge. With no water present, there's no gate current.

When water is present, the resulting unbalance turns the SCR on for one-half cycle charging C1. D1 transfers this to C2, which amounts to a voltage doubler.

You don't need to double the voltage, but you must isolate any DC from the bridge arms. The (–)DC voltage turns the triac on to power the load. D2 completes the AC cycle through the probe to prevent electrolysis.

The TCR223 needs a gate current of ~3 mA to be triggered. SCRs with 12-µA sensitivity are available. Figure 5 is an add-on circuit that senses when contamination becomes excessive. The LED lights or an optical isolator takes any action desired. If the LED is lit when the sensor is not detecting water, then it's because of the contamination.

Of course, design improvements are always possible, particularly through the use of microcontrollers. Unfortunately, I don't have room to discuss them here. If you contact me to license the patent, we can discuss them then.

Jack O'Neill has worked as a technician and engineer in electronics since it was called "radio." He has spent the last 15 years designing and developing products for personal and individual applications. You may reach him at icapp@bellsouth.net and via www.icapplications.com.

SOURCE

TCR223
Teccor Electronics
(972) 580-7777
Fax: (972) 550-1309

INTERBUS CONTROLLER BOARD

The **IBS PC ISA SC/I-T** controller board enables any PC (ISA bus) to interface to an interbus I/O network using the advanced functions of Generation 4 firmware. The functionality of this generation firmware provides up to 255 bus segments, 16 device levels, 512 devices per configuration, up to 4096 I/O points per configuration, and up to 62 devices with parameter channel.

Logical addressing and data preprocessing are provided on the controller board, and software drivers are available for most popular third-party application programming packages. High-level programming drivers are available for DOS, Windows 95/NT, and OS/2. The board can also be used without drivers by accessing the board's dual-port memory.

The IBS PC ISA SC/I-T controller board is a half-size board suitable for micro-sized PCs. The board provides direct inputs for local switches, accessible from the application program. A watchdog circuit resets the I/O in the event of application program failure. A hardware-compatible version in the PC/104 format is also available.

Applications include the direct connection of hardware, ranging from simple sensors and actuators to intelligent field devices to the PC via the interbus.

Phoenix Contact, Inc.
(800) 586-5525 • (717) 944-1300
Fax: (717) 944-1625
www.phoenixcontact.com

RUGGED SBC

The **2111 Industrial SBC** is a rugged, Pentium-based PC/104-*Plus* single-board computer designed for outdoor and mobile applications such as vision, security, and data-acquisition systems. Featuring over 250,000-h MTBF, extended operating temperatures from –40°C to +85°C, and the ability to operate under extreme shock and vibration conditions (50-G shock/10-G vibration), the unit is ideal for harsh environments.

The 2111 is based around a Pentium MMX or AMD K6 processor operating at speeds up to 266 MHz. It supports up to 64 MB of EDO DRAM and up to 72 MB of onboard flash disk memory. Two 16C550 serial ports, an ECP/EPP parallel port, dual EIDE on PCI Local bus, a floppy controller, dual USB ports, as well as keyboard and mouse ports are included.

The board also features an integrated onboard 10/100Base-T network, flat-panel display interfaces, and an onboard switch-mode DC-DC converter that provides 3.3-V output. The 2111 supports a range of OSs including QNX, VxWorks, DOS, and Windows 95/NT.

Toronto MicroElectronics, Inc.
(905) 625-3203
Fax: (905) 625-3717
www.tme-inc.com

NouveauPC

edited by Harv Weiner

MULTIFUNCTIONAL PC/104 MODULE

The **MiniModule/SES** multifunction module provides Ethernet, serial/parallel, and SCSI controllers in a highly integrated PC/104-compliant module. An NE2000-compatible Ethernet interface supports 10-Mbps data-transfer rates and provides direct connection to twisted-pair (10BaseT) media. Alternate external media (e.g., thick coax [10BaseE] and optical fiber) can be interfaced via the onboard AUI interface.

Two PC-compatible serial ports are implemented with 16C550-type UARTs that have 16-byte FIFO buffers for fast throughput. One serial port can be configured to support RS-232 or RS-485 operation; the other is RS-232 only.

The parallel port is an enhanced bidirectional port with support for the IEEE-1284 (EPP/ECP) standard. An internal FIFO buffer and DMA transfer modes allow greatly increased data rates. This port can be used as a standard PC printer port or programmed as a DIO port to provide digital sensing and control functions.

The SCSI interface is based on the Adaptec AIC6370 SCSI-II–compliant controller and supports up to 10-Mbps data rates in fast synchronous mode. The AIC6370 is fully supported with driver and utility software for compatibility with popular desktop and real-time OSs.

The MiniModule/SES sells for **$169** in quantity.

Ampro Computers, Inc.
(408) 360-0200 • Fax: (408) 360-0220
www.ampro.com

INDUSTRIAL-CONTROL MODULE

Tri-M System's **IR104** provides 20 optoisolated inputs and 20 high-capacity relay outputs on a single fully PC/104-compliant module. Each optoisolated input can accept either AC or DC input voltages ranging between 3 and 24 V. The relay contacts are rated at 5 A at 250 VDC to permit the direct control of small motors or solenoids.

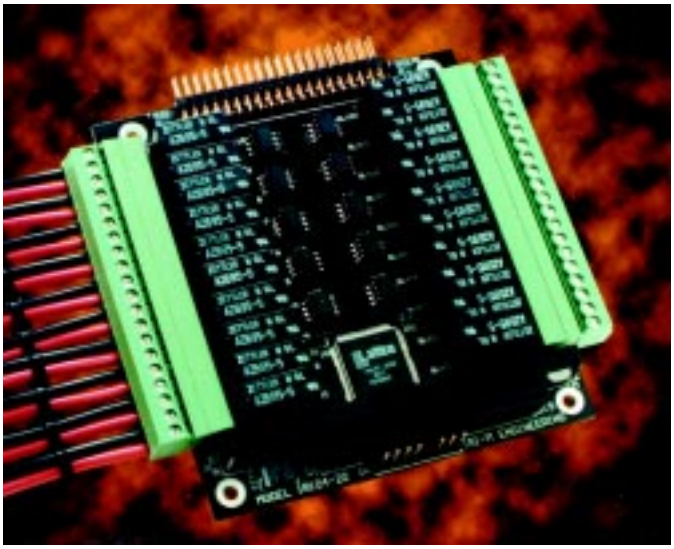
When it is connected to any PC/104-compatible CPU, the unit offers a reliable, safe, and economical alternative to older PLC-based technology used in automation of industrial monitoring and control.

The IR104 provides a convenient means to control external digital devices while maintaining electrical isolation from the equipment. Programming control of hardware is accomplished using standard desktop OSs and application software.

A Lattice in-circuit-programmable gate array easily decodes the I/O memory address on the ISA or PC/104 bus and is field programmable to custom addresses or custom I/O algorithms. This feature enables inputs to be read through an I/O memory read, and the output relay can be controlled through an I/O memory write.

The IR104 sells for **$325**.

Tri-M Systems, Inc.
(604) 527-1100
Fax: (604) 527-1110
www.tri-m.com



Nouveau PC

Richard Ames

PalmPilot Application

Using Open Source Tools for Development

Did you know the PalmPilot is a programmable device? Probably not. Richard shows us how the same feature that lets us link to a desktop PC and download our daily schedule can be used to download new application programs.

Scientists are seemingly rare these days, but certain characteristics make identifying scientists easy, should one walk near your workbench. When travelling between the study habitat and the lab habitat, a scientist often uses a pacing gait, which isn't very efficient but does make it easier to think about scientific problems.

Also, the left shirt pocket is often puffed out by a small wire-bound notepad containing notes on what the scientist is thinking about and what will be considered next. When sharing a habitat with a scientist, the ordinary engineer's attention is often fixed on the small notebook, which surely contains a wonderful archive of great ideas.

That's what I thought, but I realized that I couldn't just run out to the drugstore, buy a notepad, and start filling it with insightful notes. After all, wire-bound shirt-pocket notepads are not programmable. Being an engineer, I need to be able to write code.

Well, it's the end of the twentieth century and little programmable notepads are available at the local office supply store. Of the pocket-sized programmable devices, the 3Com PalmPilot is one that has a healthy developer community that shares tips, tools, and source code to ease the path to application development for the platform.

It may not be immediately apparent that the PalmPilot is a programmable device, perhaps because it is pitched primarily as an organizer and extension to your desktop computer. However, the same desktop PC-link that enables you to transfer the data in an address book or schedule can also be used to download new application programs into the PalmPilot. From the PalmPilot's perspective, an application looks like another internal database.

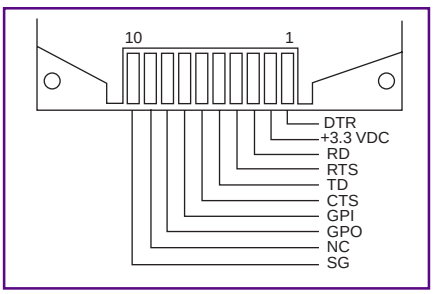


Figure 1—The connector on the back of the PalmPilot provides access to serial communication signals and general-purpose I/O.

You can also download an interpreter into the PalmPilot and develop applications entirely on the device—a native hosted environment. There are interpreters for BASIC, C, Forth, and Lisp.

With some of these systems, you can enter your program using the onboard memo writing application and execute the program with the interpreter application. This may bring back the thrill of hanging out in the back of the classroom covertly trying to set up your programmable calculator to display real-time graphics with an eight-digit seven-segment LED display.

But, the PalmPilot has a lot more than an LED display and there are more accommodating application-development environments than the native environment of the PalmPilot itself. The first PalmPilot applications were developed on Macintosh computers using a tool chain targeted to generate code for the PalmPilot.

Listing 1—This resource script provides the information for the main form for the MeterReader application shown in Photo 1.

```
FORM idReadForm 0 0 160 160 USABLE NO FRAME BEGIN
TITLE "MeterReader"
FIELD ID idLocField AT 20 20 AUTO AUTO NONEDITABLE
FIELD ID idReadField AT 20 50 AUTO AUTO EDITABLE
BUTTON "<prev" ID idPrevButton 40 120 AUTO AUTO
BUTTON "<next" ID idNextButton 60 120 AUTO AUTO
END
```

The original cross-development environment (from Metrowerks) has been ported for use in Windows environments and has been bolstered by other cross-development solutions. One of them has an interesting price tag—it's free.

The GNU C compiler has been a longstanding focal point for free software development tools. Now there's a package for the GNU compiler that enables you to develop PalmPilot applications.

You don't need to stop at the compiler. It's possible to assemble a development environment for the PalmPilot or other targets entirely from free open-source software. This software goes from the OS that boots the system to the text editor, the target simulator, and the download utility.

I'll show you how to put your own code into the PalmPilot. And just for fun, let's do it all using free software.

BRINGING UP GNU/LINUX

Linux is a Unix-like OS developed by Linus Torvalds while he was a student at the University of Helsinki in Finland. The OS has grown from being a very ambitious personal project into something of a cult. When the Linux kernel is mated with the Unix-like tools produced by the GNU project and joined by additional application software, it creates an environment that rivals your average desktop box loaded with a commercial OS and applications. Some highlights of its directory structure are listed in Table 1.

The process of assembling, installing, and maintaining the components composing a GNU/Linux system can take a lot of effort. Luckily, the task was taken on by a number of organizations that produce Linux distributions. The first steps on the road to your own Linux system are to find a suitable distribution and select some hardware to run the software.

Which Linux distribution to use is yet another one of those areas of the computer world that inspires spirited discussion. Choosing a distribution affects the ease of installation, maintenance, and the ability to upgrade your system. There's no perfect answer. The references point you to more discussion on this topic, so I'll lead on through example by discussing the the Debian GNU/Linux distribution.

DEVELOPMENT HARDWARE

Your development system needs at least a '386-family processor, 200 MB of hard disk space, a VGA display, and 4 MB of memory. If you have an older machine, you may want to put it to work running Linux. The latest hardware always seems more satisfying though, and Linux has support for multiprocessor motherboards.

The system can be set up to boot multiple OSs or you can dedicate the system to running Linux. My system here runs only Linux. In a world of change, you can expect something in this installation process to change as well (check www.debian.org/debian/install.html).

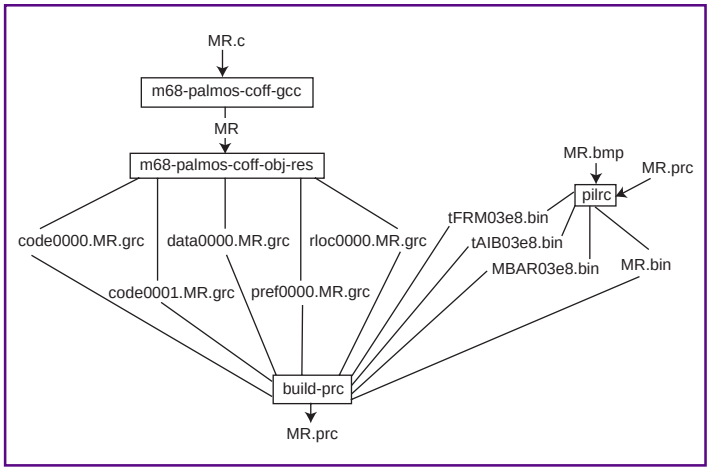


Figure 2—The PalmOS development tools are used to create the resource components that make up the application and then gather them into a prc file that can be loaded into the PalmPilot.

BOOT DISKS

To start installing your Linux system, you need a boot disk, also called the rescue disk. The image for that disk is available from the Debian web site or one of its mirrors. The mirrors usually provide faster download times and ease the burden on the main server.

You also need five disks' worth of compressed base-system software and a driver disk. You should have a spare disk handy for writing an emergency boot floppy once the system is installed on the hard drive.

The disk images can be retrieved using a web browser or ftp program. A typical disk-image file has a path such as /debian/disks-i386/base14-1.bin. Not all of the installation diskettes are in MS-DOS format, so a special utility (`rawrite2`) is used to write the disk image files to floppy disk. This utility is available at the distribution site.

Here's an overview of the installation process with the scenario I installed on an '486-based system with a 212-MB hard disk and 16 MB of RAM. It can receive information from the outside world through a 1.44-MB floppy drive and a modem.

Once you've prepared your seven special Debian boot disks, put the rescue disk into the drive and reboot the machine. The initial screen should inform you that you are about to transform your machine into a Linux box and it gives you your first chance to turn back.

Assuming that you don't have any attachment to what is on the hard drive, pressing Enter boots a minimal Linux kernel, spits out more than a screen's worth of detailed information about your system hardware, and starts in on the process of configuring your system.

The configuration dialogs use a simple text screen-oriented user interface that lets you use the tab key to move between fields and Enter to move on to the next screen. After setting up your basic video and keyboard type, it's time to divvy up your hard drive for the new OS. This step can be done with the `cfdisk` program, which should automatically start when you get to this point in the installation.

There's plenty of advice available on the best way to set up these partitions, depending on the size of the hard disk and system memory. In my example system, a typical configuration is an 80-MB root partition, a 100-MB user partition, and a 32-MB swap partition.

The root and user partitions should be set to type Linux ext, and the swap partition should be type Linux Swap. Select the Write option to write the new settings to the disk before leaving `cfdisk`.

The next step should be to initialize the swap, root, and user partitions. The root file system is then mounted and the OS kernel will be copied there. Once you have the OS on the hard disk, you are prompted to provide the driver's disk so that additional device driver support can be loaded.

If you have a minimal system, you don't need anything special from this disk. If you want to expedite the process by installing from a CD-ROM, you may need to load a CD-ROM driver.

The next step is configuring the network, which is only a matter of thinking up an appropriate name for your machine. The details of setting up your link to your ISP come later in the process.

Now it's time to feed the remaining disks into the machine, configure the clock, set up to boot into Linux, and make a backup-bootable disk. Once all of this is done, you can reboot the machine and load the kernel directly from the hard disk.

You'll be prompted to set up accounts for the superuser and yourself. Then, you can set up the PPP connection. Here you need IP addresses for name servers, user name and password, and the number to dial.

Now you have a base system on your hard drive, but it's not very useful. You're bound to miss the familiar environment of editors, compilers, GUIs, and the copy of Tetris. Don't worry, all of these will be coming because they've been set up in neat little packages.

Listing 2—The boilerplate main loop for a PalmOS application doesn't leave much room for creativity. After the initial form is set up, execution continues in the event processing loop.

```
DWord PilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)
{
  Int err;
  if (cmd == sysAppLaunchCmdNormalLaunch) {
    err = StartApplication();
    if (err)
      return (err);
    FrmGotoForm(MainForm);
    EventLoop();
    StopApplication();
  }
  return 0;
}
```

These packages are special-purpose files that integrate all the information needed by an installation utility so the contents can be installed and configured and ready for immediate use. At this point, you have the opportunity to load whole sets of packages, minimal system requirements, or the full-fledged Swiss army knife setup.

If you want to pull your software in over the PPP connection, it makes sense to skip the options and individually select the packages you want. This option is offered next from within a utility called `dselect`.

With `dselect` you can specify PPP as your access method and select specific packages. The packages can also be loaded from a Debian CD-ROM. Some software packages depend on other packages, and `dselect` advises you in selecting a set of compatible packages.

