

www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#108 JULY 1999

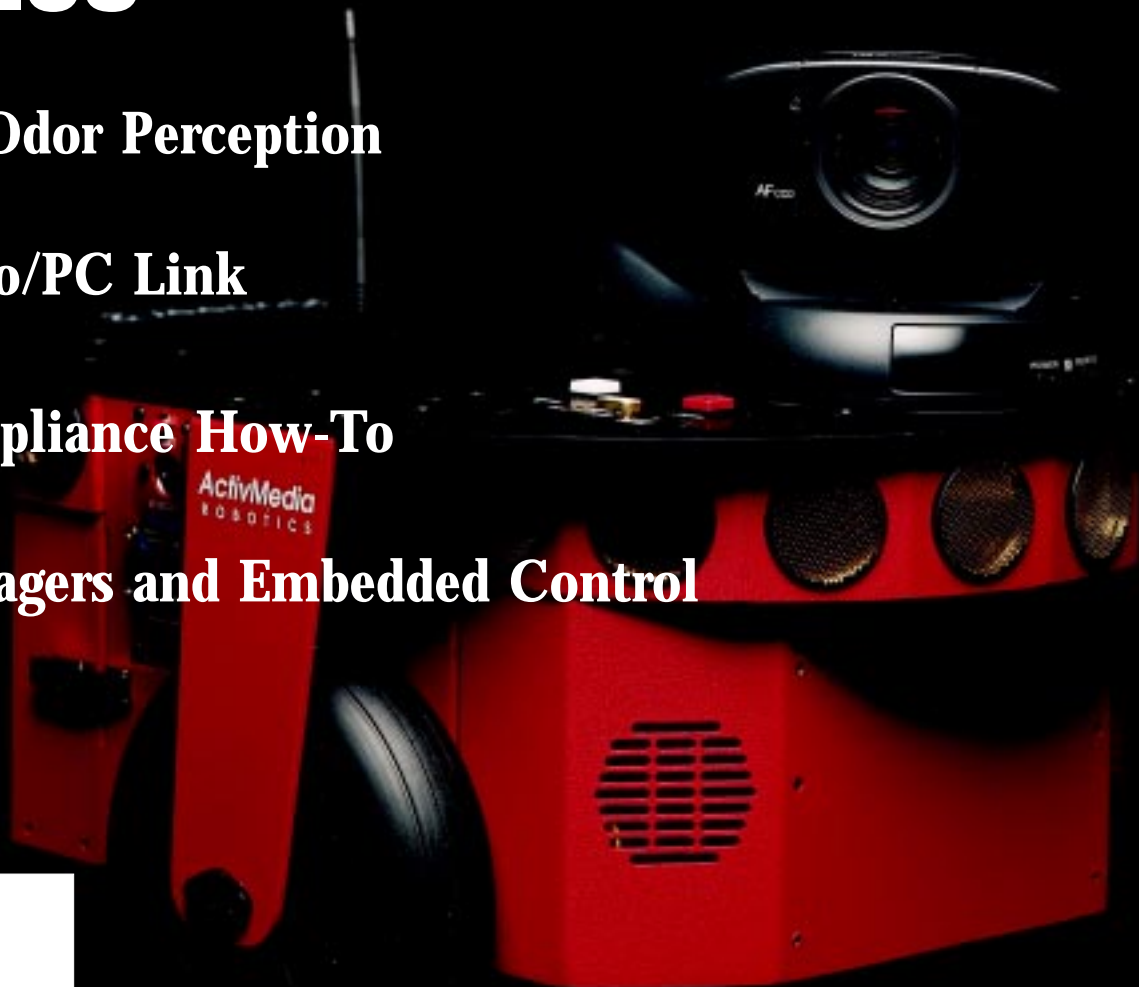
ROBOTICS

Electronic Odor Perception




The Stiquito/PC Link

Internet Appliance How-To

On Call—Pagers and Embedded Control



\$3.95 U.S. (\$4.95 Canada)

- 12 Sniffing Robot**—Robotic Odor Perception
Silvio Tresoldi
- 18 A PC-Based Controller for the Stiquito Robot**
James M. Conrad and Jonathan W. Mills
- 24 Internet Appliance Interface**
Myron Loewen
- 56 Low-Cost Data Acquisition System**
Michael Bading
- 64 Turn the Page**—Pager Technology in Embedded Control
Ingo Cyliax
- 70**  **MicroSeries**
USB Primer
Part 3: Low-Speed USB Host Controller
Glen Reuschling
- 74**  **From the Bench**
Demystifying LCD Muxing
Jeff Bachiochi
- 78**  **Silicon Update**
Eye Candy—Camera Chips
Tom Cantrell

- Task Manager** 6
Elizabeth Laurençot
- Is the Head Screwed on Straight?**
- New Product News** 8
edited by Harv Weiner
- Test Your EQ** 82
- Advertiser's Index** 95
August Preview
- Priority Interrupt** 96
Steve Ciarcia
Circuit Cellar Online

INSIDE ISSUE 108

EMBEDDED PC

- 36 Nouveau PC**
edited by Harv Weiner
- 38 PC/104 Takes a Ride**
Embedding a PC in a Robot
Jeanne Dietsch, William Kennedy, John Belanger, and Kurt Konolige
- 42** RPC **Real-Time PC**
Astronomical Issues—Part 4: Digital Radio Software
Ingo Cyliax
- 50** APC **Applied PCs**
Easy DOS it
Fred Eady

Is the Head Screwed on Straight?



As if you didn't know this already: robots are cool. When I asked a friend why he enjoys robots, his response was "because they're cool!" And when Ken proofed this issue, several times the articles were returned with the comment, "This is cool!" or even "This is very cool!" Indeed. No doubt. But, robots seem kind of like some of the teenagers I've met. You know, definitely "cool" but lacking common sense. Of course, robots are fun. Robots are interesting. Robots are educational. And importantly, robots are useful. Let me mention just a few of the thousands of possible applications.

Robots build cars. They can map the ocean floor and report on whale pod migration. Devices such as the Stiquito, which James Conrad and Jonathan Mills discuss, help us conduct research on sensors, machine vision, and cooperative behavior.

Robots entertain us. They make cute little beep-beep sounds like R2-D2 and they smash cars at monster truck rallies. They play robo-soccer, as Jeanne Dietsch and her colleagues mention in their article on Pioneer robots.

Robots can even save lives. In this issue, Silvio Tresoldi presents an odor-perceiving robot that detects certain gases and follows a path to their source, which allows humans and animals (with their perceptive noses but vulnerable lungs) to stay out of danger.

Clearly, these are all good things, all good reasons to have robots as part of our daily lives. The problem is that robots do exactly what they're created to do. We program them to go forward, they go forward. We tell them to put that box there, and well, they put that box there.

What if there's a cliff? Or a puddle of water on the warehouse floor? Do they walk off the cliff? Drop the box in the puddle? Admittedly, these situations are simplistic, and sure, you can tack on a lot of sensors and program the robot to evaluate the relative safety of every next step. But that seems like overkill for some applications, doesn't it?

Although we humans have plenty of problems making contingency plans for ourselves, when the unexpected happens, we do adjust. We use our common sense and make the best choice we can. But is it remotely possible that we can plan for all the situations our robot might encounter? No way.

It's not an issue of hardware. The circuits are capable. But the software gets in the way. That's because, in order to write the code for the robot to make the best choice given the circumstances, we have to think about how we think—and that's tough. And then there's the difficulty of explaining how we think to someone else. Philosophers and psychologists having been grappling with this for centuries!

I may be too optimistic, but given all the artificial intelligence research happening in the world, perhaps the day will come when we can teach a robot common sense like we (try to) teach our kids. Now, *that* will be cool!

elizabeth.laurencot@circuitcellar.com

Eli

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Sue Skolnick

MANAGING EDITOR

Elizabeth Laurençot

CIRCULATION MANAGER

Rose Mansella

TECHNICAL EDITORS

Michael Palumbo

Rob Walker

CHIEF FINANCIAL OFFICER

Jeannette Ciarcia

ART DIRECTOR

KC Zienka

WEST COAST EDITOR

Tom Cantrell

ENGINEERING STAFF

Jeff Bachiochi

CONTRIBUTING EDITORS

Ingo Cyliax

Ken Davidson

Fred Eady

PRODUCTION STAFF

Phil Champagne

John Gorsky

NEW PRODUCTS EDITOR

Harv Weiner

PROJECT EDITOR

Janice Hughes

EDITORIAL ADVISORY BOARD

Ingo Cyliax

Norman Jackson

David Prutchi

Cover photograph Ron Meadows—Meadows Marketing

PRINTED IN THE UNITED STATES

ADVERTISING

ADVERTISING SALES MANAGER

Bobbi Yush
(860) 872-3064

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

ADVERTISING COORDINATOR

Valerie Luster
(860) 875-2199

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

CONTACTING CIRCUIT CELLAR

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, 4 Park St., Vernon, CT 06066

AUTHOR CONTACT:

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

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

CIRCUIT CELLAR®, 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 Subscriptions, P.O. Box 698, Holmes, PA 19043-9613 or call (800) 269-6301.

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

Circuit Cellar® 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® disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in Circuit Cellar®.

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.

NEW PRODUCT NEWS

Edited by Harv Weiner

PICmicro EDUCATION BOARD

The **Qik Start PICmicro Education Board** is a pre-wired hardware platform based on the Microchip PIC-16C74 microcontroller. Designed specifically to support educators in teaching embedded control design, the board can be used to demonstrate subroutines, port configuration, interrupts, A/D conversion, and other microcontroller functions.

Software routines can be written to test a 4 × 4 matrixed keypad, display information on 2 × 16 alphanumeric LCD, and track A/D inputs from three potentiometers. Other routines include storing and retrieving information in memory, interfacing to the temperature sensor, measurement of rotary encoder signals, and communicating on an RS-232 buffered serial port.

The board comes complete with a UV-erasable PIC16C74 microcontroller (including demo software), 4-MHz crystal, keyboard, a 2 × 16 LCD display, eight LEDs, three potentiometers, and a regu-

lated power supply. A breadboarding area is also provided for building custom circuits (such as a CAN interface or motor controller), and it also incorporates an RJ connector for interfacing to the in-circuit debugger (ICD) available on the new Microchip products.