You need the following packages for PalmPilot development: xserver, man-db, pilrc, prc-tools, and xcopilot. While browsing through the list of packages, you're bound to see others that you want to check out, so add them to the shopping cart, too.

Now it's time to go through the checkout line behind the Install Packages menu item. If all goes well, the PPP connection transfers the files and after several hours you receive a report that the software installation was successful. While you're waiting, track down that Debian CD-ROM. After `dselect` finishes, you can start exploring your new system.

One of the first things you may want to do is edit a file. In the Unix world there are two common text editors—vi and emacs.

vi is popular in part because it is the most likely editor to be found on a Unix system. If you find yourself wanting to edit a file on an unfamiliar Unix system, chances are, you can find vi and get the job done. vi also happens to be rather capable.

emacs is also common and quite programmable. Say you're in the middle of an edit session and want to review your e-mail without switching to a new window. Extensions enable emacs to display your e-mail from within the editor and do a number of other useful tasks. You may never even need to leave the text editor.

There are other PC-like text editors available such as joe and xjed. joe will be comfortable if you're happy with the default DOS screen editor. xjed is more of a programmer's editor that takes advantage of the X Windows environment.

For more information about migrating from DOS to a Unix-based system, see the sidebar on page 42.

Command	Description	Command	Description
/bin	Essential command binaries	/tmp	Temporary files
/boot	Files for the boot loader	/usr/X11R6	X Window system files
/dev	Device files	/bin	User commands
/etc	System configuration files	/doc	Details on installed packages
/home/username	Your home directory	/include	Standard C headers
/lib	Essential shared libraries	/lib	Libaries for programming
/mnt	Temporary file system mounting point	/man	Home of the man pages
		/var/lock	Lock files for resources
/sbin	System binaries	/log	System event log files

Table 1—Here are some highlights of the directory structure in a GNU/Linux system.

INSIDE THE PALMPILOT

The PalmPilot is built around a Motorola 68328 DragonBall processor running at 16 MHz. There's also a 32-bit address space that provides up to 4 GB of memory.

The PalmOS and a set of built-in applications live on a 512-KB or 1-MB ROM on a plug-in memory card, which

Table 2—Here is the PalmPilot family, including discontinued models. Memory capacity follows the usual generous curve.

Product	PalmOS	ROM	RAM	Processor
Pilot 1000	1.0	512 KB	128 KB	68328
Pilot 5000	1.0	512 KB	512 KB	68328
PalmPilot Personal	2.0	1 MB	512 KB	68328
PalmPilot Professional	2.0	1 MB	1 MB	68328
Palm III	3.x	2 MB	2 MB	68328
Palm IIIx	3.1	2 MB	4 MB	68328EZ
Palm V	3.1	2 MB	2 MB	68328EZ

contains between 128 KB and 2 MB of pseudo-static RAM, depending on the model. The external data bus is 16 bits wide to reduce cost.

The system communicates to the outside world through its serial port, which has OS support for interrupt-driven output at 56 kbps. CTS and RTS signals are brought out to the contacts at the base of the device as shown in Figure 1, and there are eight hardware buttons on the system.

The system is designed to run for 40 h on two AAA alkaline cells. Because the device is likely to spend most of the time in sleep mode, the batteries are good for a few months of use. But, for constant active access to the PalmPilot or for a possible fixed application, there are suggested ways to patch a constant power source through the external connector.

Remember, the PalmPilot's purpose is to provide an electronic notepad. Thus the backlit 160 × 160 pixel LCD is overlaid with a digitizer and capable of four levels of gray. PalmOS provides up to 50 points/s from the digitizer with 0.35-mm accuracy.

PalmOS APPLICATIONS

PalmOS applications have special features that make them different from the usual embedded system software. The software in a simple embedded device can initialize the system hardware and memory and then go into an endless loop, checking for conditions and calculating responses.

Even though the underlying OS kernel does preemptive multitasking, PalmOS applications are single-threaded event-driven programs. The applications need to listen for special messages from the OS and play nicely with other applications.

The applications do have some characteristics that should be familiar to embedded-system software developers. The application must be frugal with memory because the application code and data come directly out of the system memory (which can be as little as 128 KB in the Pilot 1000). This memory allocation is less of a concern with more recent members of the PalmOS family, thanks to 2 MB of system memory (see Table 2).

There are no disk drives in the system, so instead of using a file system for application programs and data storage, a database is used to store applications, their data, and their state information. So, applications open and close databases that are maintained in nonvolatile memory rather than disk-based files. The OS is specially designed to provide efficient access to the database records, even though they may be scattered through the system memory.

PalmOS also makes good on its privileged position between the hardware and your application by interpreting pen strokes on the writing surface so your application can be fed characters instead of tracking *x-y* coordinates. There's also a rudimentary beep from the PWM on the 68328, a timer with 10-ms resolution, a real-time clock that's always running (set to wake up an application at an arbitrary time), and a TCP/IP stack.

SAMPLE APPLICATION

Let's consider an application where the PalmPilot is used to help a meter reader collect and deliver readings from utility meters. The program presents a series of addresses to visit and the readings at each address are entered into the PalmPilot. At the end of the day, the information is downloaded into a database without having to be transcribed from a log sheet.

When envisioning this application, you're likely to think in terms of how the screen of the PalmPilot will look and how the user will interact with that screen. In implementing the application, the visual layout of the screen (or form, in Palmspeak) is defined by a resource script. This script specifies objects that appear in the form, such as buttons, fields, checkboxes, and other elements.

For the meter-reader application, you can imagine that the address to visit is prominently displayed and there's an area to write the meter

Migrating from DOS to Unix

DOS	Unix	Action
dir	ls	Show working directory contents
time	date	Show current time
ver	uname –a	Show OS information
type myfile.txt | more	cat myfile.txt | more	Display myfile.txt, pausing between pages
mkdir doc	mkdir doc	Create doc dir below working directory
cd doc	cd doc	Move to doc directory
del hello.c	rm hello.c	Delete the file hello.c
copy \home\sam\hello.c	cp /home/sam/hello.c.	Copy a file to the working directory
dir /s myfile.txt	find -name hello.c –print	Search for hello.c in this dir and all subdirs
unzip -d hello.zip	tar -zxf hello.tar.gz	Unpack a heirarchical archived set of files
help dir	man ls	Show online manual info

- Long filenames are available, and case is significant.
- DOS text files indicate the end of a line with the characters CR and LF. Unix text files use just LF.
- The default shell has a number of features. Old commands can be accessed with the up and down arrow keys. Information that has scrolled off the screen can be reviewed with Shift-Pg Up. File names can be completed by pressing Tab.
- To shutdown gracefully, change to superuser with the `su` command and enter `shutdown -h now`. Wait for the message "system halted" before switching off the power.

reading at this address. The next address can be called up by pressing a forward arrow and previously visited addresses can be reviewed by pressing a back arrow. Listing 1 shows what this might look like as a resource script, and Photo 1 shows the result.

Executable code is linked with the objects in the form. For example, when the user presses the forward arrow button, the next address to visit is displayed.

STARTUP ROUTINE

In a traditional embedded system implemented in C, code execution starts at the location pointed to by the reset vector, runs some startup code, and then jumps to the `main()` function. For a PalmOS application, you can consider the system software to always be running, and the execution of the application starts in the `PilotMain()` function. Listing 2 shows the boilerplate body of a PalmOS application.

The first step of the application is to check the launch conditions. For simple applications, the only condition handled is a normal launch (when the user selects the application icon from the Application picker screen). The `cmd` parameter that is passed in the `PilotMain()` call is checked to see if the application was called under this condition. If it was, the application starts. Otherwise `PilotMain()` returns.

Some applications must retrieve state information when they are first started so they can present a screen that looks the same as it did when the application was last running. This is the purpose of `StartApplication()`, which is specific to the application. A simple application can return false from this function, indicating that there were no problems in starting the application.

Execution continues by calling `EventLoop()`, which is the main event loop for the application. This loop processes a stream of events that are fed to the application by the OS.

Many events can be handled directly by the OS without any special processing by the application. But, the application is given dibs on these events as well, in case special processing is needed. The structure of the event loop is shown in Listing 3.

The first step is to retrieve the next event in the event queue with a call to `EvtGetEvent()`. Many events can be handled directly by PalmOS functions so `SysHandleEvent()` and `MenuHandleEvent()` are first given an opportunity to act on the event.

These functions return true if they were able to completely handle the event. If the application sees these calls returned true, it doesn't do any more event processing. Opening menus and pressing buttons on the PalmPilot are examples of events that can be handled completely by these OS functions.

If execution continues to fall through beyond these functions that the OS handles, the current event may be one that should be handled by the application.

STOP ROUTINE

The stop routine shuts down the application gracefully and saves the information that will be needed the next time the application is activated.

The heavy lifting is in `ApplicationHandleEvent()`, which looks for events such as button presses and updates the screen in response to these events.

MEET THE TOOLS

Once the application is coded, it must be converted into a format that can be

loaded into the PalmPilot. The compiler (m68k-palmos-coff-gcc) used in developing PalmOS applications is a patched version of the GNU C compiler. It is used to compile and link the application C source code (see Figure 2).

PalmOS expects the application source code to be divided into a number of resources, including ones dedicated to initializing application global memory and specifying application preferences. The `obj-res` utility in the code converter (m68k-palmos-coff-obj-res) converts the object code into the expected resources.

A PalmPilot application is implemented as a collection of resources, of which the application program code is just one. The other resources are prepared using the `pilrc` resource compiler. This utility takes a text description of the menus and other resources and converts them into binary resource files, which can be combined into a PalmPilot file.

Ultimately, the PalmPilot expects to receive a new application in a format known as `prc`. `build-prc` takes all the binary resources and builds them into a `prc` file.

`pilot-xfer` speaks the appropriate serial protocol to synchronize with the PalmPilot to update an application in the system memory. This utility is also used to backup and restore application data.

`x-copilot` is a virtual PalmPilot sitting on the screen of your development system.

Photo 1—The CoPilot is a virtual PalmPilot on your computer screen that can be used to test applications without downloading to a physical device.

Listing 3—The event loop in a PalmOS application processes events as they are served up by the system, giving first dibs to default event handlers.

```
void EventLoop(void)
{
  Word error;
  EventType event;
  do {
    EvtGetEvent(&event, evtWaitForever);
    if (!SysHandleEvent(&event))
    if (!MenuHandleEvent(NULL, &event, &error))
    if (!ApplicationHandleEvent(&event))
    FrmDispatchEvent(&event);
  }
  while (event.eType != appStopEvent);
}
```

This simulator enables you to quickly load your latest code and exercise it without the having to download to the physical device.

This sample session shows the process by which these tools are used together to build an application. `M68k-palmos-coff-gcc -O1 MeterReader.c -o MeterReader` compiles the main module and stores the output in the file `MeterReader`. The `-O1` specifies level 1 optimization.

`M68k-palmos-coff-obj-res MeterReader` takes the file produced in the previous step and splits it into the set of resource components that are expected by PalmOS. `pilrc MeterReader.rcp` runs the resource script through the resource compiler to generate resources for the forms and menus.

`Build-prc MeterReader.prc "MeterReader" Mete *.grc *.bin` is the final step and combines the resources into a properly formatted `prc` file that can be loaded directly into the PalmPilot. The command line specifies an output file (`MeterReader.prc`), an application name (`"MeterReader"`), and a creator ID (`Mete`). All the files with suffixes of `grc` (code resources) and `bin` (forms and menus) are combined to create the `prc` file.

Now, the `prc` file can be loaded into the simulator or the PalmPilot, but there will be no list of addresses to visit. The list is stored in a separate database file—`MeterReader.pdb`. This file can be built from an ASCII text file using `makepdb`, a utility specific to this application.

Once the database is loaded, verify that you can scroll through the address list and fill in meter readings. After the readings are filled in, a sync operation with the desktop system can move the database to the desktop, and the readings can be extracted.

Normally, this process is integrated into the PalmPilot desktop application using special software known as conduits. But, this support doesn't exist for Linux yet.

NEXT STOP

If you program with free tools, consider sharing your discoveries with the user community that put together the framework. We all benefit from well-considered postings, improved tools, beefed-up documentation, and shared source code. EPC

Richard Ames fulfills his need to write code at Oresis Communications, preferably in the lab habitat. You may reach him at richard_c_ames@yahoo.com.

SOFTWARE

The complete meter-reader application is available via the Circuit Cellar web site.

REFERENCES

3Com PalmOS reference, tutorials, and articles, www.palm.com/devzone/info.html, Software Developer Documentation

Debian GNU/Linux homepage and distribution, www.debian.org

DragonBall databooks, www.mot.com/SPS/WIRELESS/products/m68328.html

GNU project, www.gnu.org

Linux distributions summary, www.linuxresources.com/apps/ftp.html

Linux documentation project, www.metalab.unc.edu/LDP

PalmOS software and source code examples, www.palmcentral.com

PalmPilot resource compiler, www.scumby.com/scumbysoft/pilot/pilrc

PDA software development site, www.roadcoders.com

Pilot application development tutorial, www.iosphere.net/~howlett/pilot/GNU_pilot_SDK_Tutorial.zip

Pilot-to-GNU communications tutorial, www.iosphere.net/~howlett/pilot/GNU_Pilot_SDK_Tutorial.zip

Pilot programming newsgroups, news://news.massena.com/pilot.programmer.gcc, and www.acm.rpi.edu/~albert/pilot

X-Copilot, xcopilot.cuspy.com

SOURCE

PalmPilot
3Com
(800) 638-3266
(408) 326-5000
Fax: (408) 326-5001
www.palm.com

Real-Time PC

Ingo Cyliax

Astronomical Issues

Part 2: Radio Astronomy

As Ingo looks to the sky this month, he begins measuring radio emissions from Jupiter using an all-digital wide-band receiver. After describing its construction, he details the tests it endured and shows us the resulting spectra.

Karl Jansky accidentally discovered radio astronomy at Bell Labs in the '30s. While investigating the noise sources that affect radio communications, he found that there was a correlation between some noise sources and the sidereal day.

Remember that the sidereal time is the time fixed to the right ascension angles in the sky. Jansky found that the maximum noise seemed to come from a specific region in the sky, which turned out to be the center of our galaxy.

It seems that the radio spectrum has a large window that is unaffected by the earth's atmosphere. The radio spectrum starts above frequencies that are reflected by our ionosphere (less than 20 MHz, depending on the atmospheric conditions) and goes well into the microwave band over 1 cm (30 GHz). This range is rather large compared to the available visible spectrum.

One complex and interesting radio source that is easy to detect with minimal equipment is the radio emission from Jupiter. Like the earth, Jupiter is magnetic and traps ionized particles. Because Jupiter rotates, its magnetosphere acts as a synchrotron and generates radio emissions at various frequencies. Also, the moon Io contributes to the effects, making the emission patterns complex.

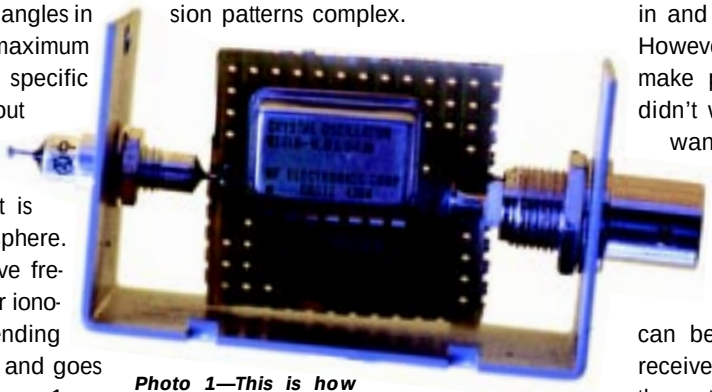


Photo 1—This is how breadboarding is done in radio work. By designing modules in small project boxes, like filters, preamps, and oscillators, it's easy to plug and play. For modules that use power, use small feedthrough bypass capacitors. Also, use linear power supplies and plenty of ferrite beads.

Jupiter emissions from about 20 to 40 MHz can be received with a simple shortwave receiver. But, it turns out that my shortwave receivers use the automatic gain control (AGC) feature to improve the quality of standard AM broadcast.

Without AGC, AM stations would fade in and out with atmospheric conditions. However, AGC makes it impossible to make power measurements. Because I didn't want to modify my receivers and wanted to make computer-based observations anyway, I decided to build an all-digital receiver.

RADIO BACKGROUND

Three types of receiver designs can be used—direct conversion, tuned receiver, and heterodyne receiver. Of these, the heterodyne receiver design is the most widely used. The "super" in a super-heterodyne receiver is a marketing gimmick used to sell heterodyne receivers in a market of tuned-radio receivers (TRFs).

All of these receivers do the same thing. They select a signal (the radio station of interest), amplify it, and demodulate it. In the case of AM signals, the demodulator is essentially a detector that measures the amplitude of the carrier.

The TRF measures amplitude by bandpass filtering the signal and amplifying it in several stages. The final stage drives a detector (usually a diode rectifier and a low-pass filter) and audio amplifier. It's a simple extension of a crystal radio set.

A heterodyne receiver uses a nonlinear component to mix the input signal from a local oscillator. Mixing is simply multiplying, so the input signal is multiplied with the local oscillator (LO) signal:

$$V_{if} = \cos(\omega_c t) \times \cos(\omega_{LO} t)$$

Using trig identity, this is equivalent to:

$$V_{if} = \frac{1}{2}\cos\left(\omega_c t + \omega_{LO} t\right) + \frac{1}{2}\cos\left(\omega_c t - \omega_{LO} t\right)$$

In other words, multiplying the input with a signal from an LO, generates two signals—one at the sum of the frequencies ($F_c + F_{LO}$) and one at the difference ($F_c - F_{LO}$).