The product is available in three packages: the standard board (905170); the full-featured board (905171), which includes memory, RS-232, and ICD interface; or the whole-course solution (905175), which includes the standard board, textbook, and lab manual.



Diversified Engineering
(203) 799-7875
Fax: (203) 799-7892
www.diversifiedengineering.net

NEW PRODUCT NEWS

SINGLE-BOARD COMPUTER

The **MMT-51/251** is an SBC designed around the Intel 8051 core. Although designed with the Intel 8xC251 in mind, it supports any of the '51 core processors and many of the Philips and Dallas derivatives.

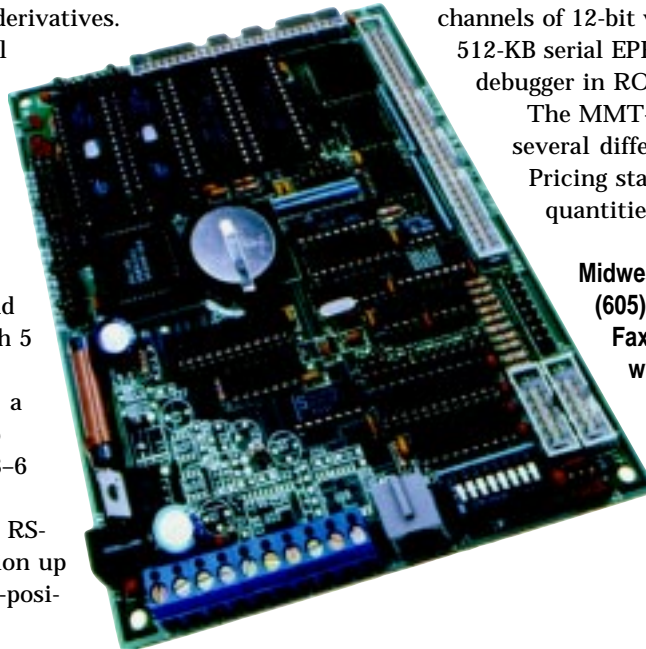
The MMT-51/251 provides all the interfaces needed for a wide variety of OEM control applications. Features include two RS-232 serial ports (third RS-232 port available with the D580C320), 24 lines of parallel I/O, up to 64 × 8 of battery-backed SRAM/flash memory, and up to 64 × 8 ROM (EPROM, flash 5 or 12 V). Also featured are low-power modes, a watchdog timer, a power-fail-detection option, two external interrupt sources, and 3–6 counter/timers.

Options for the board include RS-422/-485 levels for communication up to 5000', a clock/calendar, seven-posi-

tion DIP switch, indicator LEDs, and I²C bus (Access. bus connector). Other options include eight channels of 12-bit ADC (four channels differential), four channels of 12-bit voltage out DAC, 128–512-KB serial EPROM, and a monitor/debugger in ROM with 8-KB SRAM.

The MMT-51/251 can be ordered in several different configurations.

Pricing starts at **\$104/\$199.20** in quantities of 100.



Midwest Micro-Tek
(605) 697-8521
Fax: (605) 692-5112
www.midwestmicro-tek.com

NEW PRODUCT NEWS

GPS DATALOGGING SYSTEM

The **TDS2020GDL GPS Datalogging System** makes it easy to build a custom datalogger to store position and other information on PCMCIA or CompactFlash cards for later retrieval and examination. After the module has run for minutes or months on a small battery, the data-storage cards can be removed for easy data transfer into a PC. Applications range from satellite-fed position information on latitude, longitude, date and time, to pressures, temperatures and rotation rates.

The TDS2020GDL GPS Datalogging System consists of two parts—the TDS2020C logger module and the GPS receiver.

Data is recorded on CompactFlash cards in a Windows-compatible format so a PC can directly read the data into an Excel spreadsheet or Access database.

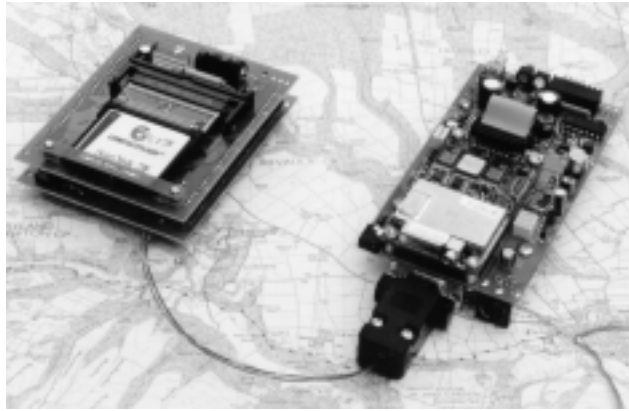
Interactive design enables an engineer with little programming knowledge to tailor the module to exact requirements, and supplied high-level

Forth programs can be user-customized. Data and GPS logging in .CSV format is provided as a ready-made program, but sampling rate and sleep times can be changed.

With a keypad and graphics display, the module is a complete portable instrument or controller.

Two serial ports and an optional CAN-bus interface enhance its communications capability.

The TDS2020GDL GPS Datalogging Starter Pack, which includes the Rockwell Jupiter GPS Receiver, sells for **\$999**.



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

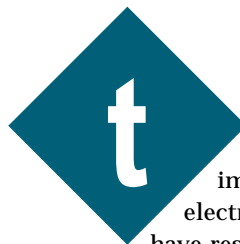
FEATURE ARTICLE

Silvio Tresoldi

Sniffing Robot

Robotic Odor Perception

Silvio's robot can perceive odor concentrations and follow a trail, which could be useful in situations where there are harmful gases or other toxic substances. Watch out, bloodhounds; this robot smells better than you!



The continuing improvements in electronic technology have resulted in machines and robots with incredible characteristics. Every day, new robots with human-like abilities are being designed.

Until now, one of the human senses not yet implemented in a robot was the sense of smell. This article presents a possible application of odor sensing in mobile robots. The robot estimates the maximum gas concentration point and moves toward it or can also follow a smelling path on the ground.

A typical application for this kind of detector would be location of gases or dangerous substances. The smelling path follower can be used as a navigation system by allowing the robot to follow an odor trail on the ground or detect it to divert its trajectory [1].

This application is an example of how a midrange microcontroller (a 40-pin DIP Microchip PIC16C67) can be used in robotics. Particular care was spent on electronic design and power consumption.

The robot includes two gas sensors (to measure gas concentration), two Hall-effect sensors (to control the

motor rotation), and a serial interface. By using the information derived from gas sensors, the robot recursively calculates concentration gradients and decides which direction to go, thereby defining the duty-cycle values of the square waves driving the motors.

Two DC motors driven by PWM signals directly derived from the micro provide movement. A fuzzy-logic control monitors the correct movement.

GAS SENSORS AND CONCENTRATION MEASURE

With today's technology, it's possible to build an electronic nose with two gas sensors. In this application, I used Figaro TGS822 solvent sensors that are tin-dioxide (SnO_2) semiconductor gas sensors [2]. When a gas is present, an oxido-reduction reaction occurs on the sensing element and the surface resistance changes.

The sensor can be seen as a concentration variable resistor described by:

$$R_s = R_0^\alpha$$

where R_s is the sensor resistance, R_0 is the sensor resistance with no gas, and α is the sensibility.

I chose to measure the concentration by means of a technique based on capacitor-charge timing. As shown in Figure 1, two sensors and two resistors are connected to a capacitor. When a pin is configured as an output and is set to a logic high, C begins charging.

All the other pins must be configured as inputs. If the pin is connected to the reference resistor R_C , C charges up to V_{thH} (high-threshold voltage of internal Schmitt trigger) in a time T_C :

$$T_C = C \times R_C \times \ln\left(\frac{V_{ref}}{V_{ref} - V_{th}}\right)$$

If the pin is connected to a sensor (R_S), C takes a time T_S before reaching V_{thH} :

$$T_S = C \times R_S \times \ln\left(\frac{V_{ref}}{V_{ref} - V_{th}}\right)$$

You can use a micro timer to monitor these delays. To guarantee the same initial condition on C, define a sampling period and use R_d to discharge the capacitor (configure the connected pin as an output and reset it to a logic low). By dividing T_C and T_S , you get:

$$\frac{T_C}{T_S} = \frac{R_C}{R_S}$$

By knowing R_C and measuring T_C and T_S , you can determine R_S and then the gas concentration. This technique can use the micro's 16-bit internal timer, which improves the measure resolution.

Even though it's possible to delete undesirable effects of capacitor and resistance tolerances, the obtained resolution is limited to 11 or 12 bits because of the electronic noise and the pins' leakage current. When pins are configured as inputs, they absorb a current that causes the capacitor to charge more slowly.

Let's say i_{leak} is the typical pin leakage current (1 μ A) [3]. Because four pins are configured as inputs and one as output, it's possible to demonstrate that the C voltage raising time is:

$$T_S = C \times R_S \times \ln\left(\frac{V_{ref} - 4i_{leak}R_S}{V_{ref} - V_{thH} - 4i_{leak}R_S}\right)$$

Let's use a particular algorithm to calculate the concentration gradient. In order to work properly, sensors must

be heated. A heater is implemented into the sensor and presents a low value resistance (30–40 Ω) that must be connected to a power supply. The corresponding high current (150 mA) increases power consumption.

Instead of a DC current flow, I used dynamic driving by means of a switching-mode transistor (BS107) driven by a square-wave signal with a duty cycle of 50%. A sensible decrease in power consumption was obtained and robot autonomy increased.

By using the gas-concentration measures, the robot decides to go toward the maximum gas concentration point. Using two gas sensors mounted on the robot at a proper distance, you can obtain information about the spatial distribution of gas.

Note that the direction where the gradient is null describes the maximum increasing direction of function U . Therefore, by sampling the gas concentration U using two sensors, it's possible to determine the gradient:

$$\vec{\text{grad}}U = \vec{u}_x \times \frac{\delta U}{\delta x} + \vec{u}_y \times \frac{\delta U}{\delta y} + \vec{u}_z \times \frac{\delta U}{\delta z}$$

As shown in Figure 2, by estimating the direction where the gradient is null, you can find angle θ , which is the steering angle in the robot motion.