In a heterodyne receiver, we select one of these products with a bandpass filter and it becomes the intermediate frequency (IF). The signal at IF is amplified, detected, and low-pass filtered. IF frequencies of 455 kHz and 10.7 MHz are commonly used, and some receivers have up to three stages.

Heterodyne receivers are popular because it's much simpler to build a high-gain amplifier at a low IF frequency than to build high-gain amplifiers at higher frequencies. Also, it's easier to build the selective filter needed to filter the station of interest at the fixed IF frequency than a variable filter in each TFR stage. Keep in mind that all of the filters in the TFR have to track the same frequency band at the same time.

A direct conversion (or synchronous) receiver is a heterodyne receiver that mixes the input signal with the exact frequency of the signal of interest. If you do the math, it gives you one product where the two cosines cancel each other, and another product that is double the frequency.

A signal without the carrier is the signal we already want. For AM modulation, the resulting signal is the amplitude information. In a direct conversion receiver, you simply need a low-pass filter and a high-gain audio amplifier.

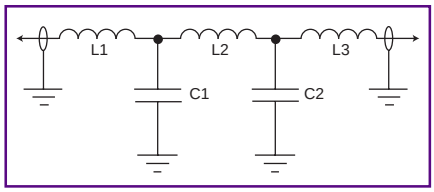


Figure 1—To configure the three-pole Chebychev low-pass filter, select the component values from a table (in the ARRL handbook). Capacitor values are usually fixed to standard values. Inductor values can be anything you like when you wind your own inductors.

Until now, I've only mentioned AM detection. You can generalize the detector in a heterodyne receiver, by mixing the input signal with both the in-phase and quadrature phase of a LO. This is done in the last stage of a receiver and results in two IF products, normally labelled as I and Q, which are then filtered separately.

With both the I and Q components, you can demodulate all of the common modulation techniques. For example, to demodulate an AM signal, measure the vector magnitude of the two signals:

$$V_{am} = \sqrt{I(t)^2 + Q(t)^2}$$

For FM signals, measure the phase difference:

$$V_{fm} = \text{atan}\left(\frac{Q(t)}{I(t)}\right)$$

Receivers that use this kind of demodulation technique are sometimes called diversity receivers because they can be used to demodulate just about anything (including digital-modulation techniques used in modems). This is especially true when the I

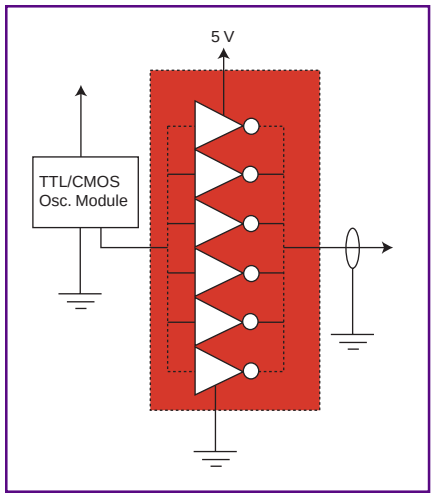


Figure 2—Standard oscillator modules don't usually have enough to drive a 50-Ω clock. The hex CMOS inverters are wired in parallel to boost current output. Don't attempt this with TTL drivers.

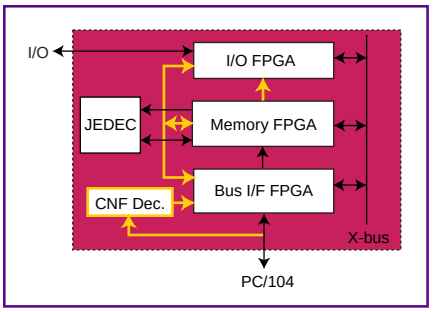


Figure 3—The I/O FPGA implements the tuner and first decimator. The bus I/F FPGA decimates and buffers the data for the host. The memory is not used in this design, although it could be used to buffer data for high-speed acquisition.

and Q components are digitized and processed with a DSP.

For radio astronomy, you usually want to measure the power of the receiver signal in a specific band. This equation is like the amplitude of an AM signal, but squared:

$$V^2 = I(t)^2 + Q(t)^2$$

Of course, "true" power would involve RMS values of the signals, but that's just a corrective factor of 1.414.

Let's turn our attention to digital receivers. By now, you've figured out that I'm not talking about a receiver with a digital-frequency readout. A digital receiver is a receiver that uses DSP techniques to select, demodulate, and detect signals.

The concepts are the same, but until recently, digital receivers have been in the realm of military receiver technology mostly because components such as high-speed ADCs and high-speed digital processors are expensive. It makes sense to use this technology when you want to build jam-proof receivers or receivers that need to be ultra-stable. As a side note, radio jammers usually target the IF frequencies of heterodyne receivers because that's where most of the gain processing is done.

Thanks to the explosion of the personal communication market, cell phones, pagers, and other components that are up to the task are available, making it more economical to build digital radios. One such component is the high-speed converter tailored for this application. It samples at over 50 MS/s.

I've played with both the Burr-Brown AD807E (51 MS/s) and the Analog Devices AD6620 (71 MS/s). Both are targeted for the cellular base station market, but they're perfect for my application.

THE PROJECT

My initial experiment was to construct a wide-band receiver with a 10-MHz bandwidth. This conservative approach provides a 20-MS/s sampling frequency and made it easy to design the signal-processing chain implemented in the FPGA I had available. In future articles, I'll discuss how to extend this receiver's frequency range.

Initially, I used a random long-wire antenna. I made no attempts to tune the wire or match it to the 50-Ω feed to the radio. This setup can be improved, of course, to improve sensitivity. A dipole antenna tuned to the frequency I'm interested in would work much better.

The antenna feed is terminated into a 0–10-MHz antialias filter built around a three-pole Chebychev filter. This filter has a cutoff frequency of around 9 MHz and a –3-dB point at around 11 MHz. Figure 1 shows the schematic. The component values were taken from a table in the ARRL handbook.

I made sure the filter was right by measuring it with signal generator and an oscilloscope. The filter performance isn't critical,

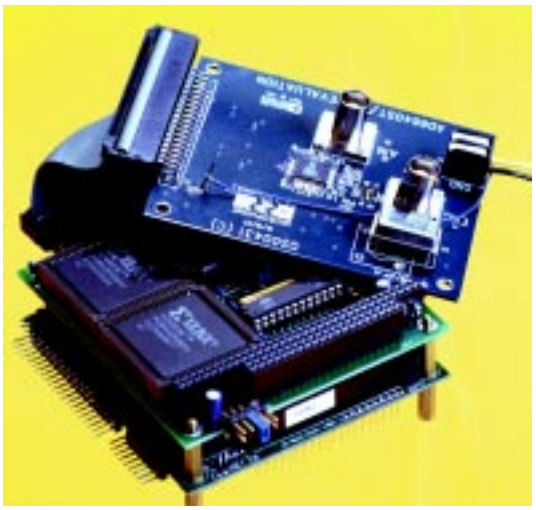


Photo 2—The PC/104-based FPGA is connected to the A/D eval module with a short ribbon cable. An antialias low-pass filter module, which would be added to the input, prevents aliasing from radio sources at higher frequencies. It also serves as a band-limiting filter for the preamp because it amplifies anything from DC to over 1 GHz.

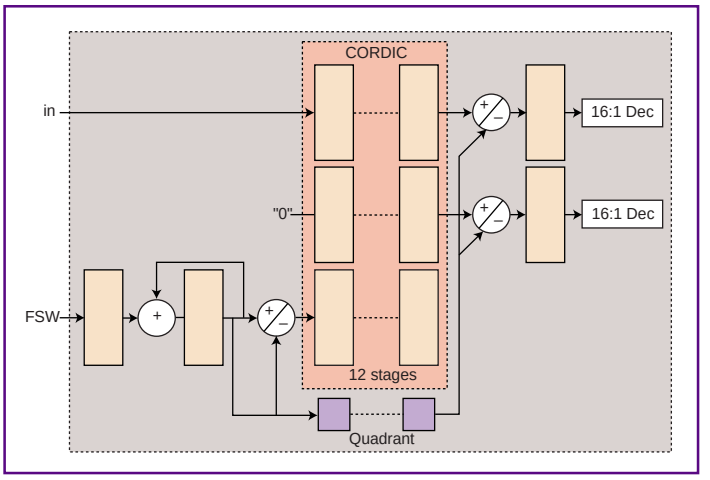


Figure 4—The FSW input sets the phase increment for the phase register. The phase register selects the instantaneous phase angle for the CORDIC computation, which computes a scaled cos/sin based on the input value from the ADC. The output of the CORDIC is decimated before being sent. Note that part of the phase indicates the quadrant of the computation because CORDIC can only compute between –90° and 90°.

unless you want to look at signals close to the top end of the spectrum. After the filter, I have a small FET-based preamplifier that boosts the signal by ~20 dB.

The filter and preamplifiers are housed in small metal project boxes with BNC coax connectors to connect them to the system. This technique is handy for "breadboarding" RF designs. For example, switching in a different filter is as easy as disconnecting the old filter and plugging in a different filter (see Photo 1).

You also need an RF signal generator and oscilloscope. My signal generator covers the frequency range I'm interested in and has an adjustable output attenuator.

That's pretty much it for the analog components. The converter evaluation board has a matching transformer to make the single-ended coax feed match up with the differential inputs of the ADC. It also has a connector for a sampling clock.

On the AD6620, the clock source can be an external signal generator fed in via a coax connector or onboard TTL clock chip. I chose the TTL-clock chip option. For driving the AD807E board, I constructed a TTL oscillator in a small project box for driving 50-Ω loads.

This step was challenging because normal TTL/CMOS oscillators can't drive a 50-Ω load directly. I had to boost the output of the oscillator chip with a hex CMOS inverters all in parallel. Figure 2 shows this circuit.

The Analog Devices and Burr-Brown ADCs are followed with bus drivers. Although the ADCs are 5-V devices, the bus drivers can be powered at 5 or 3.3 V to adapt them to your specific system needs.

And lastly, the outputs from the bus driver and a buffered version of the sampling clock are sent over a regular 40-pin IDC header. The signals are laid out so I can simply plug either board into my FPGA board with a ribbon cable.

If you put several connectors on the ribbon cable, it's possible to bus up to four of the FPGA boards onto a single ADC board. Photo 2 shows how the filter connects to the preamplifier (which feeds into the ADC board, allowing the ADC board to be wired into the PC/104 FPGA board).

The FPGA board is a Xilinx FPGA-based PC/104 module that interfaces with the ADC board and contains all of the high-speed DSP functions. Along with the availability of the ADCs, FPGAs are key components in making this project possible.

The signal processing is intensive and still somewhat out of the realm of most DSPs and certainly not possible with a Pentium (even an MMX). Note that you can have multiple FPGA boards with all of them receiving the digitized signal of the ADC module. With that arrangement, it's easy to build multichannel receivers.

The FPGA boards are organized as shown in Figure 3. There are three FPGAs and an option for adding memory. For my initial design, I put the high-end processing in the FPGA that services the I/O connector. Some of the low-level processing, the interrupt generator, and bus interface go in the FPGA connected to the PC/104 connector.

As I mentioned, a digital receiver operates on the same principle as analog receivers, except that all of the signal processing is done using DSP techniques

instead of analog components that exhibit the need for properties. The model I used is a direct-conversion diversity receiver.

In my receiver, after the signal is digitized, I mix it with the in-phase (I) and quadrature (Q) of a LO signal, which is the frequency of the signal of interest. The resulting I and Q signals are low-pass filtered until they have the bandwidth I need. In my case, the bandwidth is dictated by the I/O bandwidth from the FPGA board and the host system.

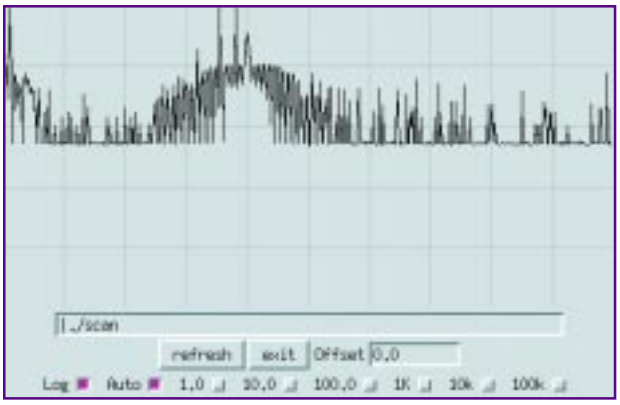


Photo 3—The initial 0–10-MHz spectrum is noisy because of dropped data in the host interface. The test signal is a 4-MHz carrrier modulated with a 1-kHz test tone. Spectrum plots are good tools when you're diagnosing signal-processing problems.

The LO is a numerically controlled oscillator (NCO). At first, implementing an NCO digital sine generator may seem simple. I use a register called a phase accumulator that keeps incrementing the phase of the sine wave by some value. This increment is called the frequency setting word (FSW).

The FSW is set to select the phase increment per cycle and can therefore be used to select the frequency. The output of the phase accumulator is sent to a ROM containing the sine values for each possible phase position. Now let's look at implementation details.

To build a 12-bit sine generator with a 12-bit phase accumulator in an FPGA, you need a 4096 × 12-bit look-up ROM. If you take advantage of the symmetry in a sine wave, you only need 1024 × 12.

Xilinx FPGAs have ROM function blocks that come in 16 × 2 and 32 × 1 varieties. To implement this look-up table, you need 32 × 12 blocks (384), plus a bunch of multiplexers to select one of the 32 blocks. This is good so far, but the biggest device I can use in the FPGA board only has 400 blocks and I haven't even thought about the multiplier yet. So far, no good.

CORDIC TO THE RESCUE

A few years ago, I wrote about an algorithm called coordinate transformation computer (CORDIC) ("Robot Navigation Scheme," *Circuit Cellar* 81). CORDIC is an efficient algorithm to implement transcendental functions (e.g., sine and cosine).

Because of its flexibility and small size, CORDIC is frequently used in digital signal processing. It's a bounded iterative technique and takes *n* steps to compute results to *n*-bit precision.

The complexity is roughly three *n*-bit accumulators, or six blocks in Xilinx. One logic block implements two bits of an accumulator. But, 12 steps to compute a 12-bit solution implies a clock rate of 12 × 20 MS/s (240 MHz), which is too fast for the particular FPGA I'm using.

CORDIC can be pipelined, which unrolls the loop into 12 separate components that can compute at the same time. The delay through the pipeline has a latency of 12 cycles, but it can produce a new result on every clock cycle.

Figure 4 diagrams the CORDIC pipeline, including the phase accumulator (which computes the instantaneous angle for the CORDIC input) and a 16:1 decimator. All this, including the interface to capture/synchronize and multiplex the output over a single 12-bit bus, takes 327 logic blocks.

This functional block performs these operations each clock cycle:

- NCO—one 16-bit add
- CORDIC—36 12-bit add/subtracts
- CORDIC—36 single-bit shift (divide by 2)
- decimators—two 16-bit adds

With a total of 75 operations performed at 25 MS/s, that's equivalent to 1875 MIPS!

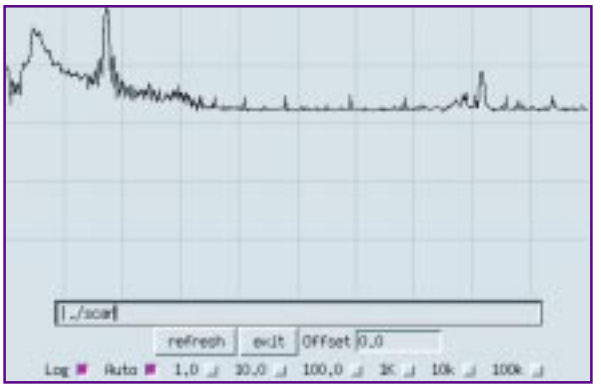


Photo 4—This is a live radio spectrum of 0–8 MHz on my antenna. Each horizontal tick is 0.8 MHz. The peak around 1.4 MHz is a local AM radio station. Each vertical tick mark is ~100 dB in this graph. You can see how much dynamic range a radio would need to pick up some of the weaker stations (little bumps) while not saturating on the local stations.

The NCO in this receiver can tune down to 305-Hz steps, defined by the size of the phase accumulator (16 bit). When the FSW is set to 0x0001, it takes 2^{16} steps to complete one phase. At 20 MS/s, that works out to 305 Hz. To get finer tuning steps, increase the number of bits in the phase register and FSW.

The PC/104 interface FPGA has another 16:1 decimator, the interrupt generator, and the bus interface. The PC/104 bus is fairly slow. I made measurements (*Circuit Cellar* 99) and determined that 400 KBps is about the maximum for this 8-MHz PC/104 bus implementation. I want to transfer I and Q samples, which end up being 4 bytes, so the practical limit is ~100 kHz.

Using the two cascaded 16:1 decimators, I got a baseband sampling rate of 78 kHz and a bandwidth of 39 kHz. For decoding AM signals, this bandwidth is too wide and doesn't provide enough selectivity, but it's sufficient for my power measurements. Although shortwave stations typically use 6 kHz, in the AM broadcast band you need about 10-kHz bandwidth.

The host system manages the interface to the receiver and is divided into a real-time component, which reads out the samples from the receiver and does the real-time processing. The host system also controls the receiver, which includes setting the FSW in the NCO to select the frequency.

For my initial experiment, I implemented a spectrum analyzer. The frequency scanning control is handled by a Tcl script that selects the frequency, reads a buffer of samples from the receiver through a FIFO from the real-time interrupt service routine, and computes the power in that spectrum. The data is saved to a file or can be viewed interactively with a Tcl/Tk. All of this is done under RT-Linux.

RESULT

I tested my receiver by taking spectrum measurements. I sampled the spectrum at 20 MS/s with a 4-MHz test signal. The test signal also had a 1-kHz test tone modulated on it. The result is shown in Photo 3. As you see, there's quite a bit of noise, which was caused by the host machine missing interrupts and samples being missed.

Photo 4 is a better graph, showing where I used a sampling rate of 16 MS/s to sample the RF spectrum from my antenna. This spectrum shows the bump of the local AM station (each horizontal division is 0.8 MHz).

Next I wrote a quick shell script and arranged for it to be called every 15 min. The output was saved in a data file that had a timestamp of the form `yymmddhhmm.out` for the filename. After a day of testing, I had a directory full of spectrum data.

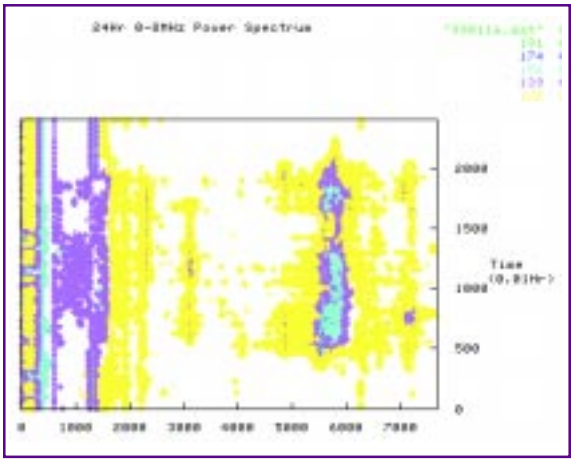


Photo 5—Here's a 0–8-MHz spectrum plot over one day. The horizontal axis is the frequency and the vertical axis is the time. The scan starts and ends at 12 PM. You can see how the AM band changes at 6 PM and 6 AM. Many AM broadcast stations have to lower the their power and transmission pattern at night. Note how the 49-m shortwave band (~6 MHz) opens up at night.

Photo 5 is the result of a typical (noon to noon) scan of the 0–8 MHz spectrum. The lower portion (left side) is rather constant and shows the clatter of the AM stations and low-frequency noise (mostly EMI). The various clusters of activity during the nigh are shortwave broadcast bands.

STAY TUNED

Next time, I'll discuss digital radios, digital filters, and implementation techniques. I'll show you how to enhance the radio's frequency range to include the Jupiter emission bands by using aliasing (a technique used in digital receivers). 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

American Radio Relay League, *The ARRL Handbook for Radio Amateurs*, ARRL, Newington, CT, 1998.
B. Brannon, *Basics of Designing a Digital Radio Receiver (Radio 101)*, Analog Devices, Greensboro, NC, 1998.
B.F. Burke and F. Graham-Smith, *Radio Astronomy*, Cambridge Press, Cambridge, UK, 1997.
J.J. Carr, *Secrets of RF Circuit Design*, McGraw-Hill, New York, NY, 1996.
J.J. Carr, *Practical Antenna Handbook*, McGraw-Hill, New York, NY, 1998.
E.C. Ifeachor and B.W. Jervis, *Digital Signal Processing: A Practical Approach*, Addison-Wesley, Reading, MA, 1993.
J.D. Kraus, *Radio Astronomy*, Cygnus-Quasar Books, Powell, OH, 1986.
P.J. Nahim, *The Science of Radio*, American Institute of Physics, Woodbury, NY, 1996.
G. North, *Advanced Amateur Astronomy*, Cambridge Press, Cambridge, UK, 1997.
W.L. Orr, *Radio Handbook*, SAMS, Carmel, IN, 1993.

SOURCES

ADCs

Analog Devices
(617) 329-4700
Fax: (617) 329-1241
www.analog.com

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

FPGAs

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

PC/104 FPGA board

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

Applied PCs

Fred Eady

Embedded Internet

Part 1: On the Network

To get your embedded device on the Internet, you have to be able to write TCP/IP-enabled code. This month, Fred lays down the networking knowledge you'll need to install the kernel, build the system, and pass the data.

Believe it or not, I have friends who aren't on the Internet in any way. To them, an URL is royalty in some foreign land. But the URL is probably an integral part of your everyday life and so is its well-known companion, TCP/IP.

Many of us use TCP/IP in some manner every day. We send and receive e-mail, gather needed information, maybe even play on the Internet. In fact, if you use the Internet at all, you eventually use networking principles based on TCP/IP.

The purpose of this discussion is not to teach you TCP/IP but to take it to the embedded level so you can write TCP/IP-enabled code. Let's begin by investigating the parts of TCP/IP and their relationship to the embedded platform model.

EMBEDDED TCP/IP

TCP/IP was designed from the ground up to be platform independent. This collection of protocols is standardized across the networking world. Although it's not the total solution to all networking problems, TCP/IP stands as a model of what all internetworking should be.

To implement an embedded model of TCP/IP, you must use a TCP/IP stack. Phar Lap's Realtime Embedded ToolSuite kernel includes a version of the TCP/IP stack. This stack is accessed using a set of APIs.

Bringing up a TCP/IP embedded application without a network is like owning a surfboard in Tennessee. You can stand on it, but you can't surf. Every machine on a TCP/IP-based network is called a host. That includes clients as well as servers.

The idea is that all hosts can communicate with each other, which implies that all hosts on all networks can communicate host-to-host across differing networks. Impractical? Maybe, but it's the vision. These hosts communicate by passing messages, accessing data, and transferring files. If that sounds like you on the Internet, it is.

I mentioned the host-to-host and network-to-network impracticality of such a system. If you really wanted to accomplish that task outside a small local network like the one in the Circuit Cellar Florida Room, the wire would have to be rather long.

Distance between host machines and the number of hosts presents a problem when attempting to communicate on any network. Now add multiple networks to the equation. In the TCP/IP scheme of things, it's nonsense to string a dedicated wire between a host in Florida to one in Canada.

First of all, you'd have to string a wire to every host on every network you wanted to communicate with. To compound the confusion, every host you directly wired to would have to do the same for every other host and network it wanted to talk to. Some networks out there do just that, but they're expensive to operate and are usually proprietary as to who owns them, who talks on them, and what protocols they possess.

Fortunately, in the world of TCP/IP there's an easier way to internetwork. The Internet uses gateways or routers to transfer data between hosts on differing networks. A gateway is a network machine that is physically connected to each net-

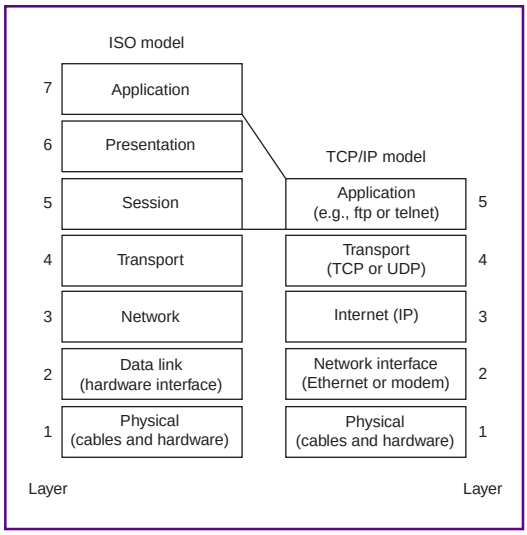


Figure 1—The ISO model is refined, but the TCP/IP layers still hold their own.

work it serves. Because each gateway has an address, it's relatively simple to connect a new network and its gateway device to an existing network superstructure.

A typical gateway or router is a standalone box driven by an embedded engine that's programmed to sit between networks and examine every message passed on the networks it is connected to. The router determines if a message should remain on the local network or be routed to another network.

Routers aren't designed to keep up with every host on every network, but they are programmed to map all of the routers and thus networks that they find electronically. Just as hosts seek out and communicate with other hosts, routers seek out and communicate with other routers. In the process, network-to-network communication is effected.

Network-to-network communications doesn't have to be across continents, either. Having found that it's best to experience the hardware and software I write about, I deployed routers between the living-room network and the Circuit Cellar Florida Room network. OK, now that you've changed your perception of me from nerdy to just plain weird, let's talk about addressing in an embedded network.

In the TCP/IP world, messages are usually short packets of data. Each data packet is addressed to reach a particular host on a particular network. An address may be an Internet or a physical address.

The Internet address is the familiar four bytes divided by dots scheme (e.g., 193.34.56.5). Each integer between the dots is a binary number that the embedded host sees as a host ID and network ID.

Which of the bits within the four octets determine the host and network IDs depends on the class of the network being addressed. I won't go into network classes here. Suffice it to say that the dot code is something only a host machine could love or remember.

Physical addresses are hardware addresses assigned to network cards installed in host machines. These physical addresses are set at the factory.

Here's where it gets interesting. The host network card can only recognize its physical address, not its Internet address. So, how do you get data to a host with a network card that doesn't know what an Internet address is?

A low-level protocol address resolution protocol (ARP) enables a host to discover the physical address of another host on the same network by broadcasting a general query. Data from the originating host is passed from gateway to gateway until it arrives at the destination gateway attached to the destination network.

The destination gateway uses ARP to determine the destination host's physical address and deliver data to it. The addressing data captured as a result of the ARP is stored in cache for later use. This process, in effect, enables the physical address to be concealed. Thus, the sending host need only address the target host by its Internet address.

To make it easier for humans, TCP/IP associates that dot address with a name. (e.g., circuitcellar.com). You'll soon see that TCP/IP does a lot of concealing, which, for the embedded network developer, is a good thing.

RELIABLE UNRELIABLE DELIVERY

Matter and antimatter—stuff that Star Trek episodes are made of. The same can be said for TCP/IP. The combination of transmission control protocol and Internet protocol wasn't done by accident but by design.

IP is the antimatter or unreliable component of the pair. IP is termed unreliable because there's no way to guarantee that a data packet will actually be delivered to its destination. All of the hosts in the Internet give their best effort to deliver an IP data packet. But, nobody cares how it looks when it gets there.

I spend more time in airports than I like, and it's funny how you notice things when you're bored out of your mind. For example, I liken an IP data packet to a piece of freight in the hands of the people loading freight onto an airplane.

They get the package (IP datagram) from a baggage cart (the host) and sling it towards the moving-belt ramp (Internet and gateways). Sometimes, the package hits the ramp hard and lands on the tarmac. Someone helping the freight handlers (other gateways in the Internet) picks up the damaged package (corrupted IP datagram) and either slings it back on the ramp (pass it on to the next gateway) or puts it directly on the plane (deliver the damaged IP datagram to the receiving host).

The plane (receiving host), although being loaded with a damaged parcel, never reported back to the cart (sending host) that the package (IP datagram) was damaged in transit. I spend too much time in airports. Anyway, let's continue.

My little story also implies that each piece of freight or each IP datagram is independent of any other IP datagram. Datagrams aren't connected in any logical or physical way. They are connectionless. IP is a connectionless protocol.

Before you characterize IP as a useless, unreliable means of data packet transportation, realize that it does have a helper—Internet control message protocol (ICMP). ICMP is a software component that enables gateways along the chain to report back to the originating host as to the health of the data packet in transport.

ICMP sends control messages to the Internet software on a host. Any host can use ICMP, but routers usually employ it.

Application	TCP Port	UDP Port
Echo	7	7
Daytime	13	13
ftp	21	–
Telnet	23	–
Simple Mail Transfer	25	–
Time	37	37
Whois	43	–
Trivial File Transfer	–	69
Finger	79	–
http	80	–

Table 1—Most of these applications are familiar, especially http.

STACK IT UP

The TCP/IP stack is composed of five layers, (see Figure 1). The physical layer is the simplest layer, with the application layer chiming in as the most complex. Each layer of the stack has a job to do.

Confusion between layers is eliminated by encapsulation. Each layer passes only properly formatted output to the next layer for processing. Encapsulation also enables each layer to treat data the way it prefers without affecting how the data is treated in other layers. For example, the transport layer likes to pretend that data enters in a constant stream but the Internet or IP layer sees data as separate connectionless datagrams.

To write a successful embedded TCP/IP application, you need to understand each layer's function. Physical layer is another way of saying hardware layer. This is the wire and cable and the network glue (routers) that connect the networks.

The network interface layer is the network interface card (NIC), modem, or serial interface that physically connects the host to the physical layer. This layer takes data from IP and formats it into network-specific frames. The frames are then transmitted to other hosts via the physical layer.

The network interface layer is where the address-resolution processes take place. Address resolution, in this sense, is mapping the IP address used by other layers of the TCP/IP stack to physical addresses used by the network cards. Protocols spoken at this level are Ethernet, PPP, and SLIP.

As we get further into embedding TCP/IP, you'll see that the IP datagram is the fundamental information that flows over the Internet. The Internet layer encapsulates messages passed from the transport layer and produces datagrams. Internet layer protocols include IP, ICMP, and ARP.

accept()	getservbyport()	select()
bind()	getsockopt()	send()
closesocket()	htonl()	sendto()
connect()	htons()	setsockopt()
gethostbyaddr()	inet_addr()	shutdown()
gethostbyname()	inet_ntoa()	socket()
gethostname()	ioctlsocket()	WSACleanup()
getpeername()	listen()	WSAGetLastError()
getprotobyname()	ntohl()	WSASetLastError()
getprotobynumber()	ntohs()	WSAStartup
getservbyname()	recv()	
getsockname()	recvfrom()	

Table 2—Thank goodness! Calls that almost read like English!

Between the application layer and the Internet layer lies the transport layer, which passes data between the application and Internet layers using TCP or UDP protocols. Again, matter and antimatter, with TCP (a connection-based reliable protocol) being matter and UDP (an IP-like connectionless unreliable protocol) as the antimatter.

TCP is what makes the transport layer famous. Unlike UDP, TCP uses a virtual connection to ensure that data arrives at its destination intact and in order. It accomplishes this connection via handshaking and special codes in each data segment.

The topmost layer—the application layer—is where the programmer reigns. There are more protocols used in this layer than I care to mention.

In simple terms, data flows from the application layer of the originating host through the TCP/IP stack and out the physical layer across to the physical layer of the destination host. Once the data enters the destination host's physical layer, the process reverses and data flows to the application layer, where it's processed.

One more subject stands between us and sorting out a TCP/IP application—ports. As you know, a host can run multiple Internet applications at once. How does the stack know where to route the messages?

TCP/IP assigns each network connection its own protocol port, an internal TCP or UDP address that is passed down the stack in the header of each data packet. IP sends host addresses; TCP and UDP send protocol port addresses. You may know of well-known ports like the standard reserved protocol port addresses in Table 1.

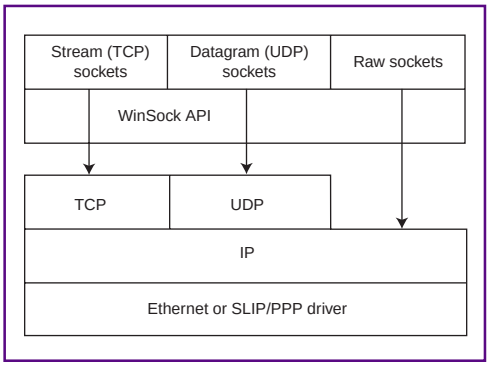


Figure 2—In this logical diagram, note the Ethernet NIC at the network interface layer.

SOCKET TO ME

The TNT Embedded ToolSuite supports the Berkeley subset of the WinSock 1 API. Remember that I spoke of the TCP/IP stack being accessed via APIs? Well, here we are. Table 2 lists the WinSock APIs supported by the Realtime Embedded ToolSuite kernel. Let's talk a bit about sockets.

A socket is Softwarese for network connection. A program creates a socket handle by calling `socket()`. Each socket handle forces the network stack to maintain connection information like protocol (TCP, UDP, IP), port numbers, and IP addresses (local and remote).

An application program can create three socket types. Because TCP resides in the transport layer and we know this layer likes to manipulate data as a stream of packets, the TCP socket is also called a stream socket.

Although UDP is in the transport layer, it associates with IP, which implies datagram activity. Thus, the UDP socket can be termed a datagram socket. Leave out TCP and UDP, and the socket is raw. Figure 2 depicts how a socket is created and what layers of the TCP/IP stack affect it.

A port number identifies a socket generated by an application, and an IP address identifies a physical host machine on a network. Together, they form a network address. The port number is generated when the server application invokes the network API `bind()`. The same is true for a client when the network API `connect()` is called.

Server apps tend to specify a well-known port so clients can easily connect, and the client apps let the network API assign a random port number. Take a look at Figure 3 as we build our application.

First, a client or server TCP/IP program must create a socket by calling `socket()`. `socket()` creates a data structure containing all of the necessary fields for data to maintain a network connection.