To calculate the gradient, I implemented an algorithm (the Crank-Nicholson differentiator) [4] in which each directional difference is determined as:

$$\frac{\delta U}{\delta x} = \frac{\frac{T_{S22} + T_{S12}}{2T_C} - \frac{T_{S21} + T_{S11}}{2T_C}}{\frac{T_{S22} + T_{S12}}{2T_C} + \frac{T_{S21} + T_{S11}}{2T_C}}$$

$$= \frac{T_{S22} + T_{S12} - T_{S21} - T_{S11}}{T_{S22} + T_{S12} + T_{S21} + T_{S11}}$$

$$\frac{\delta U}{\delta y} = \frac{T_{S22} + T_{S21} - T_{S12} - T_{S11}}{T_{S22} + T_{S21} + T_{S12} + T_{S11}}$$

where time (T_S) is a concentration measure (in fact, it's directly proportional to the sensor's resistance, R_S). Thanks to this equation of the directional difference, it's possible to demonstrate that the effect of leakage currents is deleted.

By having the gradient expression, the robot can determine the angle θ and, knowing the direction, move toward it. This method can be used when the robot is a maximum concentration point detector or a smelling-path follower.

ROBOT CHARACTERISTICS

The robot must have small dimensions to move in restricted spaces, so small motors were a necessity. I used two 3–6-VDC motors that provided low price, good characteristics, and easy driving control.

These motors are sources of strong electromagnetic emissions that could compromise the micro's functionality, so mechanical shielding and two different PCBs were introduced. On the first PCB, there are gas sensors, a micro, a serial interface, and the circuitry for the Hall-effect sensors and driving bumpers (see Figure 1). The motor drivers are on the second PCB.

A PIC16C67 with a 4-MHz quartz oscillator was used and the PCB has a 5-VDC voltage supply obtained by a 9-VDC battery and a DC/DC step-down voltage regulator (a National

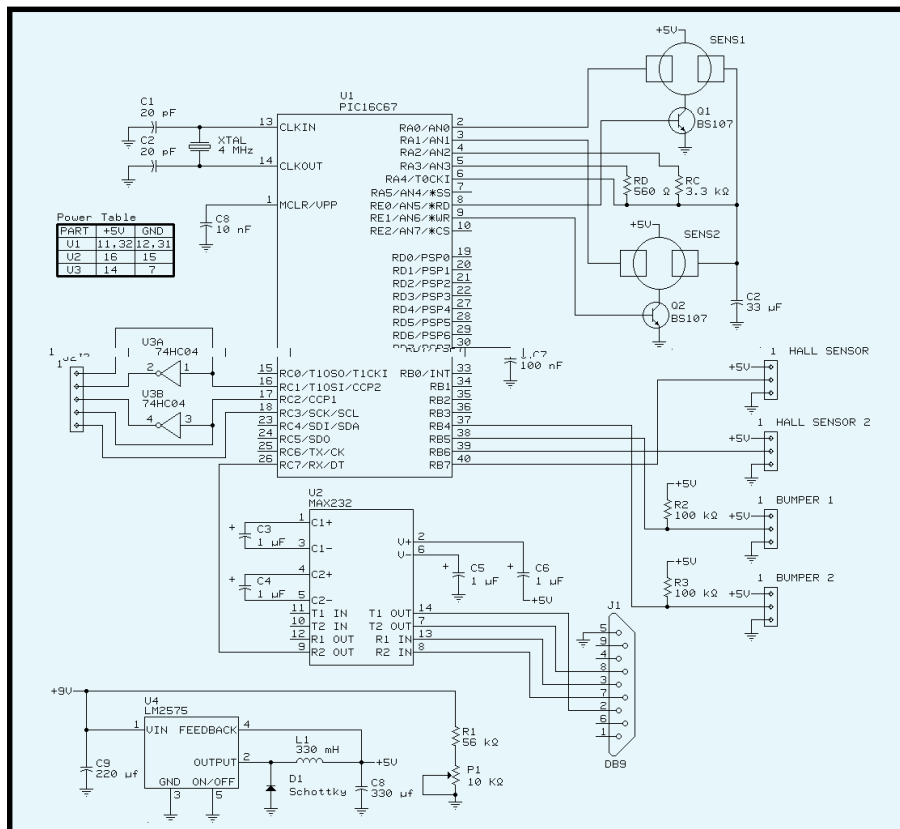


Figure 1—Here are the electronics for gas measurement and fuzzy control. You can see the voltage regulator, two gas sensors, serial interface (with ICL232) and inverter 74HC04 (to have antiphase signals drive DMOS), and the connection to the second PCB and Hall-effect sensors.

Semiconductor LM2575), which can decrease power losses and assure the correct voltage even when the battery voltage drops.

A resistive partitor was introduced to monitor battery charge. A micro pin is connected to the central point and when the voltage reaches the low threshold voltage of the internal Schmitt trigger (V_{thl}), the robot is stopped. This technique ensures that the correct input value is sent to the voltage regulator and then to all the circuitry.

I chose pulse-width modulation to drive the motors. Modulating the duty cycle of square-wave signals enables you to change average motor voltages (V_{Lm}), as shown by:

$$V_{Lm} = V_s(2D - 1)$$

where V_s is the voltage supply and D is driving signal duty cycle.

Driving a motor with different voltages permits different rotation velocities. The robot is steered by rotating the driving wheels. It's possible to vary motor rotation and have the robot steering only work on duty-cycle values.

It's also possible to guarantee negative average voltages (which reverses the motor rotation). During gas-concentration measurement, the robot switches off the motor (working on the ENABLE pin) to reduce noise effects.

Full-bridge DMOS drivers (L6202 SGS-Thomson) are mounted on the second PCB (see Figure 3). Thanks to the switching unipolar devices, power losses are strongly reduced and because of internal implementation, the DMOS driver can connect directly to the micro.

A locked anti-phase control is used, and with a single PWM input signal, it's possible to drive a diametrically

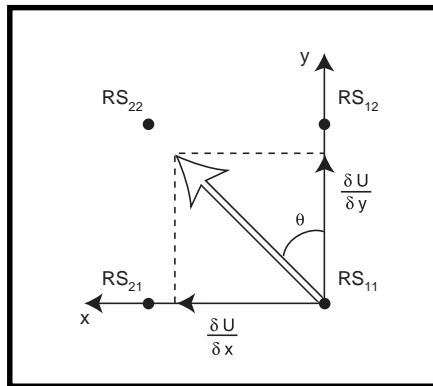


Figure 2—Having a concentration measure enables you to calculate the direction where the gradient is null, thus estimating the steering angle θ of robot motion.

opposite pair of DMOS switches together. The two pairs of switches are driven by the anti-phase signal obtained by the 74HC04 on the first PCB. For avoiding cross-conduction (two DMOS on the same side switching on contemporary and then short-circuiting the power supply), an internal dead-time logic is implemented.

By having two separate voltage supplies for the PCBs, you remove the effect of conducted noise from the motors, which are directly mounted on the robot chassis. The chassis is made of aluminum (to have a light structure) and consists of a plane under which the motors are mounted (see Photo 1a). The metal plane can serve as an electromagnetic shield to reduce the motors' electromagnetic emission.

After estimating the direction to follow (and the angle θ), it's possible to determine the difference of the duty cycle, thus giving the proper signals to the motors. The robot is monitored to ensure it doesn't get diverted because of mechanical or dynamic phenomena. Sensors track the wheel rotation and estimate the robot's direction.

Hall-effect sensors and eight little magnets were put on each wheel. Every time a magnet passes the sensor, it generates a signal and the micro increments a counter (if the signal is derived from the right wheel) or decrements it (if the signal is derived from the left wheel). Optical sensors were not used because any powder or dust along the path influences the sensors' response.

The gas sensors must be positioned in different places when the robot is working as a maximum-concentration-point detector and when it is working as a path follower. As a maximum-concentration-point detector, the sensors must be as far apart as possible to provide more information about the gas distribution. When following a path, the sensors must be as near as possible to the smelling path.

To ensure a correct operation without moving the sensors, I used an aspiration system, which lets me choose where the robot "sniffs" by simply bending two tubes (see Photo 1b). Fuzzy logic is used to link information about the gradient estimation and the direction of travel.

ROBOT LEARNING

Because of differences among motors and sensors, a learning function is used to reach two goals—to determine both the R_0 and duty-cycle values for each gas sensor to guarantee identical motor rotation.

As I mentioned, R_0 s are different from sensor to sensor. It's possible to delete this undesirable tolerance by performing lots of measures without gas and calculating the average value of R_s for each sensor (without gas, $R_0 = R_s$). The measure with gas (R_s) is then compared with this value.

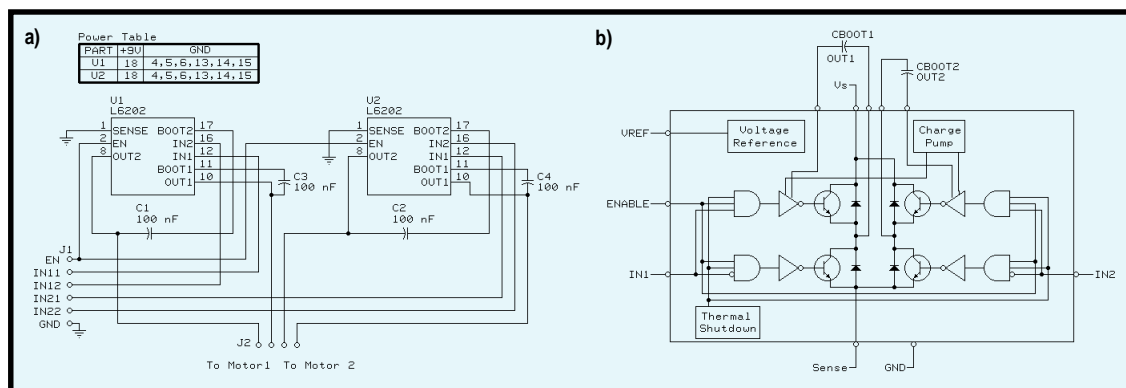


Figure 3—These schematics show the PCB (a) and internal electronics (b) of the SGS-Thomson L6202 DMOS driver. It is possible to bootstrap the capacitor, which is directly connected to the outputs and the internal charge pump, voltage reference, and thermal shutdown.