For example, `hsock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);` creates a client socket handle called `hsock`. The `AF_INET` tells the Realtime Embedded ToolSuite kernel to use a specific address family. The final two parameters are self-explanatory: this socket is a stream socket using TCP.

The server-side call is `hListen Sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);`. No difference except the handle name, which implies that the server will be listening.

Next, initialize the client socket data structure with the IP address and port number of the server you want to contact. `pHostEnt = gethostbyname (pHost Name);` can be used to perform a DNS lookup for remote host IP address.

If a DNS server is active on the network and `pHostName` is found, the IP address and port number are returned and put into the correct fields of the structure pointed to by `pHostEnt`.

On the other hand, if this call fails, `pServEnt = get servbyname ("finger", "tcp");` will get the well-known port and protocol from a local table or file.

On the server side, at this point it's only necessary to bind a well-known port number to the new socket. Once the server bind is complete, the server goes into a passive listening mode, where incoming connections are queued for the application to process.

The bind and listen code is `bind (hListenSock, (PSOCKADDR) &Local Addr, sizeof(LocalAddr)) listen (hListenSock, SOMAXCONN).`

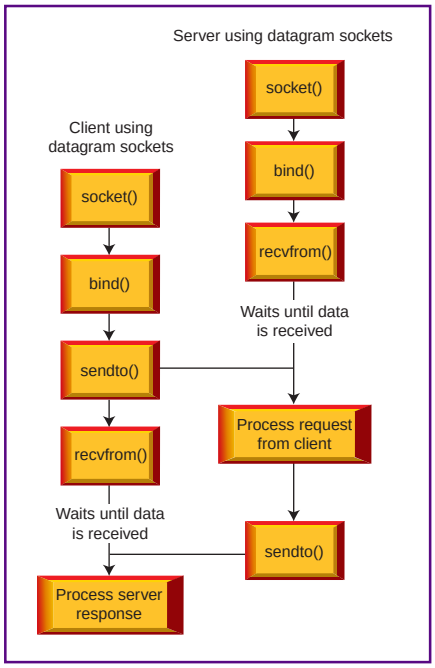


Figure 3—There's quite a bit more going on behind the scenes than this diagram tells us.

`hListenSock` is the handle returned from the server `socket()` call. `LocalAddr` is the address structure for a local well-known port. `SOMAXCONN` is a WinSock-defined constant.

The client, after finding the remote network address parameters, connects with the network API `connect()` function. `connect()` puts the client socket and remote network address together and assigns an arbitrary local port number for this connection. The code is `connect (hSock, (PSOCK ADDR)&Remote Addr, sizeof (RemoteAddr))`.

The first parameter is the socket handle returned from `socket()`. `RemoteAddr` is the structure describing the remote host followed by the size of the structure.

Remember that the server had to bind the port address to the new socket. Well, the client `connect()` performs the same kind of bind. The address information that resulted from `connect()` is used by the recipient to point back to the originator so that a reply may be sent.

Client requests are removed from the listen queue by `hConnectSock = accept (hListenSock, (PSOCKADDR) &RemoteAddr, &RemoteLen);`.

`accept()` creates a new socket with the IP address and port number of the pending connection to keep the original socket free to service other requests. All that's left to do is generate a query on the client side, process the queries on the server side, and reply to the client.

WE'RE NOT DONE YET...

But, I've run out of allotted buffer space. I've laid a solid networking knowledge that I'll build on next time as we install the Realtime Embedded ToolSuite kernel and pass some datagrams.

TCP/IP is a large subject and it can be confounding. But, TCP/IP doesn't have to be complicated to be embedded. APC.EPC

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.

SOURCE
Realtime Embedded ToolSuite
Phar Lap Software
(617) 661-1510
Fax: (617) 876-2972
www.pharlap.com

DEPARTMENTS

MICRO SERIES

Mike Zerkus, John Lusher, & Jonathan Ward

USB Primer

Practical Design Guide

Part 1 of 4

Before getting into the nitty-gritty of working on Universal Serial Bus projects, you need to know the basics. But Mike, John, and Jon offer more than an intro to USB. Their demo gets you ready to work on your own.

Universal serial bus (USB) promises to be the next major advance in PC functionality, completing the PC's transition to a plug-and-play system. But, for all its possibilities, USB is bit of a mystery.

For the average engineer with an idea for a USB product or who has been commanded to convert an existing system, the journey to enlightenment can be an arduous struggle. Rather than merely providing information on USB, we want to show you how to get your USB device up and running.

As a high-speed bus for connecting external devices to the PC, USB is the next step for external peripherals on Windows and Macintosh computers. By allowing hot-plug installation, reconfiguration becomes less of a hassle.

USB enables 127 devices to be on the bus simultaneously. This arrangement solves the problem of limited serial ports.

USB operates at 12 Mbps (there is a low-speed mode of 1.5 Mbps for some devices), and it supports isochronous and asynchronous data transfers. Because USB devices can be bus powered, the transformer ganglion behind the computer can be reduced.

PC users now have a simple user-friendly peripheral bus that supports up to 127 devices and that can be installed without configuring or altering their

current system. Gone are the days of figuring out which interrupt settings and I/O addresses were available and altering the device's settings to fit the available resources.

With USB, you just plug the device into the port. The OS takes care of the rest. There are no jumpers, power packs, powerdowns, resets, or taking the case off. The PC automatically installs the appropriate driver and configures the device as needed.

HOW DOES USB WORK?

RS-232 serial communication with UARTs, transfer rates, stop bits, and so on traces its heritage back to mechanical devices in the days of teletype. In the heyday of TTY, you could repair a UART with a wrench. Adjusting the transfer rate was more like tuning a car than working on electronics.

USB doesn't represent an electronic analog of a mechanical system. In a USB system, the line between hardware and software function is blurred. USB exploits the full potential of a computerized communication system.

The two sides of a USB system are the device and the host. The device side consists of the USB device (e.g., modem or printer), which usually contains a USB microcontroller (e.g., the Intel '930) and the code to properly initiate USB communication to the host. The host side is the PC running an OS that supports USB. The device and host communicate over the USB cable.

USB devices can be self-powered or bus powered, so they can be produced without including a bulky wall-mount transformer. The device gets its power from the host computer or USB hub.

BUS TOPOLOGY

USB uses a tiered/star bus topology in which each device plugs into a hub. The hub is a traffic cop that enforces the low-level rules of the bus. Figure 1 shows the physical arrangement of a USB system. For the most part, hubs are transparent.

Classes are the device categories that share common I/O requirements. In USB there are currently 11 classes: common class, audio, communications, hub, human interface device (HID), image, monitor, physical interface device (PID), power, printer, and storage.

Classes introduce a set of standard drivers native to the OS (Windows 98) and enable you to use them as is, write your own driver, or have a mini-driver.

PACKETS

A packet is a combination of special fields. All packets begin with the Sync field to synchronize the phase-locked loop and have a packet identifier (PID) that helps USB devices determine the kind of packet being sent. The packet is followed by address information, a frame number, or data. There are four types of packets; each has several subtypes.

The first packet type—the token, shown in Figure 2a—is a 24-bit data

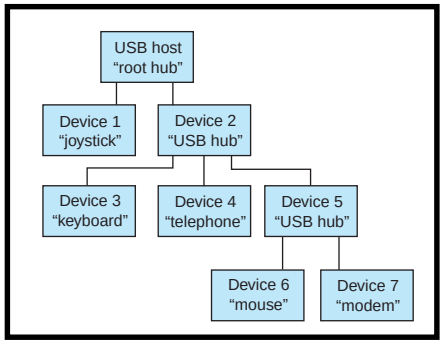


Figure 1—*This diagram shows the physical arrangement of a USB system.*

packet that represents what is happening over the bus. The first eight bits represent the packet identifier. The next seven bits are the address of the device that the host is communicating with. The next four bits are the endpoint address, which is where the data is going in the device. And, the last five bits are the CRC to check the token for errors.

There are four types of tokens—In, Out, Start of Frame (SOF), and Setup. Check the glossary of terms in Design Forum for more details. An SOF packet is illustrated in Figure 2b.

As you see in Figure 2c, data packets contain PIDs for further data error-checking. Data packets alternate between DATA0 and DATA1. The only exception to this format is the Setup packet, which always uses the DATA0 packet.

Data packets have a format of the DATA0/1 PID followed by the data, which ranges in length from 0 to 1023 bytes. The packet is checked with a 16-bit CRC field.

The handshake packet is shown in Figure 2d. These packets inform the sender of the data as to

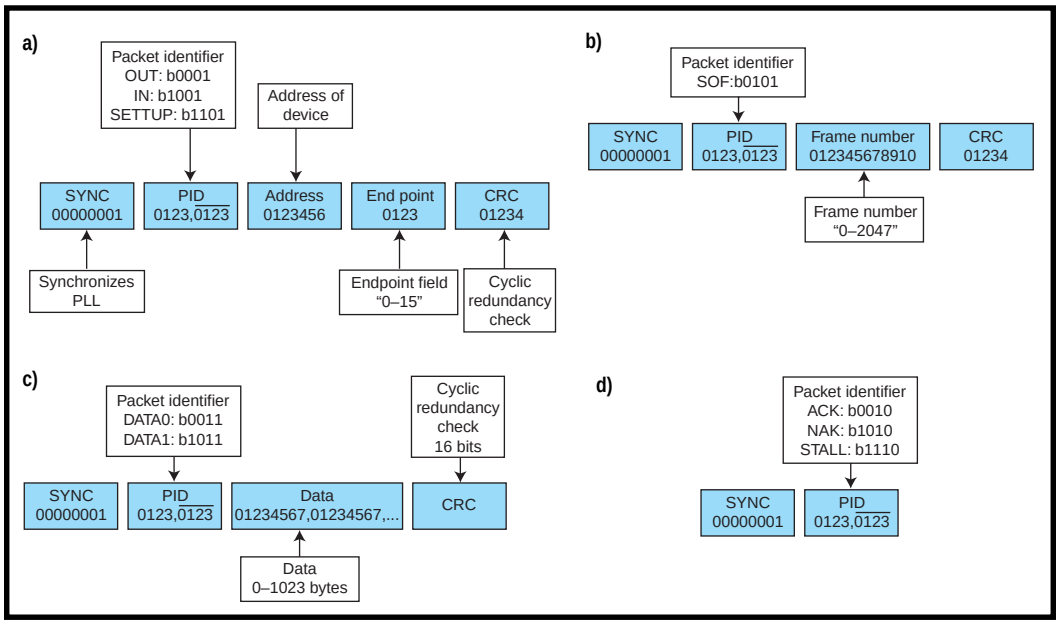
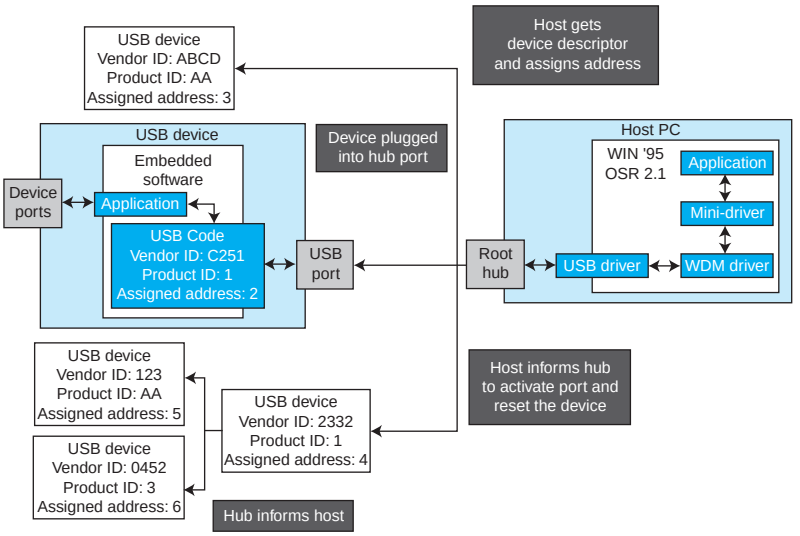


Figure 2—*These diagrams show four different types of packets: token (**a**), SOF (**b**), data (**c**), and handshake (**d**).*



1. Device is attached to a hub port.
2. Hub notifies host of change-of-port status when polled for status updates.
3. Host queries the hub as to the change in port status.
4. Host issues a port enable and reset command for the port.
5. Device now in powered state and replies to the default address.
6. Host retrieves the device descriptor and determines the maximum packet size for endpoint zero. It also retrieves the VID and PID to know what device driver to use with the device.
7. Host assigns unique address to USB device.
8. Host retrieves the configuration descriptors. Based on available power and bandwidth, the host assigns a configuration to the device.
9. The device is now addressed, configured, and ready to use.

Figure 3—*This diagram and the accompanying list illustrate the enumeration of a USB device.*

the condition of the received data packet. Handshake packets are ACK, NAK, and STALL.

The special preamble packet establishes low-speed communication on the bus. This token is sent full speed to the hubs, and the hubs then enable their low-speed outputs.

DESCRIPTORS

The descriptor includes general information about the device. The Vendor ID and Product ID fields play the key role in the enumeration of the device. The descriptor also informs the host about the number of configurations of the device.

Configuration descriptors tell the host the number of interfaces, the device's power requirements, and its attributes. Interface descriptors are the number of endpoints and what class they belong to as well as the interface protocol.

Endpoint descriptors describe the direction and attributes of the endpoints belonging to a specific interface, including the address of endpoint, direction of endpoint, attribute of endpoint, and maximum packet size.

DATA TRANSFERS

A transfer or transaction consists of a number of packets moving back and forth along the bus between the host and a device. There are four types of data transfers in a USB system:

- control—controls the bus, bidirectional, setup, data, status
- bulk—asynchronous data, bidirectional, CRC
- isochronous—time-critical data, no CRC, unidirectional, up to 1023 bytes per frame, guaranteed bandwidth per frame
- interrupt—receives data at timed intervals, input only, 1–255-ms intervals

Enumeration is the bus-configuration process, which takes place anytime the bus is started or a device is plugged into or unplugged from the bus. This process is shown in Figure 3.

The whole USB system is not provided by any one vendor. The OS provides some parts; other parts come from third parties and the developer.

For the next few years, most USB development projects will have to function with both Windows 95 and 98. There are some key items to be aware of when using USB with Windows 95.

Windows 95 doesn't have native USB support. You must have OSR 2.1 build 1214 or better installed on the system.

Windows 95 also has some minor bugs. One such bug is when the USB device has no alternate settings. If this occurs, Windows 95 freezes up when the device is unplugged.

Windows 98 handles USB right out of the box and resolves the above-mentioned bug. It also has a program to assist in developing USB peripherals. `usbview.exe` enables you to monitor the activity on the USB bus as well as get the device descriptors.

END-TO-END EXAMPLE

As promised, here's an example of how to get a USB device working. Using a commercially available evaluation board, the goal of this system is to blink LEDs on the eval board from the PC and to blink indicators on the PC screen from the eval board.

For this project, you need the Anchor Chips EZ-USB evaluation kit V.C or better, the USB specification, Windows PC with USB support, Windows 95 (OSR 2.1 build 1214 or better) or Windows 98, and Visual C++ or Visual Basic (V.5.0 or better). To produce drivers, you need Windows 98 and the Windows 98 DDK.

We hooked up Port A of the Anchor Chips device to a switch/LED circuit and created a DLL using the driver's `IOCTL` functions. A VB program calls the DLL and gets the data from the driver. Basically, VB requests device descriptors by calling the DLL (passes an empty pointer to buffer) and the DLL calls the driver using `IOCTL_Ezusb_GET_DEVICE_DESCRIPTOR`.

Data is passed to a buffer, and the buffer is filled and returned to VB. The driver calls the `USBD.SYS` driver to communicate with the device and OS.

HARDWARE TESTING

When you get your development board, you want to make sure the hardware works. First, install the software, which puts the EZ-USB driver into the Windows system and the install information (`INF`) file into the `INF` directory.

The `INF` file informs Windows as to what driver to load for the particular vendor ID (VID) and product ID (PID) combination. The USB Implementers Forum provides the VID; you assign the PID.

Once the software is installed, plug in the USB device with the included USB cable. It's impossible to hook up the cable backwards because the cable has two different connectors (A and B).

When the device is connected, the red light lights up, signaling that the board has power. Windows informs you that it has found new hardware, finds the appropriate `INF` file, and installs the driver for the new device.