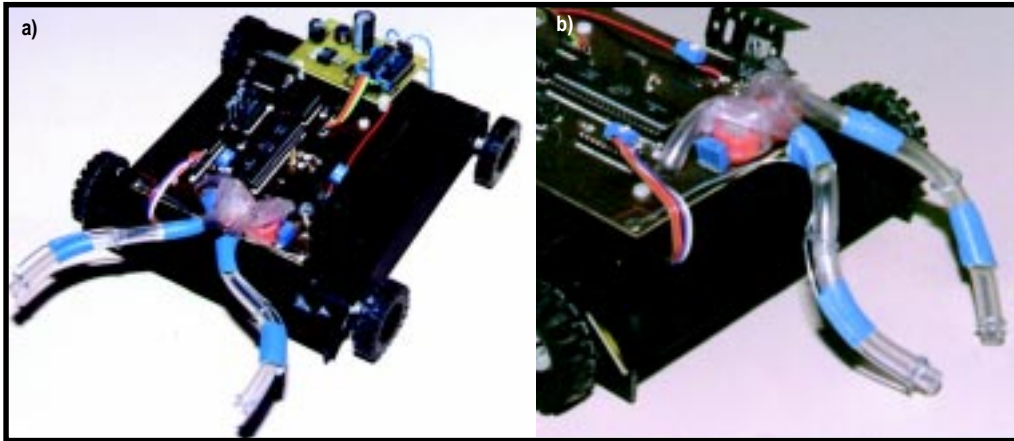


Photo 1a—Note the two PCBs on the sniffing robot. While the robot is working as a maximum gas point detector, the aspirator tubes are far from each other. **b**—When the robot works as path follower, all you have to do is bend the tubes down toward the ground and the sensors can follow the odor trail.

When the same voltage is supplied to the two DC motors, because of internal electrical and mechanical differences, different rotations are noted. To delete this effect and guarantee identical rotation, the duty-cycle value must be measured. This measurement is obtained via signals from Hall-effect sensors and a continuing variation of duty-cycle values.

At the beginning of the process, a threshold value is loaded into the micro's counter and the same duty-cycle values are supplied to the motor drivers. Rotations are measured as different signals from the Hall-effect sensors and thus different values of the micro's counter (remember, its value is increased or decreased each time a signal from the left or right wheel is sensed). This value is used to estimate the difference of the duty cycle to apply to motor drivers.

When no difference between the counter and threshold values is detected, the duty-cycle values are loaded into memory cells and added to the gradient information. When the angle θ is null, you have a linear trajectory even though there are strong differences in the motors' characteristics. This procedure annuls the intrinsic motors' diversity and guarantees that steering is a result of gradient estimation, not dynamic or mechanical problems.

FUZZY-LOGIC CONTROL

After you estimate the direction the robot needs to go, you have to combine gradient information and Hall-effect signals to control the correct movement. Fuzzy-logic control helps solve this problem. I used *fuzzyTECH-MP* devel-

opment tools (see Photo 2) to combine fuzzy variables gradient and `diff_motor` to have the output fuzzy variable `diff_duty`.

The variable gradient comes from gradient estimation and presents three MBFs—left is activated if the gradient estimation gives a left direction to follow, null if the direction is straight, and right if the gradient is to the right.

`diff_motor` derives from the Hall-effect signal (and represents differences of rotation). It is also described by three

MBFs. Each MBF is activated if the robot is going toward the left (`diff_left`), the right (`diff_right`), or straight (`diff_null`). The output variable `diff_duty` represents the difference of the duty cycle to apply.

Typical fuzzy rule structure makes control easy. By introducing three MBFs for each fuzzy variable (as shown in the lower side of Photo 2), it's possible to have nine activation rules (as shown in the upper right side of Photo 2). A typical activation rule is made of:

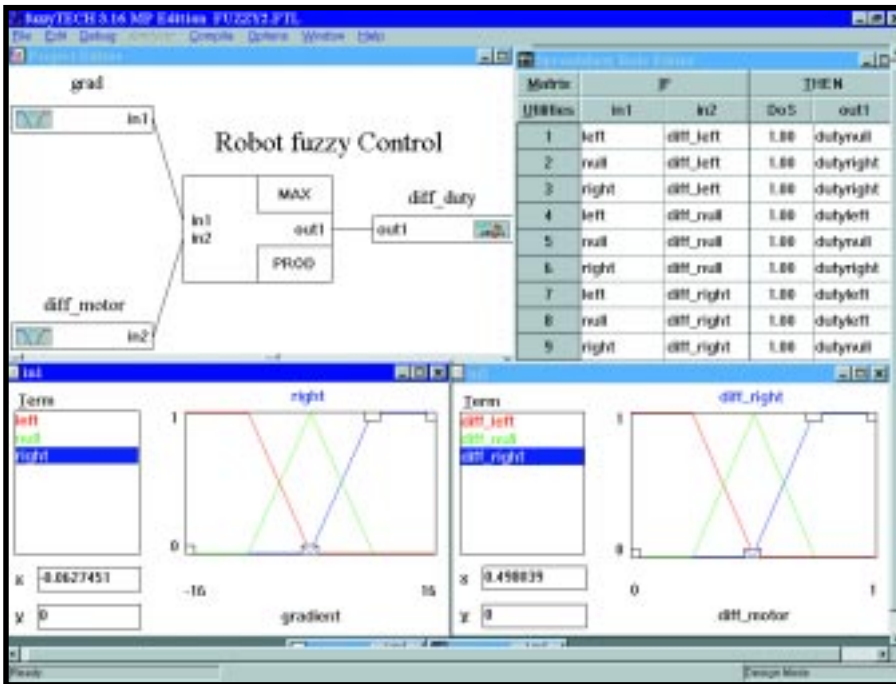


Photo 2—This is what the fuzzy-logic controller, spreadsheet rule editor, and MBF of the input fuzzy variables gradient and diff_motor look like.

IF cond1 AND cond2 THEN cond3

where cond1 is a condition of the variable gradient, AND is a Zadeh logic operator, cond2 is a condition of diff_motor, and cond3 is the activation diff_duty.

It's possible to use human thought to write activation rules. The gradient information is compared with the diff_motor information. If they match, the regulation action is light. In fact, this situation means a correct robot direction, and dutynull is activated to ensure a light regulation.

If the gradient and diff_motor information contrast, regulation is strong. In this case, the robot isn't going in the correct direction and has to divert, so the fuzzy variables dutyleft or dutyright are activated.

Defuzzification is obtained by the CoM (Center of Maximum or Sugeno) algorithm to ensure good resolution and easy calculation. Thanks to the RS-232 serial interface, it's possible to adjust the fuzzy parameters as MBF or index of activation of each rule.

RESULTS

This project required many tests. Initially, the robot had to learn in a gas-free environment. At first, the robot moved chaotically, but because of the

dynamic regulation of the duty cycle, it soon began to follow a linear trajectory.

To test the robot as a maximum-gas-concentration-point identifier, a plate with alcohol and naphthalene was set up to simulate a gas source. Given the solvent sensors, the electronic-nose response was good even though we didn't use gas. At 4–5 m from the source, the robot came close to the plate (less than 15 cm away) and even touched it several times.

To test all possible behaviors, different starting positions were tried. The main setback was airflows, which tend to spoil the spatial distribution of gas and divert the robot from the gas source.

To test the robot as a path follower, we drew a path using the same alcohol and naphthalene mixture but more concentrated. The odor trails had different shapes to test the robot's ability to follow paths.

If the trail contained an angle less than 120°, the robot had problems following the path. A better aspiration system and mechanical structure may solve this problem.

In other tests, we found that by using a switching voltage regulator and unipolar motor driver, it's possible to decrease power losses and increase robot energy autonomy.

Using a micro, we integrated all the work into one device. Good software development decreases the number of components on the PCB, lowering cost.

An electronic nose may seem strange, but an odor-sensing robot can be useful when the presence of gas or toxic substances is dangerous to humans. ☒

Silvio Tresoldi works as an electronics designer in microcontroller applications. You may reach him at set_electronics@usa.net.

REFERENCES

- [1] R.A. Russell, "Laying and sensing odor marking as a strategy for assisting mobile robot navigation tasks," *IEEE Robotics and Automation*, Sept. 1995.
- [2] Figaro Engineering, *TGS 822 for solvent vapor detection*, Technical Information, 1996.
- [3] Microchip Technology, *Microchip PIC16/17 Handbook*, 1997.
- [4] M. Proakis, *Digital Signal Processing: Principles, Algorithms and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

SOURCES

PIC16C67

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

LM2575

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

L6202

SGS-Thomson
(617) 259-0300
Fax: (617) 259-4421
www.scp.sidex.ro/stonline/index.htm

TGS822

Figaro USA, Inc.
(847) 832-1701
Fax: (847) 832-1705
www.figarosensor.com

fuzzyTECH

Inform Software
(630) 268-7550
Fax: (630) 268-7554
www.inform-ac.com

FEATURE ARTICLE

**James M. Conrad &
Jonathan W. Mills**

A PC-Based Controller for the Stiquito Robot

Small. Inexpensive.
Easy to develop.
The Stiquito meets
all of these require-
ments. If you've
never worked with
one before, listen up
as James and
Jonathan explain
how they made this
little robot walk with
a tripod gait. Simple.



The typical legged robot is large, complex, and expensive. Naturally, such factors have limited the use of legged robots in research and education.

Few universities can afford to construct robot centipedes or 100 six-legged robots to study emergent cooperative behavior. Even fewer universities can give each student in a robotics class their own walking robot.

The introduction of the Stiquito, which is shown in Photo 1, changed all of that. The Stiquito was developed from a larger and more complex robot called Sticky (because it looked like an insect commonly called a "walking stick").

Photo 1—The stiquito is an inexpensive hexapod robot that uses nitinol wire for propulsion. When nitinol is heated by running current through it, the wire contracts, moving the legs back, and the robot forward. Watch out! Nitinol has a tendency to eat batteries in no time, so an external power supply is suggested.