All information concerning driver and VID/PID combinations are in the Windows system registry. If, during

Listing 1—*This Visual Basic code calls a DLL to get the device descriptor.*

```
Private Type UnsignedInt       ‘ Type define to overcome VB's limitation
                               ‘ in not having unsigned 16-bit numbers
  lobyte As Byte
  hibyte As Byte
End Type

Private Type LongData
  Number As Long
End Type

Private Type USB_DD            ‘ Type define for USB Device Descriptor
   Descriptor_Length As Byte
   Descriptor_Type As Byte
   Spec_Release As UnsignedInt
   Device_Class As Byte
   Device_SubClass As Byte
   Device_Protocol As Byte
   Max_Packet_Size As Byte
   Vendor_ID As UnsignedInt
   Product_ID As UnsignedInt
   Device_Release As UnsignedInt
   Manufacturer As Byte
   Product As Byte
   Serial_Number As Byte
   Number_Configurations As Byte
End Type

'DLL functions used to communicate with USB device
Private Declare Function ReadBulkByte Lib "lusher_USB.dll" (ByRef InByte As Byte,
  ByVal PipeNumber As Byte, ByVal DeviceDriver As String) As Integer
Private Declare Function WriteBulkByte Lib "lusher_USB.dll" (ByVal OutByte As
  Byte, ByVal PipeNumber As Byte, ByVal DeviceDriver As String) As IntegerPrivate
Declare Function GetDeviceDescriptor Lib "lusher_USB.dll" (ByRef DevDes As USB_DD,
    ByVal DeviceDriver As String) As Integer
‘ get device descriptor and parse it into appropriate fields
Private Sub Get_USB_Device_Descriptor()
   ‘ Gets device descriptor from the USB device as well as verify
   ‘   that USB is communicating correctly and that correct
   ‘   source code is running
   Dim CheckData As Byte
   Dim ProdID As LongData
   Dim VendID As LongData
   Dim SpecRel As LongData
   Dim DevRel As LongData
   Dim USB_Device_Descriptor As USB_DD
   Dim Result As Integer

   Result = GetDeviceDescriptor(USB_Device_Descriptor, "\\.\ezusb-0")

   ‘ If all transactions met with success, then set up screen and
   ‘   allow user interaction
   ‘ Else alert user to the fact that no USB device is present and
   ‘   do not allow user interaction
   If Result = 0 Then
      Call USBError        ‘ Informs user of error and set form attributes
                           ‘   to offline mode
      Exit Sub
   End If

   Status.Caption = "USB Device Connected"
   Status.ForeColor = &HFF00&
   LSet ProdID = USB_Device_Descriptor.Product_ID
   LSet VendID = USB_Device_Descriptor.Vendor_ID
   LSet SpecRel = USB_Device_Descriptor.Spec_Release
   LSet DevRel = USB_Device_Descriptor.Device_Release
   DesForm.Type.Caption = USB_Device_Descriptor.Descriptor_Type
   DesForm.Spec.Caption = SpecRel.Number
   DesForm.Class.Caption = USB_Device_Descriptor.Device_Class
   DesForm.SubClass.Caption = USB_Device_Descriptor.Device_SubClass
   DesForm.Protocol.Caption = USB_Device_Descriptor.Device_Protocol
   DesForm.PacketSize.Caption =
     USB_Device_Descriptor.Max_PAacket_Size
   DesForm.VendorID.Caption = VendID.Number
   DesForm.ProductID.Caption = ProdID.Number
   DesForm.DevRel.Caption = DevRel.Number
   ProductID.Caption = ProdID.Number
   VendorID.Caption = VendID.Number
End Sub

‘ Example calls to read and write functions
Result = WriteBulkByte(Data_Out, 0, "\\.\ezusb-0")
Result = ReadBulkByte(Data_In, 7, "\\.\ezusb-0")
```

Listing 2—*This example DLL handles calling the EZ-USB driver.*

```
// ReadBulkData(BYTE *OutBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
//   LPSTR DeviceDriver)
// Function reads data from specific pipe over USB port for device in question
int _stdcall ReadBulkData(BYTE *OutBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
  LPSTR DeviceDriver)
{
   HANDLE hUSB_DeviceHandle; // Declare variables
   DWORD  nBytes = 0;
   BOOL bResult;
   BULK_TRANSFER_CONTROL bulkControl;
   BYTE Input[64];
   BYTE index;

   bulkControl.pipeNum = (ULONG)PipeNumber;
   if (NumberOfBytes > 64 || NumberOfBytes < 1)
   // Limit ammount of transfer to 64 bytes maximum
   // If greater than 64 or less than 1 then return an error
   {
      return 0;
   }
   // Get handle to USB device in question
   hUSB_DeviceHandle = CreateFile(DeviceDriver, GENERIC_WRITE,
    FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
   if(hUSB_DeviceHandle == INVALID_HANDLE_VALUE)
   {
   return 0;         // If not a good handle then abort!
}

// Else it is a good handle; read data to USB pipe by calling system driver
  bResult = DeviceIoControl(hUSB_DeviceHandle,IOCTL_EZUSB_BULK_READ, &bulkControl,
    sizeof(BULK_TRANSFER_CONTROL), &Input[0], NumberOfBytes, &nBytes, NULL);
  CloseHandle(hUSB_DeviceHandle);  // Close handle
  // Fill result with that of input array
  for (index = 0; index < NumberOfBytes; index++)
  {
    *OutBuffer = Input[index];
    OutBuffer++;
  }
  return (int)bResult;   // Return our result: success or failure
}

// WriteBulkData(BYTE *InBuffer, BYTE PipeNumber, LPSTR DeviceDriver)
// Function writes data from specific pipe over USB port for device in question

int _stdcall WriteBulkData(BYTE *InBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
  LPSTR DeviceDriver)
{
   HANDLE hUSB_DeviceHandle;    // Declare variables
   DWORD  nBytes = 0;
   BOOL bResult;
   BULK_TRANSFER_CONTROL bulkControl;
   BYTE Output[64];
   BYTE index;

   bulkControl.pipeNum = (ULONG)PipeNumber;
   // Limit ammount of transfer to 64 bytes maximum
   // If greater than 64 or less than 1, return an error
   if (NumberOfBytes > 64 || NumberOfBytes < 1)
   {
     return 0;
   }
   // Fill output array with that of input buffer
   for (index = 0; index < NumberOfBytes; index++)
   {
     Output[index] = *InBuffer;
     InBuffer++;
   }
   // Get handle to USB device in question
   hUSB_DeviceHandle = CreateFile(DeviceDriver, GENERIC_WRITE,
      FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
   if(hUSB_DeviceHandle == INVALID_HANDLE_VALUE)
   {
     return 0;        // If not a good handle, abort!
   }
   // Else it is a good handle; write data from USB pipe by calling system driver
  bResult = DeviceIoControl(hUSB_DeviceHandle,IOCTL_EZUB_BULK_WRITE,&bulkControl,
    sizeof(BULK_TRANSFER_CONTROL), &Output[0], NumberOfBytes, &nBytes, NULL);
  CloseHandle(hUSB_DeviceHandle);  // Close handle
   return (int)bResult;   // Return our result: success or failure
}
```

development, you need to delete these entries, they are located at HKEY_LOCAL_MACHINE\Enum\USB and HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Class\USB.

You can use regedit.exe to edit and browse the registry entries on your computer. You can now unplug and replug the device as much as you need. Until you delete the registry entries, Windows remembers what driver to load and doesn't inform you of any hardware detection again. This step is called enumeration.

Enumeration is when the OS recognizes that there is new hardware on the bus and determines its particular needs. The appropriate driver is then loaded and it gives the device a unique address. Enumeration takes place each time you plug a device on the bus and on bootup of Windows.

At this point you should try the software package that comes with the evaluation kit. EZ-USB Control Panel lets you get the descriptors from the USB device, download firmware to the device, and run the Keil debugger.

THE GOAL

Our goal is to build a demo of a working USB device. The concept is that a user application sends data to a USB device and vice versa. Our USB device is the Anchor Chips development board running firmware we created.

On the host side, we have an application made using Visual Basic. The program communicates with the device via the general-purpose driver (GPD) from Anchor Chips and a DLL we created to implement a bridge between the user interface (VB) and the GPD.

The user interface is an experiment board with four DIP switches and four LEDs. The user selects a four-digit binary combination that appears on the LEDs and vice versa for the DIP switches.

In our VB program, we call functions in a DLL that communicates with the GPD. Listing 1 shows how to call the DLL that communicates with the USB device. It also shows how to parse up the device descriptor and read and write a byte from the USB device.

A DLL was written to communicate between the Visual Basic program and

Listing 3—*Here are a couple of sample routines from PERIPH.C.*

```
void TD_Poll(void)              // Called repeatedly while device is idle
{
  OUTA = byte_in;               // Place byte retrieved from endpoint on port A
  byte_out = 0xF0 & PINSA;      // Read port A and mask to get only upper 4 bits
}
void ISR_Ep1out(void) interrupt 0
{
  if (OUT1BUF[0] == 0x80)       // If READ COMMAND (0x80) then send data to host
  {
    IN1BUF[0] = byte_out;
    IN1BC = 1;  // Inform processor it has a byte to send out to host
  }
  else          // Else set variable to the input byte
  {
    byte_in = OUT1BUF[0];
  }
  OUT1BC = 0;                   // Arm OUT so it can receive next packet
  EZUSB_IRQ_CLEAR();            // Clear the IRQ
  OUT07IRQ = bmEP1;
}
```

the GPD. The DLL gets a device handle to the device driver in question.

We want to communicate to the first instance of the device driver (i.e., the first device to use this driver). If we get a valid handle, we can communicate to it. Otherwise, the device isn't on the bus and the driver isn't loaded. We communicate to the driver via DeviceIOControl. This function passes data to and from the device driver and it returns success or failure.

Listing 2 shows how a DLL can be used to communicate with the GPD.

Listing 4—*This code shows you an example INF file.*

```
;FILE:   EXAMPLE.INF

[Version]
signature="$CHICAGO$"
Class=USB
Provider=%Exanoke%
LayoutFile=LAYOUT.INF

[Manufacturer]
%Example%=Example

[PreCopySection]
HKR,,NoSetupUI,,1

[DestinationDirs]
DefaultDestDir=11

[LusherTech]
;
%USB\VID_06E5&PID_8000.DeviceDesc%=FIRMWARE, USB\VID_06E5&PID_8000
%USB\VID_06E5&PID_8001.DeviceDesc%=USBDEV01, USB\VID_06E5&PID_8001

[ControlFlags]
ExcludeFromSelect=*         //removes all devices from device installer list

[FIRMWARE]
AddReg=FIRMWARE.AddReg

[FIRMWARE.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,firmdown.sys

[USBDEV01]
AddReg=USBDEV01.AddReg

[USBDEV01.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,ezusb.sys

[Strings]
Example="Example USB"
USB\VID_06E5&PID_8000.DeviceDesc="USB Firmware Download"
USB\VID_06E5&PID_8001.DeviceDesc="USB Actual Device"
```

The DLL does the necessary communicating with the system driver and if there is an error, responds to the calling application with an error status. This arrangement provides an easy-to-use interface to the GPD.

FIRMWARE

For your USB microcontroller, we recommend you have the full version of the C compiler because the example files may exceed the evaluation limit of most evaluation-level compilers. Most of the code needed to communicate with the host is already written. Just fill in your peripheral and I/O code.

The development kit has two firmware files called PERIPH.C and FW.C. These files (supplied by Anchor Chips) contain the framework for the whole 8051-based USB control code. The PERIPH.C source file contains the polling loop code segment, as well as the endpoint interrupts for communicating with the host. You merely write your peripheral code in the poll loop.

When data is to be exported, a set of ISRs is called (seven in and seven out). These are the endpoints of the communication pipes. In the initialization section of the code, you need to set the direction of the port pins. For our example, port A is used. The upper nibble is input and the lower nibble is output. Listing 3 shows example routines from PERIPH.C.

In USB the host initiates all communications. If the device has something to tell the host, it must place the data into an output array (IN1BUF[0]).

After the firmware is finished, it must download to the chip because Anchor Chips' USB paradigm calls for the device-side application code to be transferred on startup of the processor. There are two methods for downloading the firmware—B0 load and B2 load. We describe B0 here.

The micro is basically a state machine that does simple USB tasks without 8051 code. Using the B0 protocol, the firmware is sent over the USB to the chip and an external EEPROM contains the device descriptor (VID and PID). This information tells Windows to load a driver.

The driver was made using the ezloader.sys driver source file, which lets you implement your firmware as part of the driver. The device is enumerated to download the firmware to the micro's RAM.

The micro reenumerates and reports a new device descriptor. We used the same VID but different PIDs (x8000 for download, x8001 for device). The new VID/PID combination tells Windows to load the real driver (EZUSB.SYS).

An INF file tells Windows which VID/PID combination goes with each driver. Listing 4 is a typical INF file entry for the VID/PID combo. Our VID is 0x06E5, and the PIDs are 0x8000 and 0x8001. The sample INF file tells Windows which drivers to load according to the VID and PID information that the system retrieves from the device.

READY TO GO

Basically, you treat most of the firmware code as if you were in regular 8051 development, except that the code resides in the POLL loop, not MAIN. There are ample instructions in the kit manuals. The GPD is well documented, and their program handles the rest.

Mike Zerkus has 15 years of experience working on devices and inventions ranging from space devices to consumer products. Mike is the president of CM Research, a development company that specializes in bringing products from concept through prototype to production. You may reach him at mzerkus@cmresearch.com.

John Lusher is an electrical engineer and has been involved with USB development for the last two years. You may reach him at jlusher@lushertech.com.

Jonathan Ward is president of Keil Software and has been involved in the design, implementation, and documentation of embedded systems since the early 1980s. You may reach him via (972) 735-8052.

RESOURCES

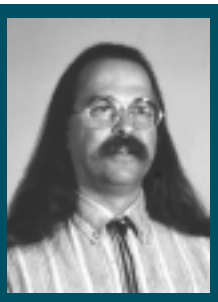
A glossary of USB terms, a checklist for building a USB device, and a list of USB suppliers are available online in Design Forum in May.

FROM THE BENCH

Jeff Bachiochi

Dallas 1-Wire Devices

Part 2: All on One



If you're looking for a needle in a haystack, maybe you could use a Dallas 1-wire device. Well, maybe not, but Jeff shows how its unique addressing systems enable multiple devices to run off one I/O pin.

We didn't have to be in Boston until Saturday evening, but Kristafer (my youngest) and I left home Friday to spend the night in Waltham where my oldest son lives.

That gave the three of us a whole day to bum around Boston before Kris and I were scheduled to meet the rest of the Cub Scouts at the Boston Museum of Science for a special camp-out, er, camp-in.

Getting around Boston isn't too tough. The Mass Transit Authority (MTA) offers multiple subway routes into and out of the city. The four main lines converge at a hub of four midtown stops where you can switch between the red, blue, orange, and green lines.

We spent a lot of time looking at the MTA maps, and Kris had the task of determining our route between destinations. He quickly learned the importance of knowing which direction to head and where to switch lines. All of this got me thinking about networking and the importance of addressing.

Before you could say "Paul Revere," it was time for us to head over to the museum. Just imagine hundreds of Cub Scouts funneling into the museum, all carrying armfuls of sleeping paraphernalia! Not a pretty sight, but well choreographed on the part of the museum. Apparently, these sleep-ins are a common occurrence.

You might think that coordinating so many kids would be a nightmare, but we were quickly split into small groups and whisked away to our own little corner of the facility. Scheduled events kept us busy until midnight. When they say, "lights out at midnight," that's what they mean. Everything goes off; the lights, the exhibits. It's eerie, camping under a black sky with no stars.

As I drifted off to sleep thinking about the number of people staying here, I wondered how they would ever be able to find anyone. Just then a small light appeared and a voice asked, "Is there a Josh Nixon here? I found a bottle of medication with your name on it." Josh piped up that he was indeed here, but what intrigued me was how we were located so easily.

It seems that at registration each visitor was given an identification slot. The schedule for the whole weekend was preprogrammed into a computer to identify where each visitor would be (or at least should be) during that time. With the comforting notion that big brother was watching over us, I fell fast asleep dreaming of buses driving around with large hexadecimal addresses painted on them.

THE BUS STOPS HERE

Park those buses in an infinite loop for now. I want to continue where I left off last month, talking about the 1-wire Dallas devices. In their play for easy expansion, Dallas created a potential monster by enabling multiple 1-wire devices to share the same single I/O bit.

Many of these devices might be the same, like the multiple thermometers that monitor equipment temperatures, or they might be different, like the 1-wire weather station Dallas offers for about $80 on their web site (see Photo 1).

The plastic enclosure contains the electronics and hardware for measuring temperature, wind speed, and wind direction. All of this is done via Dallas 1-wire devices. How is it possible, using a single wire?

There are two levels of commands to talk with 1-wire devices—ROM functions and memory functions. All 1-wire devices are familiar with all ROM functions, which are bus-level functions used to communicate with

all 1-wire devices. (See Part 1 for 1-wire timing and communication specs.)

Following a bus reset, all slave devices patiently listen for a ROM function. After a device is selected via a ROM function, it is ready to respond to a memory function. Each 1-wire device family has its own list of memory functions that support the special qualities of that device. In other words, ROM functions are general commands and memory functions are specific.

The READ-ROM command instructs all devices to forget about the ROM functions and respond with their ID (family code + serial number + CRC). Obviously, multiple devices will answer and the colliding data is worthless, making READ-ROM only good for use with single devices on the bus.

SKIP-ROM instructs all devices to forget about the ROM functions and respond to the following memory function. Again, good for single devices.

MATCH-ROM contains a unique ID (family code + serial number + CRC) of the device you're looking for. All 1-wire devices that don't match the ID you requested are disabled. Only a single device (if it is actually there) pays attention to the following memory function commands. MATCH-ROM is good to use with multiple devices—that is, if you know who's connected.

56-BIT ADDRESSING

If you paid attention last month, you know how each 1-wire device has its own ID number. Each device has its own 1-byte family code (e.g., 10h for all DS1820 thermometers). The following six bytes hold a unique serial number for each manufactured device. No two devices within a family have the same serial number.

A single-byte CRC check byte offers some semblance of correctness. To determine what devices are connected to the 1-wire bus, you must search all 2^{64} combinations to see who's home. But, that task could take years! Luckily, Dallas included a search function to help narrow down the possibilities, but you need to know how to use it.

CLIMBING THE TREE

Think first of a tree on its side with the base of the trunk on the left. A short distance from the trunk, it splits off into two branches; one bends upward, the other downward.

A little further on, those two branches split again, which is the second bit division in our search. And so it goes for 64 (bit) divisions (8 bytes worth).

This scenario gives you an idea of the massive number of possibilities. To keep things straight, at every division, label the branch paths that bend up 1, and the branches that bend down 0. Following the bottom path, all the branches are labeled 0. This is the path we take to identify a 1-wire device with all 0s for its family byte, six ID bytes, and CRC.

To help determine who's there, Dallas takes advantage of the open-collector architecture of the 1-wire bus. When SEARCH-ROM is issued (following a RESET timeslot), you must read a time slot twice. All 1-wire devices that are not disabled (hold that thought) answer first with their present address bit (address first) and then with the complement of that address bit.

Photo 1—*Dallas's 1-wire demo weather station comes with a PC serial interface. Ditching the interface provides a good platform for investigating the 1-wire bus for use with a microcontroller.*

There are four possible outcomes to these two reads: 11, 10, 01, and 00. Two 1s indicate that no devices were present. There must be complementary bits if a single device answers 01 or 10.

When multiple devices are present, they may both have the same or opposite address bits in the present position. If they are the same, you won't be able to differentiate between them at this present bit position. Don't worry about that now. It will all work out later. If they are different, each one forces a 0 on the bus and you know that there is an address conflict at this fork.

From this information, you must choose a path to take. Naturally, if there are no address conflicts, you choose that path on the tree. If there is a conflict, you can choose either path, but remember to come back to this branch later and follow it to find the complete address for the conflicting device.

Choose a path by sending a write-1 or write-0 time slot. All devices that have conflicting bits at this position are disabled until you issue another RESET. These devices are now off the bus for the rest of this path search.

Repeat this combination of double reads with a decision path write until all 64 bits are searched. You now know that there is a device on the bus with the 64-bit address of the path you just completed.

Now, you must reset the bus and do a new search to resolve any conflicts you found along the way. To keep track of where you've been, try this method.

Always take the lower path of any conflict first. Keep track of the last conflict in a path and resolve that conflict on the next path. When you get to the 64th bit of each search, if you've had no conflicts on the present search, you're finished.

The program in Listing 1 uses the assembly routines from last month to search for all the devices on the 1-wire bus. The program follows the flowchart in the Dallas 1-wire databook.

The 1-wire weather station has a few unique points that are worth dis-

Listing 1—*This program, written in PicBASIC, searches the 1-wire bus and reports every device's unique ID number. It uses the assembly routines from Part 1.*

```
// variable definitions
// b0 used in bit tests

symbol  temp=b7                 // data being passed
symbol  templ=b8                // data being passed
symbol  temph=b9                // data being passed
symbol  cntr=b10                // bit counter
symbol  ctemp=b11               // temp counter
symbol  rtemp=b12               // temp zero counter
symbol  byte0=b21
symbol  byte1=b22
symbol  byte2=b23
symbol  byte3=b24
symbol  byte4=b25
symbol  byte5=b26
symbol  byte6=b27
symbol  byte7=b28
symbol  bytenum=b29
symbol  bitnum=b30
symbol  A=b31
symbol  B=b32
symbol  LD=b33
symbol   D=b34
symbol  RV=b35
symbol  RBI=b36
symbol  DM=b37

START:  call SEARCH
  serout 7,n1200,("That is all",13,10)
  pause 10000
  goto START
SEARCH: call FIRST
SEARCH1:if RV=0 then SEARCH3
  temph=byte0/16 : templ=byte0//16 : call PRNT
  temph=byte1/16 : templ=byte1//16 : call PRNT
  temph=byte2/16 : templ=byte2//16 : call PRNT
  temph=byte3/16 : templ=byte3//16 : call PRNT
  temph=byte4/16 : templ=byte4//16 : call PRNT
  temph=byte5/16 : templ=byte5//16 : call PRNT
  temph=byte6/16 : templ=byte6//16 : call PRNT
  temph=byte7/16 : templ=byte7//16 : call PRNT
  serout 7,n1200,(13,10)
SEARCH2:
  if D=1 then SEARCHX
SEARCH3:
  if LD=0 and D=0 then NODEVICE
  call SECOND : goto SEARCH1
SEARCHX:return
NODEVICE:
  serout 7,n1200,("No Device",13,10)
  goto SEARCHX
PRNT:   temp=templ : call LOOK
  templ=temp : temp=temph : call LOOK
  temph=temp : serout 7,n1200,(temph,templ," ") : return
LOOK:
  Lookup
  temp,(48,49,50,51,52,53,54,55,56,57,65,66,67,68,69,70),temp
  return

FIRST:  LD=0 : D=0
SECOND: RV=0
  if D<>1 then RESET
  D=0 : return
RESET:  call RESET_P
```

(continued)

Listing 1—*continued*

```
  if temp=0 then NONE
  RBI=1 : DM=0
  temp=$F0 : cntr=8 : call W_B
QUERY:  cntr=1 : call R_B : b0=temp : A=bit7
  cntr=1 : call R_B : b0=temp : B=bit7
  if A=B then SAME
  call PUT
PICK:   call GET
  bit0=A : temp=b0 : cntr=1 : call W_B
  RBI=RBI+1
  if RBI<=64 then QUERY
  LD=DM
  if LD<>0 then MORE
  D=1
MORE:   RV=1 : return
SAME:
  if A=0 then FIGHT
NONE:
  LD=0 : return
FIGHT:
  if RBI<>LD then LET0
  A=1 : call PUT : goto PICK
LET0:
  if RBI<LD then CHK0
  A=0 : call PUT
MARK:
  DM=RBI : goto PICK
CHK0:
  call GET
  if A=0 then MARK
  goto PICK
PUT:    BITNUM=RBI-1
  BYTENUM=BITNUM/8
  BITNUM=BITNUM//8
  call  B02BYTE
  call  BIT2A
  call  BYTE2B0
  return

B02BYTE: branch
  BYTENUM,(BNUM0,BNUM1,BNUM2,BNUM3,BNUM4,BNUM5,BNUM6,BNUM7)
BNUM0:  b0=BYTE0 : return
BNUM1:  b0=BYTE1 : return
BNUM2:  b0=BYTE2 : return
BNUM3:  b0=BYTE3 : return
BNUM4:  b0=BYTE4 : return
BNUM5:  b0=BYTE5 : return
BNUM6:  b0=BYTE6 : return
BNUM7:  b0=BYTE7 : return
BIT2A: branch BITNUM,(NUM0,NUM1,NUM2,NUM3,NUM4,NUM5,NUM6,NUM7)
NUM0:   bit0=A : return
NUM1:   bit1=A : return
NUM2:   bit2=A : return
NUM3:   bit3=A : return
NUM4:   bit4=A : return
NUM5:   bit5=A : return
NUM6:   bit6=A : return
NUM7:   bit7=A : return
BYTE2B0: branch
  BYTENUM,(PNUM0,PNUM1,PNUM2,PNUM3,PNUM4,PNUM5,PNUM6,PNUM7)
PNUM0:  BYTE0=b0 : return
PNUM1:  BYTE1=b0 : return
PNUM2:  BYTE2=b0 : return
PNUM3:  BYTE3=b0 : return
PNUM4:  BYTE4=b0 : return
PNUM5:  BYTE5=b0 : return
```

(continued)

cussing here. The first is a 1-wire device with controllable I/O bits. The DS2407 has two I/O bits, which can be set or cleared through the 1-wire bus.

In the weather station, this device enables or disables eight 1-wire serial number chips. When it is disabled, all eight devices are disconnected from signal ground and unable to communicate. When it is enabled, each device's data pin can be connected to the 1-wire bus when a magnet closes a reed switch.

A wind vane positions a magnet over one of the reed switches (which are set in a circle around the vane's center of rotation). When the magnet closes the reed switch, the device is connected to the 1-wire bus and its serial number can be read.

Each of the eight devices corresponds to one of the eight compass directions. One of these devices is popping on and off the bus as the wind changes direction. Of course, you must know which serial numbers indicate which direction before the weather station can be used.

With all of this data, you may notice there are two ways to determine wind direction. You can attempt to use MATCH-ROM to determine whether or not each device is attached, or you can use SEARCH-ROM to determine which one is there. Figure 1 shows the output produced by running the search program in Listing 1.

Listing 1—*continued*

```
PNUM6:  BYTE6=b0 : return
PNUM7:  BYTE7=b0 : return

GET:    BITNUM=RBI-1
  BYTENUM=BITNUM/8
  BITNUM=BITNUM//8
  call  B02BYTE
  call  A2BIT
  return
A2BIT: branch
  BITNUM,(GBIT0,GBIT1,GBIT2,GBIT3,GBIT4,GBIT5,GBIT6,GBIT7)
GBIT0:  A=bit0 : return
GBIT1:  A=bit1 : return
GBIT2:  A=bit2 : return
GBIT3:  A=bit3 : return
GBIT4:  A=bit4 : return
GBIT5:  A=bit5 : return
GBIT6:  A=bit6 : return
GBIT7:  A=bit7 : return
```

DON'T TOUCH THAT

Most 1-wire devices come packaged in a variety of ways. Sometimes, a device is permanently connected to the rest of your circuitry, so the plastic-encapsulated through-hole or surface-mount package is the style of choice.

But, some applications require semipermanent or momentary touch-memory packaging. Touch memory for these apps is encased in a metal can about as thick as a nickel. Although difficult to see, an insulator between the lid and the body of the can provides two isolated contacts—ground and data.

Dallas has a variety of receptacles for use with touch memory—for example, keyfobs, plastic cards, and even jewelry (e.g., as the stone of a ring).

Dallas has been pushing touch memory for all sorts of security products. Let's look at how these devices might be used. For example, a touch receptacle at your front door can be locally monitored by a micro.

When your unique ID touches the receptacle, the micro polls the device, recognizes your ID, and commands a DS2407 addressable switch to energize an electromagnetic door strike, which unlocks the door. The micro can directly control the electromagnetic door strike, but when we expand coverage to multiple doors, you'll see how the addressable switch becomes an integral part of the circuit.

Imagine having multiple entrances monitored on the same 1-wire bus. When you touch your ID ring to the front door, how will the 1-wire bus know you're not at the back door?

Because the addressable switch has two switch outputs, one switch can control the door strike and the other can connect the receptacle to the bus. By turning on only one receptacle at a time, you know where the touch ID is coming from.

This setup can be expanded to include many doors, such as an apartment building where the main door must give access to many occupants and each apartment's door must only give access to its owner.

SLAVE AS MASTER

The only time a slave 1-wire device can try to take over the bus is when it needs to indicate an interrupt event. Some 1-wire devices are capable of indicating an alarm condition. The alarm can be a bit in a device's register that indicates a certain condition. The master might poll devices looking for these conditions, or if the device is interrupt enabled, it can issue two types of master-like bus operations.

```
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 E1 A1 60 03 00 00 A9  — N
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 D2 A1 60 03 00 00 1D  — NE
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 EA A1 60 03 00 00 51  — E
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 E7 A1 60 03 00 00 1B  — SE
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 E4 A1 60 03 00 00 42  — S
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 F3 A1 60 03 00 00 9C  — SW
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 DB A1 60 03 00 00 8B  — W
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 DE A1 60 03 00 00 60  — NW
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 E1 A1 60 03 00 00 A9  — N
1D BF D4 00 00 00 00 A1
That is all
10 23 D3 25 00 00 00 F5
12 12 BD 0A 00 00 00 0C
01 D2 A1 60 03 00 00 1D  >— NNE
01 E1 A1 60 03 00 00 A9
1D BF D4 00 00 00 00 A1
That is all
```

Figure 1—*When Listing 1 is compiled and executed on the PicStic (16C84), this output is created by rotating the direction vane to each of the eight cardinal points of the compass. Notice that after completing one revolution, I stopped between N and NE. Here, two magnets were closed, indicating a direction of NNE.*

A type-1 interrupt enables the slave to immediately signal an interrupt on the 1-wire bus as long as the master has left the bus in a reset state (i.e., issued a bus reset with no following command). A type-2 interrupt is withheld until the next bus reset.

Interrupts can hold the bus low for almost 500 ms. It's easy for the master to pick up the interrupt without other 1-wire devices being affected because the master's reset pulse has no maximum time limit for holding the bus low.

Using interrupts doesn't in any way signal which device initiated the interrupt. It remains the master's responsibility to poll devices to determine which device is signaling.

OVERDRIVE

When it comes to speed, the 1-wire bus won't win any races. It wasn't designed for speed. It was designed as a minimal cost interface, drawing minimal current while withstanding large mechanical stresses, with built-in error checking for data transfer reliability.

But, these people couldn't leave well enough alone, so they created devices capable of overdrive. Overdrive is a special timing specification that enables particular devices to operate at 10× the speed of standard devices.

Here's how it works. Remember that once a standard bus reset is given, all devices are waiting to respond. When a device is chosen via MATCH-ROM, all devices without that ID are essentially turned off (until the next bus reset).

Only one device can now reply to further commands, so it doesn't matter what the timing is like, as long as it doesn't create any timing that resembles a bus reset and as long as the master and slave agreed on the timing.

The reduced timing parameters for overdrive let communication increase by a factor of 10 to devices supporting the feature. When the bus reset is sent again (normal timing parameters), the overdrive device reverts to normal timing.

LAST STOP

Well, we've come to the end of this bus line. Although we could transfer to one of many alternative device discussions, there are too many to explore with only one token. I hope I've piqued your interest enough so that you research these 1-wire devices on your own.

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

SOURCE

1-wire devices
Dallas Semiconductor
(972) 371-4448
Fax: (972) 371-3715
www.ibutton.com

SILICON UPDATE

Tom Cantrell

Betting On Webware

Tom's rooting for the future of embedded Internet apps, but he knows it won't be easy. There are some big obstacles to clear before reaching the goal. At least Lawrence Taylor isn't blocking the path!

I have the utmost respect for Steve. Fact is, he and I go back about 20 years and if it weren't for his drive and vision, I'd probably still be peddling chips with nothing more to look forward to than scratching my way up to VP of chip peddling.

Instead, I have a really fun job—poking around Silicon Valley looking for neat stuff to write about. Believe me, it's a lot nicer checking out new gadgets all the time rather than delivering the same old pitch day in and day out.

But, just because I admire Steve, doesn't mean we always see eye to eye when it comes to predicting the future. In fact, I just got off the phone after a bit of a flame session with him: does the embedded Internet hoopla fall more into the hype or hope category?

At last year's Embedded Internet Workshop, I got things rolling with a brief discussion of the major issues and challenges. I tend to think that when it comes to silicon, if it can be done it will be done (and sometimes even should be done). Meanwhile, Steve is a show-me kind of guy.

Anyway, I firmly believe all manner of appliances will be on the web if the silicon wizards have their way. That's not to say it's going to be a trivial process to get from here to there. I see progress as more of a two-yards-off tackle than a long bomb.

But, the long drive starts with the first snap. Let's check our field position today, contemplate the playbook, and see what it will take to score.

PEEWEE PC

Although an Internet refrigerator may seem goofy, I could go to any computer store and put it together today with little muss and fuss. The fact is, the web is still overwhelmingly a home for PCs rather than appliances.

Thus, the brute-force approach to embedded Internet devolves to embedded PCs, a technology for which I have mixed feelings. It's a good news/bad news situation. The good news is that embedded PCs get to leverage the incredible software tools and know-how of desktop PCs. But, the bloatware tendencies of the software is the price to pay.

Just as quickly as hardware prices drop, software bloat makes up the difference. It's difficult to cobble together an entire EPC hardware and software solution for less than $100—where it needs to be in order to boost appliance designs and volume.

Nevertheless, once you've got your pseudo-PC in place, software's a breeze. There is a huge variety of web-enabled RTOSs and such. Datalight offers a minimalist TCP/IP stack with direct support for low-end modems and serial lines. Don't forget that Linux (as Ingo's recent Real-Time PC columns showed) is quite a viable embedded option.

The hardware situation's a bit better with Windows CE, but you're still going to need a rather hefty pile of silicon to get on the air. One unique advantage for CE is that it runs on other than '*x*86 chips, notably the two most popular RISCs—MIPS and ARM.

The latter has been passing out licenses left and right and probably has close to a couple dozen major players signed up. Even Intel offers an ARM option (at 200 MHz, quite a speedy one) with their SA-1100, acquired in a deal to buy DEC's fab.

Annasoft has Windows CE running on SA-1100 and knows all the ins and outs of that port. Meanwhile, Mentor Graphics also fully supports ARM

across the board. With the merger between the original silicon tool side of Mentor and the longtime embedded tool powerhouse Microtec, full support means silicon IP, development tools, and now CE adaption kits and more, all wrapped in a "seamless coverification environment."

How low can EPC go? One of the leanest platforms I've come across is the IPump reference design from Vadem, which you see in Photo 1.

IPump showcases Vadem's latest PC-on-a-chip, the VG330. This puppy is highly integrated with a CPU, real-time clock, 8254-compatible timer, dual 82569A interrupt controller, 16450-compatible UART with HP-infrared support, PCMCIA, and an LCD controller. It also has a built-in no-glue memory connection for SRAM, PSRAM, flash memory, DRAM, and SDRAM.