The Stiquito is a small, simple, and inexpensive six-legged robot that has been used as a research platform to study computational sensors, subsumption architectures, neural gait controllers, emergent behavior, cooperative behavior, and machine vision. It has also been used to teach science in primary, secondary, and high school curricula.

Jonathan Mills announced the availability of the Stiquito in 1992. For \$10, you could order a kit from Indiana University to build the small robot. Jonathan didn't envision the number of requests he would receive, which by 1996 had reached more than 3000. The volume of orders strained his personal ability to fulfill them and he soon stopped supplying the kits.

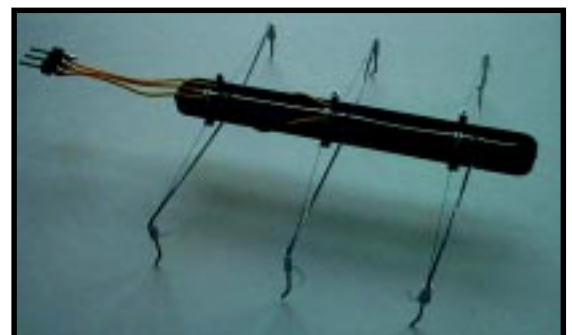
At the same time, we were finishing the book *Stiquito: Advanced Experiments with a Simple and Inexpensive Robot*. The book contains instructions on building the Stiquito and its control circuits as well as a robot kit.

One of the most flexible ways to control the walking gait of a Stiquito robot is by using a PC and writing a program. The program controls the contractions of the nitinol wire, thus making the Stiquito walk.

This article contains the instructions for making a circuit that can be plugged into the parallel port of a PC. We also discuss the concepts of the PC parallel port and provide instructions on how to write a program to make the Stiquito walk with a simple tripod gate.

THE BIG PICTURE

The PCB plugs into the PC's parallel port and generates enough current to light up LEDs on the board and make the Stiquito walk. The LEDs help you develop your computer program, and



the board provides an easy-to-see report on how your program is executing. Once your program correctly works and the LEDs show a viable gait, you can plug the Stiquito control wires into a socket on the board.

The black lines in Figure 1 show the logic on the circuit board when the Stiquito isn't attached. The ULN2803 driver chip inverts the value of the input so the LED will light up when current is drawn towards the ULN2803.

The addition to Figure 1, shown in blue, illustrates the attachment of the Stiquito robot. In this circuit, the LEDs light and the nitinol legs contract.

The circuit in Figure 1 should be used without the Stiquito attached to test your hardware and software. This precaution protects the Stiquito's nitinol actuators from damage while you are developing your circuit.

The parts needed to build this simple board are readily available from electronics suppliers and cost about \$5. In addition to the circuit board, we made a tether to connect the board to the Stiquito robot.

MAKING THE PCB

Although there are many circuits you can build to attach to the parallel port, we recommend using a circuit that doesn't draw current from the PC. We recommend a dedicated power supply, a 5-6-VDC transformer, or a 9-V battery.

To make the parallel port board, simply insert the sockets, integrated LEDs, and connector into the board and start soldering. After that, insert the ULN2803 chip into the socket and solder your power source to the two-pin jumper post. Make sure you insert the LEDs into the board in the correct orientation. We used integrated LEDs, which have a diode and resistor combined in one package.

The component and solder sides of the PCB are shown in Photo 2. Although we made a custom PCB, we have also used a Radio Shack perf board 276-150, which is particularly handy because it has board holes electrically connected like a breadboard.

Your PC parallel PCB is now complete. Using your ohmmeter, put one

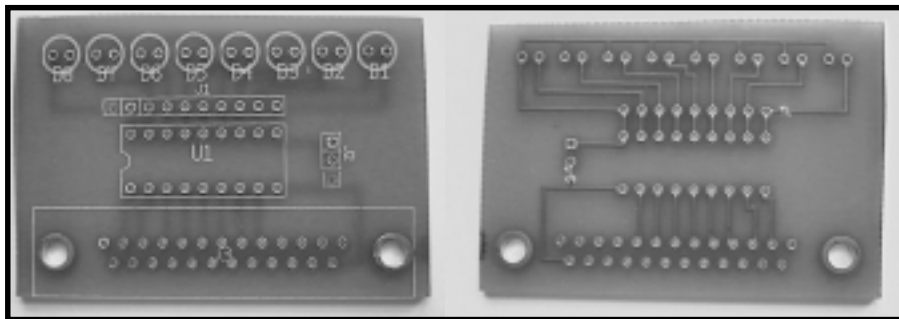


Photo 2—This board was custom made for this circuit, but you can use perf board material as well. We also recommend socketing the ULN2803 instead of soldering the chip to the board in case the chip fails.

contact on the pin labeled 1 or 9 of the header and the other contact on each of the other eight pins, one at a time. Make sure the ohmmeter registers some resistance, but not infinite resistance. Check your work to make sure you have no shorts or broken traces.

GETTING ATTACHED

Now that you've built the parallel port controller board, you need to prepare the Stiquito robot and make its control tether.

Cut a small length of wire-wrap wire and solder it to the center Stiquito bus bar and then to the center pin of the three-pin jumper. Solder the other two Stiquito tripod control wires to the two outside pins of a three-pin jumper.

Sand all six ends of the three wires of the magnet wire group. Next, you need to solder the three wires at one end of the magnet wire to the three pins of a three-pin socket. Identify the wire soldered to the center of the three-pin socket and solder it to the first pin of a nine-pin jumper post.

Solder one of the remaining wires of the tether to the next four pins of the nine-pin socket and use some of the wire-wrap wire to connect these four pins together (repeat this step with the remaining wire of the tether).

Plug the tether into Stiquito, but don't plug it into the parallel port card yet. Use the board to test your Stiquito walking

program by observing the LEDs (perhaps preventing damage to the nitinol wires because of a programming error).

THE PARALLEL PORT

The parallel port was designed to serve as an output port from a PC and attach to a printer. Some parallel ports allow both input and output, but we only used the port as an output.

Although there are 25 pins for a parallel port, we only use nine. Eight lines are used as data output lines and one line serves as the electrical ground.

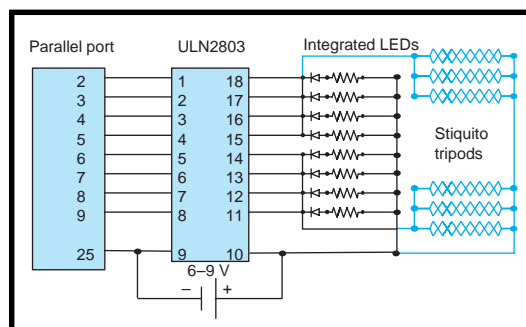
When using the parallel port, computer programmers usually write information to two locations. One register location controls the port, and the other contains the data to send. We used only the data register.

Some computers have more than one parallel port, generally labeled LPT1, LPT2, and LPT3. Each has a different data register. You can access them by using a different register address.

Typically, the register address for the single parallel port (or LPT1) is &H378, but your PC may use another address like &H278 or &H3BC. You can verify this by using the Microsoft diagnostics program (MSD.EXE) and examining the port address.

We use all eight of the parallel port output lines to control our Stiquito

Figure 1—Only nine of the parallel port pins are used. The ULN2803 Darlington transistor array is a common chip used as a current driver. The integrated LEDs are used to help program Stiquito's walking gaits. Four transistors drive three Stiquito legs.



robot, and we control each line with a binary digit, or bit. A response of 1 means turn on the line, and a response of 0 means turn off the line.

To write to the parallel port, write eight bits of data to the parallel port's data register. For example, to write the signal 1 to the top four bits and 0 to the lower four bits of the register, send the eight bits 11110000 to the port. In QBASIC, this is written as `OUT &H378, data`, where *data* is the bit pattern 11110000.

Unfortunately, we can't represent the bit pattern 11110000 as data in the QBASIC language. But, we can convert it to hexadecimal representation.

For our Stiquito control application, we only used nibble values of 0000 (0) and 1111 (F). To define the value in a hexadecimal number, we put &H in front of the digits. The line is now written `OUT &H378,&HF0`.

GAIT PROGRAMMING

The mechanisms of arthropod locomotion are complex and have been extensively studied. The structure of

Listing 1—*This program makes Stiquito walk with a tripod gait. This code assumes that the upper nibble controls one tripod, and the lower nibble controls the other. We allow the nitinol to rest after it is activated.*

```
REM "OUT &H378" sends an 8-bit value to the printer port. The
REM data sent is hexadecimal.
```

```
    DELAY = 14000
10 OUT &H378, &HF0 : REM &HF0 is binary 11110000
    FOR x = 1 TO DELAY : NEXT x
```

```
    OUT &H378, 0
    FOR x = 1 TO DELAY : NEXT x
```

```
    OUT &H378, &HF : REM &HF0 is binary 11110000
    FOR x = 1 TO DELAY : NEXT x
```

```
    OUT &H378, 0
    FOR x = 1 TO DELAY : NEXT x
```

```
REM If a key on the keyboard was pressed, then end.
REM Otherwise, walk some more!
a$ = INKEY$
IF a$ = "" THEN GOTO 10
END
```

an insect leg is also quite complicated. But even though the Stiquito is simple, small programs can demonstrate the fundamental features of arthropod locomotion.

Later on, you can develop more realistic models of gait controllers based on neural networks or central pattern generators and feedback from strain gauges or other sensors that

Listing 2—This code piece shows how to keep nitinol contracted with a 33% duty cycle.

```
REM High-frequency pulses initially contract actuators
FOR a = 1 TO 20
  OUT &H378, &HF0          : REM &HF0 is binary 11110000
  FOR x = 1 TO 100 : NEXT x
  OUT &H378, 0
  FOR x = 1 TO 100 : NEXT x
NEXT a

REM Low frequency pulses maintain actuator contraction
FOR a = 1 TO 40
  OUT &H378, &HF0          : REM &HF0 is binary 11110000
  FOR x = 1 TO 100 : NEXT x
  OUT &H378, 0
  FOR x = 1 TO 200 : NEXT x
NEXT a
```

mimic the sensorimotor loop in a real insect.

The gaits of insects are believed to be a result of central pattern generators that vary the animal's gait from a metachronal wave to a tripod gait and all the variations in between. Each gait conserves energy as it preserves the balance of the insect. As the sequences in Figure 2 indicate, the insect

is always in a stable position with at least three legs (and often more) on the ground at all times.

The metachronal wave is the slowest and most stable gait. It's seen when a "wave" of leg movement ripples down each side of the insect or arthropod. The animation sequence in Figure 2a shows two waves flowing down each side of a ten-legged insect robot.

The tripod gait is the fastest stable gait, with two legs on one side of the insect and one on the other side alternately on the ground or in the air, as shown in an animation of an advanced six-legged insect robot (see Figure 2b).

This tripod gait relies on a leg that has two degrees of freedom. The Stiquito assembled using our book has only one degree of freedom.

Our Stiquito walks with a simpler form of the tripod gait shown in Figure 2b. The legs only flex and relax while they are on the ground. This is the same way it is controlled using the manual controller explained in our book.

Using QBASIC to control the walk, you'll need to use the `OUT` statement to activate and deactivate the legs. You should add a delay in your program to hold the activation signal for about 1 s, then hold the deactivation signal for 1 s. The code to perform this task for one tripod is shown in Listing 1.

The number 14000 is an arbitrary value that is computer dependent. You may have to make this number

higher if your computer is faster than a '486-based machine (66 MHz).

SAVING POWER

Driving the nitinol actuator with the same amount of current is unnecessary after the nitinol contracts. Only enough current to keep the nitinol contracted (i.e., just enough to replace the energy that escapes as heat) is needed. The current and the voltage supplied to the nitinol cannot be changed dynamically, but the power can be varied using a technique called pulse frequency modulation (PFM).

PFM means that the number (frequency) of pulses is varied over time. The PC parallel printer port and the interface card can generate a PFM signal because the nitinol reacts slowly compared to the speed with which a BASIC program can turn the ULN2803 driver chip on and off.

By varying the length of time that the driver chips are left off, the frequency of the pulses can be increased or decreased. This arrangement allows the power used to drive the robot to be varied dynamically.

The nitinol actuator behaves as a leaky integrator of the current pulses sent to it and responds to the heat generated by the current pulses and lost to convection from the wire. Figure 3 shows how a PFM driver program works, and Listing 2 shows how to use PFM to control one tripod of the Stiquito robot.

JUST THE BEGINNING

The purpose of the Stiquito robot kit is to enable you to create a platform

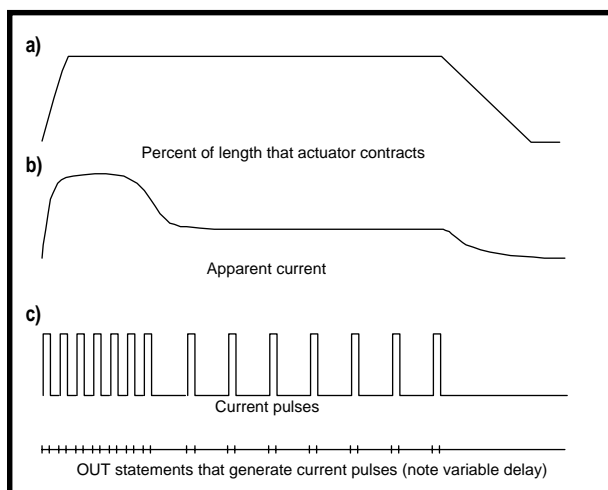


Figure 3a—The nitinol wire will contract and stay contracted until it cools. **b**—To stay contracted, nitinol wire needs only 25–35% of the current needed to initially heat it. **c**—Pulsing current with a 25–35% duty cycle will keep the wire contracted.

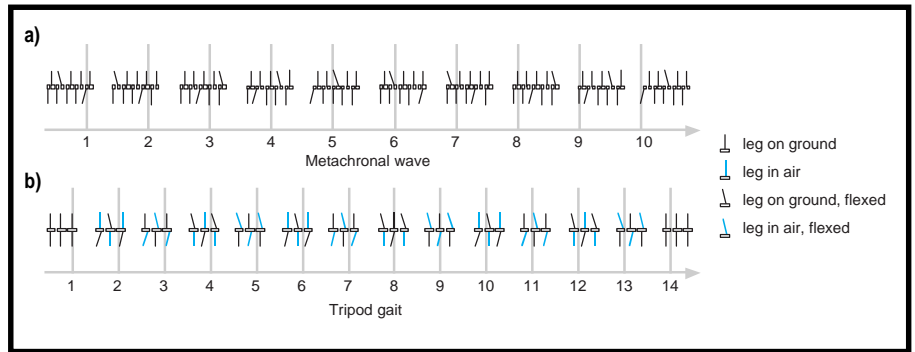


Figure 2—The robot walks best on a slightly rough surface, like linen tablecloths or roughly sanded wood. Compare the metachronal wave gait (a) versus the tripod gait (b). Check out the Stiquito supplemental web site for BASIC programs used to make the robot walk in a tripod gait.

from which you can start experimentation for making the robot walk. The instructions in the book show how you can create a Stiquito that walks in a tripod gait. The circuitry and computer programs we've shown in this article enable you to control this tripod gait via a PC parallel port.

If your plans include independent control of each of the Stiquito's legs, you should modify the assembly of your robot such that you attach control wires to each leg individually. If the design of your robot includes putting something on top (e.g., a circuit that enables it to walk on its own), you should consider how you want it to walk.

If you simply want the robot to walk, a tripod gait may be sufficient. But, if you plan to put complex circuitry like a microcontroller on top, you may want the flexibility of being able to control all six legs. ☒

James Conrad is an engineer at Ericsson Inc., and an adjunct professor at North Carolina State University. He has written on the topics of robotics, parallel processing, artificial intelligence, and engineering education. You may reach him at jconrad@stiquito.com.

Jonathan Mills is an associate professor in the Computer Science Department at Indiana University and director of Indiana University's Analog VLSI and Robotics Laboratory, which he founded in 1992. Jonathan invented the Stiquito to use in multirobot colonies and to study analog VLSI implementations of biological systems. You can reach him at stiquito@cs.indiana.edu.

SOFTWARE

Software for this article is available via the Circuit Cellar web site. The parts list and photos of the finished product are posted there as well.

REFERENCES

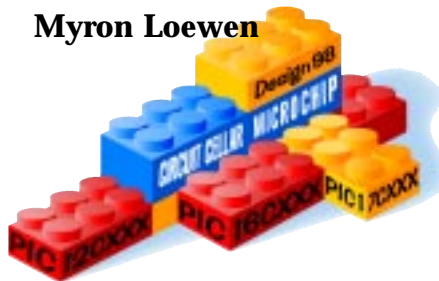
- J.M. Conrad and J.W. Mills, *Stiquito: Advanced Experiments with a Simple and Inexpensive Robot*, IEEE Computer Society Press, Los Alamitos, CA, 1997.
- J. W. Mills, *Stiquito: A Small, Simple, Inexpensive Hexapod Robot. Part 1: Locomotion and Hard-wired Control*, Technical Report 363a, Computer Science Department, Indiana University, Bloomington, IN, 1992.
- Stiquito information, www.computer.org/books/stiquito; www.stiquito.com

SOURCE

Stiquito books
 IEEE Computer Society
 (800) 272-6657
 (714) 821-8388
 Fax: (714) 821-4641
www.computer.org/cspress/catalog/bp07408.htm

FEATURE ARTICLE

Myron Loewen



Internet Appliance Interface

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



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

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

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

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

tion of systems-on-a-chip complete with TCP/IP stacks.

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

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

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

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

SIMPLE INTERNET

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

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

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

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

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

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

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

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

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

Your friend then passes it to another friend who is closer to the final desti-

nation, and the process repeats. The mail doesn't always take the same path and sometimes it gets lost along the way. With luck, the large envelope eventually ends up at its destination.

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

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

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

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

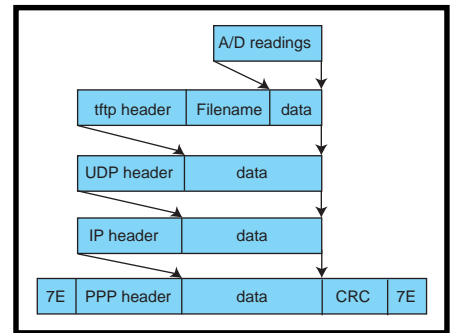


Figure 1—The tftp packet shows how the A/D readings are wrapped in layers of packets to get a PPP packet.

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

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

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

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

NEGOTIATING PPP

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

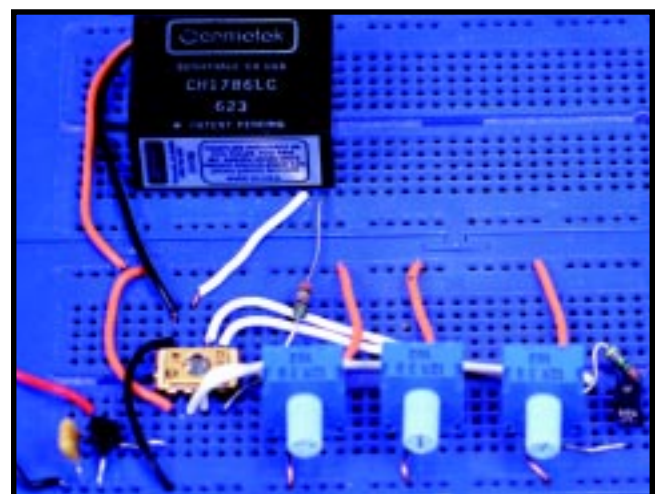


Photo 1—The first prototype of the Internet Appliance uses three potentiometers for remote inputs and an LED to test remote control. The 2400-bps Cernetek modem provides a fast enough Internet connection to exchange the control packets.

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

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

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

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

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

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

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

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

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

Basically, the packet can request, deny, and accept options. Both sides

must issue an accept before the LCP negotiation is complete. Figure 3 depicts the packet exchanges.

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

a NAK and a list of the options that it rejected with parameters that would be acceptable.

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

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

Digging Deeper

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

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

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

To: rfc-info@ISI.EDU

Subject: getting rfc's

Message body: help: ways_to_get_rfcs

Here are some helpful RFCs to get you started:

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

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

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

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

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

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

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

Once the LCP options are agreed on, the PAP authorizes your user ID and password as I described. Then, the CCP compression options are negotiated. These options compress the entire serial stream and could easily use up the entire memory space by themselves. Because my packets are

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

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

tiny and traffic is low, I chose to disable them and go under the puddle-jumper option number three.