The '330 (based on the NEC V30) only runs up to 32 MHz, clear recognition that volume cost-sensitive apps are the focus rather than the desktop. Other practical concerns include power consumption, which the VG330 targets with extra power managment (e.g., hibernate, doze, sleep) and clock control logic that cuts standby power to microamps.

Of course, the main criterion for boosting Internet-appliance volume is price. According to Vadem, a high-volume OEM building a minimal variant (e.g., 14.4-kbps modem) of their IPump reference design could probably get unit cost down to $50.

Part of the reason IPump is so inexpensive is because it relies on a DOS- (rather than Windows-) class OS and bypasses the eye-candy side of the web (i.e., http and html) in favor of moving data via simpler e-mail (smtp) and file transfer (ftp) protocols.

CUT THE FAT

There's just so far that you can shrink things if you require your Internet appliance to carry all that protocol baggage (SLIP, PPP, IP, TCP, http, ICMP, ftp, smtp, SNMP, etc.).

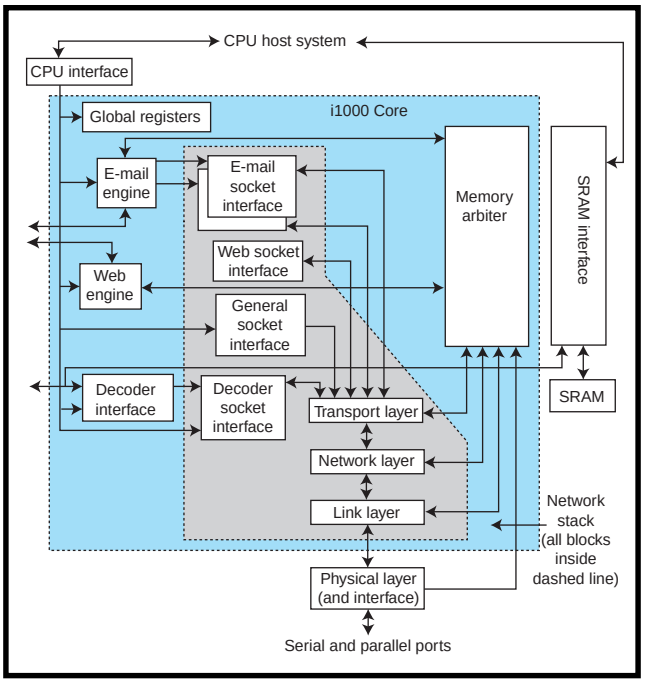


Figure 1—*Delivered as IP and through licensees like Toshiba, the iReady i1000 implements protocol stacks in hardware.*

For goodness sake, all you want is to be able to check your fridge from the office to see if you should stop at the store on the way home. Seems like it shouldn't take more than an 8-bit MCU and a few screens of code.

As you've seen before, that's emWare's claim to fame. It's not a matter of Herculean code optimization. They just shove all the server bloat onto a gateway that handles the I-way on one side and lightweight point-to-point connections to the appliances.

GoAhead Software uses the same strategy. Each appliance requires only an 8-KB MicroAgent that hooks up with the 'Net via the 400-KB gateway server. The server itself, known as Infusion, may be big on bytes but not on bucks. In fact, it's free for the taking (with source code) at www.goahead.com.

Both emWare and GoAhead make it easy to merge existing apps onto the I-way. Rather than a complete rewrite, all that's necessary is to specify which of the apps' existing data items should, in effect, publish and subscribe to the 'Net. Then, the agent code supplied by emWare and GoAhead is linked with your existing application to transparently (i.e., consuming only a small percentage of local processing power) move the data to and from the 'Net via the gateway.

These distributed configurations that shift the protocol processing and high-speed communication burden to a gateway may represent the best hope for truly consumer-class (and price) Internet appliances.

For instance, imagine the Internet refrigerator as just one node in the wired kitchen of the future. It and the other appliances could connect to a local gateway via dedicated links, perhaps power-line modem or short-range RF.

Does your refrigerator really need its own IP address? Probably not. After all, individual appliance data bandwidth requirements are low (you don't need streaming video to watch ice freeze) so these dedicated links can trade off performance in favor of lowest possible cost.

IN THE CHIPS

Another trend in the race to "webify" everything is the emergence of highly integrated chips dedicated to the cause.

Ethernet may not be the right choice for the kitchen, but it's a good match for office and (in the opinion of many) factory web gadgets. As for the latter, I explained in *Circuit Cellar* 92 about how companies like HP opine that the ubiquity of Ethernet will steamroll nitpicking about whether it's theoretically ideal for factory apps. It isn't ideal, but it's here, it's cheap, and it can usually be coerced into getting the job done.

If Ethernet works for you, consider the Net+ARM from NETsilicon. As

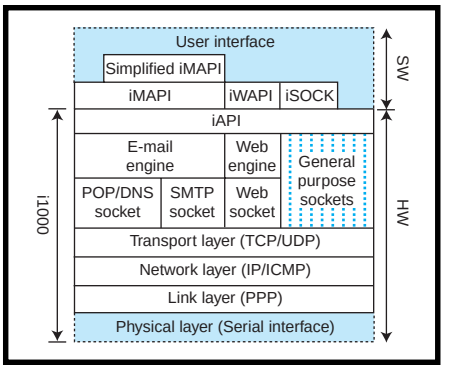


Figure 2—*From the programmer's perspective, the i1000 APIs offer high-level mail, web, and socket services.*

the name implies, it combines a 32-bit ARM7 CPU core with a 10/100 Ethernet MAC (media access controller). Included in the price ($32.50 for 10k units) is a production license for the complete pSOS RTOS package from ISI. This package not only provides real-time hooks for your application but also includes the complete suite of 'Net services including web (http), file (ftp), and mail (smtp).

Besides the CPU and MAC, the NET+ARM handles local processing with two UARTs, an IEEE 1284 (also known as a PC-type or Centronics) parallel port, and 24 parallel I/O lines. Of course, more app-specific I/O can be grafted on to the no-glue memory bus interface that hosts the external ROM and RAM, which contain the RTOS and networking software.

According to NETsilicon, about 80% of the ARM CPU bandwidth is available for local application processing. As well, they're planning a number of variants, including low-cost versions that break the single-digit price barrier.

Listing 1—*As this send–an–e-mail example shows, the i1000 makes it simple for practically any gadget to get on the 'Net.*

```
#include "itypes.h"
#include "imapi.h"
#include "ihlp.h"
#include <stdio.h>

void main(void)
{
  ihlplnit(0);
  imlnit(1000000L);
  printf("Talking to Email revision %x\n".imGetEmailRevision());
  imSetMailServer ("192.168.2.1");
  imInitiateConnection (IMF_SMTP);

  imSMTPSetFrom("\"Test User\"testuser@somewhere.com");
  imSMTPSetTo("\"Another User\"another@somewhere.com");
    imSMTPStartData();
    imSMTPSetData("From: Test User\r\n");
    imSMTPSetData("To: Another User\r\n");
    imSMTPSetData("Subject: Hello there\r\n");
    imSMTPSetData("\r\n");
    imSMTPSetData("How have you been lately?\r\n\r\n");
    imSMTPSetData("Sincerely,\r\n\r\n");
    imSMTPSetData("Test User\r\n");
    imSMTPEndData();
  imSMTPQuit();
}
```

Before you rush off to call, be advised that the NET+ARM isn't targeted to the experimenter or onesy-twosy buyers—unless you're prepared to cough up $20k for a development kit. Of course, the kit includes everything from a development board to a JTAG-based ICE and telephone support and training.

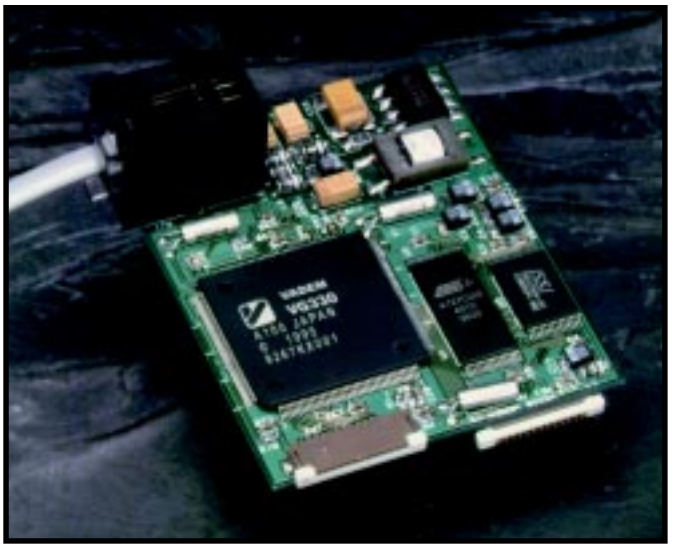


Photo 1—*The Vadem IPump reference design uses their VG330 PC-on-a-chip to enable EPCs to get a piece of the Internet-appliance action.*

ARE YOU READY

The i1000 from iReady is another take on the web-chip angle. But, it's arguably even less accessible to end users than the NET+ARM since the i1000 isn't really a chip per se. Rather, it's a bunch of intellectual property in the form of synthesizable logic written in Verilog. As shown in Figure 1, you can cut and paste the pieces you need to optimize gate count for a particular application.

iReady is investigating deals with licensees that might make standard merchant-market chips available at some point in the future. But for now, only ASIC designers need apply.

The i1000 virtual chip, unlike the NET+ARM, doesn't include a CPU for application processing. It's essentially a dedicated peripheral that can be added to any design, big or small.

The i1000 hooks into a system using various interfaces including a general-purpose bus that connects to your CPU and a dedicated RAM bus for web-related data structures (64–128 KB, depending on the number of protocols and sockets).

A generic physical transport interface lets you have it your way when it comes to exactly what kind of wire (or wireless) medium you prefer. Or, incoming data can be preprocessed and routed onto the chip via a separate decoder bus (e.g., JPEG, MP3).

At 20 MHz, the i1000 can move data at up to 10 Mbps and beyond. But for low-end apps, the clock rate can be cut significantly. The chip need only run at 35 kHz or so to keep up with a 28.8-kbps modem.

It won't be trivial to craft an i1000 ASIC, not to mention designing the rest of your appliance hardware. But, the hardware pain is more than made up with software gain.

That's because the i1000 does a lot more than offer a low-level socket-type interface. It handles much of the protocol-dependent processing that calls for lots of host-CPU headscratching.

For example, the i1000 automatically detects and processes many html tags on its own, reducing the burden on the host CPU. It does so by converting any recognized tags (unrecognized ones are passed up the ladder for your code to deal with) into an iReady command stream format (iCSF).

Instead of your application having to parse html tags like "<TITLE>" and "</TITLE>", the i1000 delivers a single-byte code saying in effect, "Here's a title," along with the text in between the tags. The i1000 does a lot of the grunt work associated with handling text, including attributes and fonts, via the iReady text encoding format (iTEF).

The application software interface in Figure 2 consists of APIs for e-mail, web, and socket services as well as lower-level register access. For example, using the iMAPI mail interface, it's a no-brainer for your app to send e-mail (see Listing 1).

SLIGHT MAKES RIGHT

Between peewee PCs, minimalist micro servers, and dedicated I-way chips, we're making pretty good progress toward a day when "Internet appliance" isn't an oxymoron.

Yeah, Steve, I know the two yards off tackle and a cloud of dust isn't as glorious as the long bomb, but I bet Internet appliances are going to put points on the board sooner than you think.

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

TCP/IP stack
Datalight
(360) 435-8086
Fax: (360) 435-0253
www.datalight.com

IPump
Vadem
(408) 467-2100
Fax: (408) 467-2199
www.vadem.com

Net+ARM
NETsilicon
(781) 647-1234
Fax: (781) 893-1338
www.netarm.com

i1000
iReady
(408) 330-9450
Fax: (408) 330-9451
www.ireadyco.com

EMIT
emWare
(801) 256-3883
Fax: (801) 256-9267
www.emware.com

SA-1100 CE adaptation
Annasoft Systems
(619) 673-0870
Fax: (619) 673-1432
www.annasoft.com

SA-1100
Intel Corp.
(602) 554-8080
Fax: (602) 554-7436
www.intel.com

CPUs
ARM Ltd.
(408) 399-5199
Fax: (408) 399-8854
www.arm.com

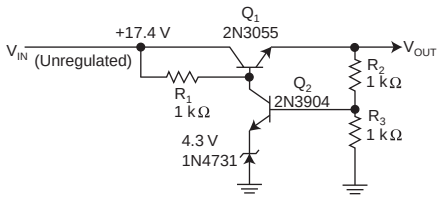
MIPS Technologies, Inc.
(650) 567-5000
Fax: (650) 567-5150
www.mips.com

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

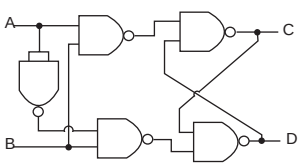
Infusion
GoAhead Software
(425) 453-1900
Fax: (425) 637-1117
www.goahead.com

CIRCUIT CELLAR Test Your EQ

Problem 1—What does this circuit demonstrate and, with the values shown, what is V_{out}?



Problem 2—Describe the logical function and output activity of this circuit.



Problem 3a—In the C programming language, if *x* is a 16-bit integer, what is its range if it is a signed integer? Unsigned integer?

b) Express the following binary numbers in hexidecimal:

10101110 11010010

c) What is two's complement and what is it used for?

Problem 4—The following code is in PIC assembly language. What does TASK do with BYTE?

```
TASK:   movlw 8
        movwf COUNT
HERE:   rlf BYTE
        movlw H'30'
        btfsc STATUS,C
        addlw 1
WAIT:   btfss PIR1,TXIF
        goto WAIT
        movwf TXREG
        decfsz COUNT
        goto HERE
        return
```

Have you forgotten the basics since getting that management job? Did your university ignore real circuits and tell you that HTML was the only path to fortune? Each month, Test Your EQ presents some basic engineering problems for you to test your engineering quotient. What's *your* EQ? The answers are posted at www.circuitcellar.com along with past quizzes and corrections. If you have any good circuits, programs, or engineering brainteasers, fax or e-mail them to us along with the solutions (editor@circuitcellar.com; fax: (860) 871-0411). The best submissions will be published monthly here in Test Your EQ. You receive $50 for each half page of your questions that we publish.

This month's questions were provided by Bob Perrin at Z-World and Circuit Cellar Staff.

PRIORITY INTERRUPT

What's in a Name?

You may not have noticed, but we've been slowly dropping the *INK* from our name. No, we aren't going through another episode of schizophrenic name reference. We just finally have the right to legally be who and what we've always been. I know that sounds confusing. If you don't know the story, there's a little history in the explanation.

Back at the beginning of the personal computer revolution, *BYTE* magazine was the king of periodicals and I was an aspiring engineer at Control Data Corp. As you might expect back then, when you mentioned the word "microprocessor" in a big computer company, the answer was always "micro-what"? My frustration with "big iron" people led me to start writing for *BYTE*.

To make a long story short, after the first few articles, it was obvious that I was becoming a fixture around the place. One day Carl Helmers made a joke that each month I went home and built all these projects in my "circuit cellar." The name stuck and my column became known as Ciarcia's Circuit Cellar.

After 11 years of writing for *BYTE*, I left to start this magazine. But what would I call the new publication—*PC Adventures*? Nope. Obviously, if I was going to take advantage of 11 years of publishing reputation, it had to be called *Circuit Cellar*.

When you start a new periodical, the first priority is to protect the name. When I called my patent attorney and said "*Circuit Cellar*," his answer was that plain language can't be trademarked. I could file the application but he didn't think it would fly. Making it *Circuit Cellar INK* eliminated the confusion and guaranteed acceptance. For 11 years now, that has been our registered trademark.

Of course, registering a trademark and keeping it are two different things. You have to be prepared to defend your exclusive ownership. Over the years, we've contributed heavily to the legal community in a continuing effort.

Last year, I got a new attorney who basically said, "Hogwash." Magazines and newspapers are special cases where simple common-language exclusion isn't cut and dry. Ever hear of *Sports Illustrated*, *Time*, or *US News*? We filed for *Circuit Cellar* and it was granted immediately.

So, what's in a name? I guess it depends on whether we think our readers and advertisers need to be told the subject of the magazine in the title. Surely, there's no question what *Embedded Systems Programming* and *Electronic Design* are about. Something like *Nuts and Volts* might be a little harder to identify. There was a time when I was concerned whether I had to be more explicit about our purpose. For a while, I shrunk the *Circuit Cellar INK* logo and called the magazine *The Computer Applications Journal*. Perhaps you remember. I still have a ton of stationery with that design.

I had this fantasy that if these PR guys didn't have such a tough time knowing what *Circuit Cellar* was, they might be more apt to advertise in a *Computer Applications Journal*. This experiment ratified two things for me—the loyalty of the readership and a recognition of the slow-motion attention in big-company advertising departments. My money would have been better spent on a few schmooze lunches with Silicon Valley PR people instead of new stationery and cover art. The readers didn't care. They called it *Circuit Cellar* before and *Circuit Cellar* after. It was never *The Computer Applications Journal* to them.

We're finally getting it right, and no, this isn't another bout of name schizophrenia. Yes, I've occasionally messed with the name, but our quality and spirit have prevailed through it all. For over 20 years, we've been the *Circuit Cellar*. Now and in the future, we'll continue to be the *Circuit Cellar*.

steve.ciarcia@circuitcellar.com