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

IP PACKETS

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

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

The 16-bit identification identifies each packet and is used with the following flags and fragment offset to

reassemble fragmented packets. Following all that is the time-to-live byte, which sets the maximum number of hops this packet can be routed toward the destination before giving up. The next byte indicates what type of protocol is riding in the IP data.

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

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

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

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

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

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

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

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

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

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

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

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

EXCHANGING DATA

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

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

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

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

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

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

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

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

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

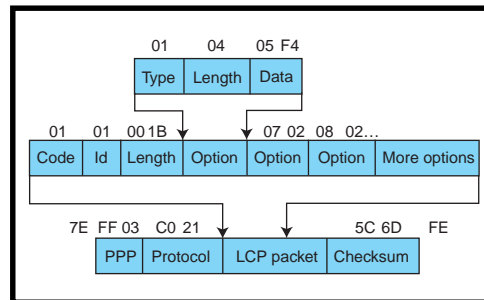


Figure 2—The LCP packet shows how the configuration options are sent in a PPP packet. This example highlights the MRU option.

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

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

THE HARDWARE

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

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

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

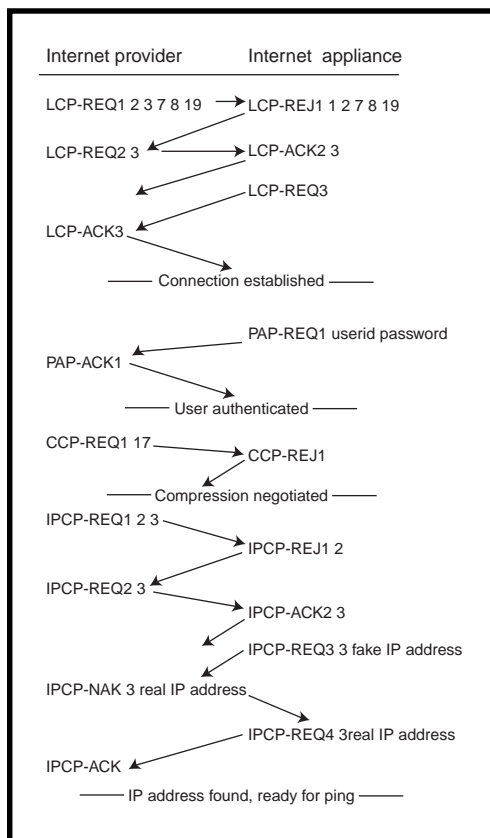


Figure 3—This is a typical exchange of option requests in the negotiation of a PPP connection. Each request (REQ) is numbered and the replying acknowledgment (ACK) or reject (REJ) must reference the same number. The list of numbers at the end of a packet identify each option as highlighted in Figure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

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

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

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

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

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

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

THE SOFTWARE

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

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

I did all the PPP algorithm development on a laptop computer with Borland's Turbo C. It was great because I

could use the internal modem, sprinkle `printfs` everywhere, and trace through the problem areas of code. This method worked so well that I logged onto the Internet after only two weekends of coding and debugging.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

I started off thinking the whole TCP/IP stack could be squeezed

in and ended up with little more than the PPP negotiation, ping, and tftp. I was excited at how easily it worked with one ISP server but was frustrated by differences between other servers.

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

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

SOFTWARE

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

REFERENCES

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

RESOURCES

Microchip PIC12C672 datasheet, www.microchip.com/download/lit/picmicro/12c67x/30561a.pdf
Cermatek 1786LC datasheet, www.cermetek.com/pdf/ch1786%20rev%20o.pdf

SOURCE

PIC12C672

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

Modem

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

Potentiometers

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

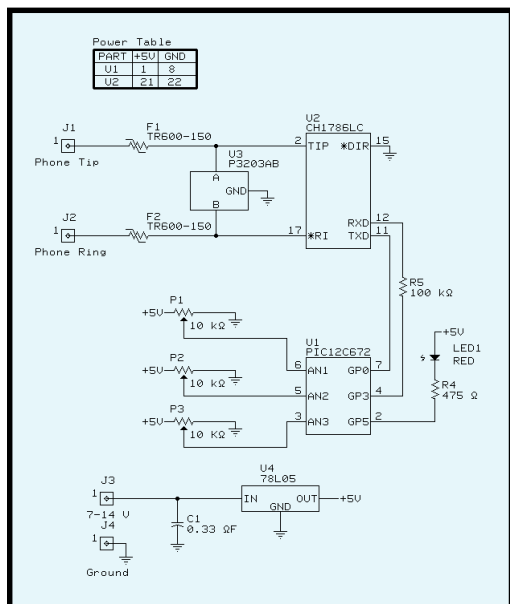


Figure 4—This schematic shows the details of the prototype in Photo 1. These processor ports were chosen to maximize the versatility of the remote accessible pins. Pins AN1, AN2, AN3, and GP5 can be configured as digital inputs or outputs, or pins AN1, AN2, or AN3 can be 8-bit analog inputs. R5 limits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

ANALOG AND DIGITAL I/O BOARD

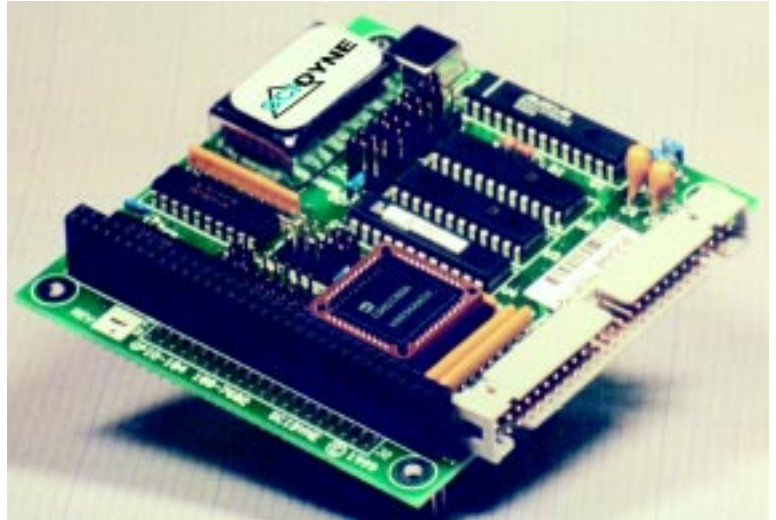
The **GPIO-104** is an 8-bit analog and digital I/O module that combines the most-requested analog and digital I/O functions in one PC/104-compliant module. Applications include industrial automation and process control, and scientific apparatus and instrumentation.

The eight single-ended 12-bit analog inputs are independently programmable to operate in one of four ranges: ± 10 , ± 5 , $+5$, or $+10$ V. This effectively increases the dynamic range to 14 bits when using range-switching software techniques.

The ADC operates at up to 100 kilosamples per second and enables the separate acquisition and conversion intervals to be controlled by the host software or automatically timed by the GPIO-104 hardware. Overall timing is precisely maintained by a crystal oscillator.

Four 12-bit analog outputs are provided, each with its own range: ± 5 , $+5$, or $+10$ V. An onboard DC/DC converter permits the bipolar and $+10$ -V ranges to be achieved while operating the module from a $+5$ -V power supply. The analog outputs can be updated simultaneously using one software command—a necessity in phase-critical applications like x-y positioning.

The 24 DIO lines are TTL-/CMOS-level compatible and offer programmable port directions and strobed handshaking. A standard J1/P1 stack-through connector enables the GPIO-104 to reside anywhere within an 8-bit PC/104 stack. An optional J2/P2



connector provides 16-bit stack-through compatibility and access to all upper interrupt request lines.

The GPIO-104 sells for **\$237**.

Scidyne
(781) 293-3059 • Fax: (781) 293-4034
www.scidyne.com

SINGLE-BOARD COMPUTER

The **PCM-5894** is a PC-compatible, Pentium MMX/K6 SBC in a compact 5.75" \times 8" form factor. Certified by Annasoft for Windows CE 2.0 operation, it enables customers to use their knowledge of Windows API to develop embedded applications with minimal size and complexity.

The PCM-5894 supports Intel P54C/P55C, AMD K5/K6, and Cyrix M1/M2 CPUs. It supports Socket 7, features the SiS 5582 chipset, and has Award flash BIOS. Two 72-pin SIMM sockets can handle up to 128 MB of DRAM, and 512 KB of onboard cache is included.

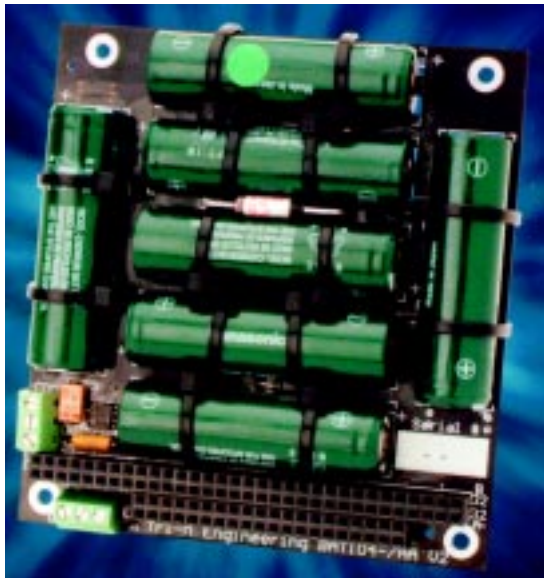
Other features include Chips and Technologies 65554 video chipset (64-bit video with 1024 \times

768 video resolution at 64K colors), four COM ports (three RS-232 and one RS-232/-422/-485), a fast 100BaseT Ethernet, up to 72 MB of DiskOnChip with boot-up support, superb power management, multimode parallel port (SPP/EPP/ECP), an Ultra DMA/33-enhanced IDE controller, keyboard interface, and PS/2 mouse interface.

The PCM-5894 sells for **\$444** with integration and **\$404** without integration.

Emac, Inc.
(877) 724-3963
(618) 529-4525
Fax: (618) 457-0110
www.emacinc.com





PC/104 BATTERY BACKUP MODULE

Tri-M Systems' **BAT104-7AA Battery Backup Module** works with power management, battery charger, and power supply units to create a complete UPS system. For loads less than 30 W, the BAT104-7AA can supply backup power for up to 5 min. If a larger load is needed, two units can be plugged together.

The BAT104-7AA can be used as a battery backup in embedded vehicle applications based on the PC/104 architecture. Power interruptions resulting from engine startups or power system switchovers aren't compatible with modern OSs. Unix and Windows systems require the PC to properly close all files and applications before the power can be terminated. This power-down cycle may take 30–90 s, which is far longer than the hold-up charge in the capacitors. The BAT104-7AA is a perfect solution to these power interruptions in embedded applications.

The BAT104-7AA has a thermal fuse and a current fuse for protection against overcharging and shorts on battery output. This LM35 temperature sensor provides temperature feedback for charge termination and can be read by the power management unit.

A low-resistance isolation MOSFET connects the batteries to the output whenever a +5-VDC signal is present. The power management system can therefore isolate itself from the BAT104-7AA by turning off the power-supply output.

The BAT104-7AA is priced at **\$149**.

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

INDUSTRIAL SBC

The **Model 5811** is a PC/104-*plus* industrial SBC that supports the Pentium MMX and 450-MHz AMD K6-2/3D on a 100-MHz system bus. It addresses up to 512 MB of ECC SDRAM on two 168-pin DIMM sockets. The ECC features can reduce system failures, increasing the success rate in many mission-critical applications.

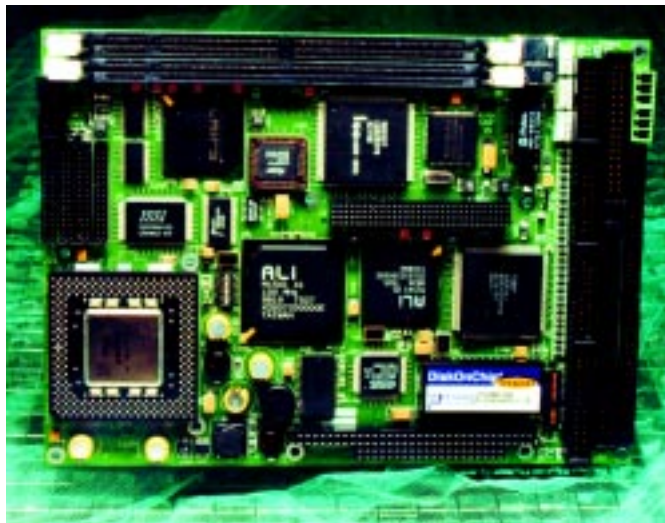
The SBC supports all of the features typically required in a flat-panel computer, including touchscreen interface, four serial ports, LCD interface, network interface, 144-MB flash disk, and ultra fast and wide SCSI interface. An onboard LVDS interface capable of driving the 36-bit high-resolution flat-panel display at distances up to 20' is included.

The integrated 10/100-Base-T network interface (which uses an Intel 82558) and the PCI-based bus master ultra fast and wide SCSI (using Sym-

bios 53C-875) ensures that software compatibility and performance needs, as well as the long-term support required in many industrial applications, are satisfied. System designers only need to add power supply, flat panel, and the enclosure to complete their system.

The Model 5811, with a 350-MHz K6-2/3D processor, 10/100Base-T, LCD, SCSI, touchscreen interfaces, and four serial ports, sells for **\$775**, in quantities of 100.

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



Nouveau **IPC**

Jeanne Dietsch,
William Kennedy,
John Belanger & Kurt Konolige

PC/104 Takes a Ride

Embedding a PC in a Robot

When Activmedia Robotics was looking to improve the Pioneer robot, the number-one requested enhancement was an onboard PC. But how do you embed a PC on a mobile robot? The first task: catch up with the robot!

Starting a new project typically involves running up against some rather basic constraints or problems. The project of embedding a PC in a robot is no exception.

The first realization that confronts you when you try to put a PC in a robot is that your PC is no longer sitting placidly at your fingertips. In fact, your PC is now scooting around the room, doing its best to avoid you, as well as the desks and chairs.

Unless you plan to chase the beast around the room watching its display scroll, you'll want an offboard means to monitor what's happening. You'll probably want to program via an Ethernet connection—at least for debugging.

In a way, linking to an offboard PC appears to defeat the purpose of embedding the PC in the first place. So, why did we choose to embed a computer in our robot?

In fact, when Kurt Konolige of SRI's Artificial Intelligence Center first

designed the Pioneer 1 robot, he used an external (nonembedded) PC partly for that very reason—not to mention the fact that it costs more to build a robot with an internal embedded PC. Because the \$2500 Pioneer 1 robot could perform many tasks

that previously required a \$20,000 robot, it was an instant hit even though it only had a 68HC11 microcontroller server inside.

A DAY IN THE LIFE OF A MOBILE ROBOT

As we pass into the new millenium, highly intelligent mobile robotic butlers like C3PO still only exist in the minds of visionaries like George Lucas. What *can* today's mobile robots do?

Don't fire your cleaning service yet, but mobile robots can vacuum. Although they don't vacuum at rates anyone but the Gates estate can afford, the day of Ray Bradbury-like vacuuming robotic mice is not so far off (also, see Frank Jenkins' "HomeR" article in *Circuit Cellar* 81).

NASA and DARPA have taken over support of many of the activities of the artificial intelligence and robotics research and development community. They are building a

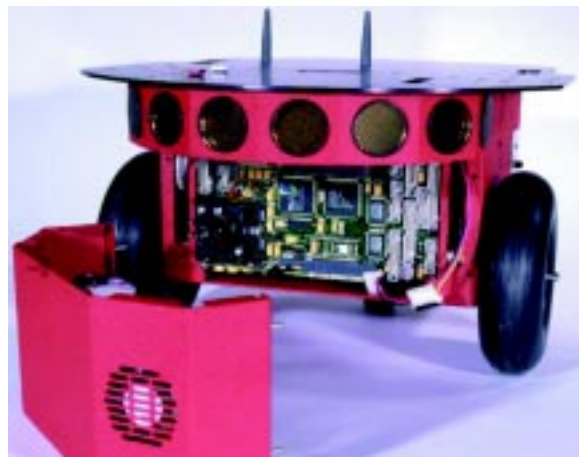


Photo 1—The Pioneer 2-DX shows off its new embedded computer, complete with PC/104 and PCI buses. The Pioneer's removable nose allows easy access to add PC/104 accessory cards for GPS, video capture, speech recognition, and more.

wealth of new knowledge that is hastening the day when remote handling and reconnaissance robots form the front line in space and military operations.

So, the highly affordable Pioneer 1 found a ready home. Engineers looking for prototyping platforms or a COTS base for robotic experiments or applications found the Pioneer appealing. The artificial-intelligence research community adopted it as the VW of mobile robotics research.

Because the Pioneer came with built-in libraries of obstacle avoidance and other navigation basics, it easily jump-started many projects. Instead of raising tens of thousands in grant money or building their own robots, researchers could now focus on the true topic of their research: robotic behaviors, experiments, or commercial applications.

Since 1997, Pioneers have dominated American Association for Artificial Intelligence annual contests. They also took first and second prizes in the World RoboCup Soccer Championship in 1998. But, even these winners made clear the Pioneer 1's shortcomings because all six Pioneer teams at the World RoboCup Soccer Championships wore strapped-on PCs! One team went so far as to saw their Pioneer robot in two, in order to fit computers inside!

And if we didn't have proof enough, when we asked team members for a wish list for the next generation of Pioneer, guess what was number one on their wish list. That's right, an onboard PC!

WHY BUILT-IN?

Why were they so desperate to have PCs onboard if they have to program them from the desktop anyway? First, in a communications-jammed environment, any

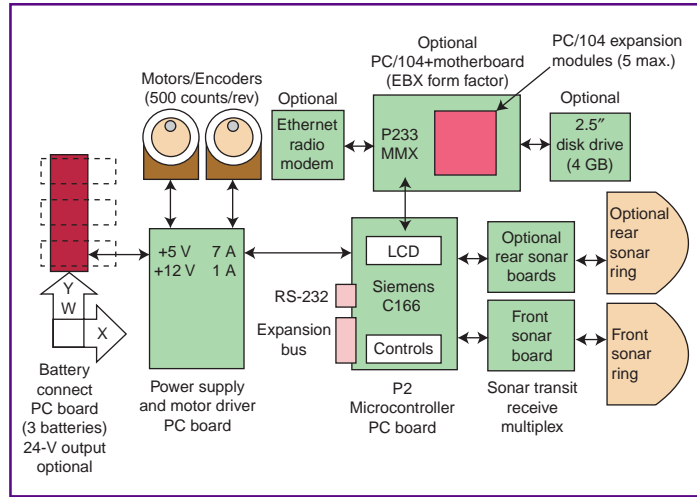


Figure 1—*This illustration shows the various printed circuit assemblies mounted in the robot. The basic configuration includes a battery connect board, power supply/motor driver board, microcontroller board, and front sonar board.*

Speaking of Ethernet, higher communication speed was another carrot enticing us to move to an embedded PC. Many bleeding-edge AI research

projects (and the government grants funding them) deal with distributed intelligence and multiagent systems.

The Pioneer 1s messaged each other via radio-modem pairs, but this peer-to-peer arrangement required a pair for each robot or custom engineering to manage the signals. With an embedded Ethernet port on every robot, we could vastly increase our communication speeds. Radio Ethernet access points and our Aylla multi-agent software directs messaging among the various computers, embedded and otherwise.

Another reason for moving to embedded is sensor based. We wanted to add a few more sensors to the robot, including a laser range finder (10 KBps, 10–30 Hz), inertial navigation system (1 KBps, 100Hz), motion radar (20 KBps), and video camera (5 MBps, 30 Hz).

Many of these sensors use serial ports for their data output, and it would take custom hardware to package these outputs and ship them out by wireless Ethernet. Even more problematic is synchronizing the signals.

For example, many robot mapping and localization algorithms rely on fusing

wireless link is a weak link. In any contest, demo, or even most applications environments, available bandwidths are jammed with competing signals.

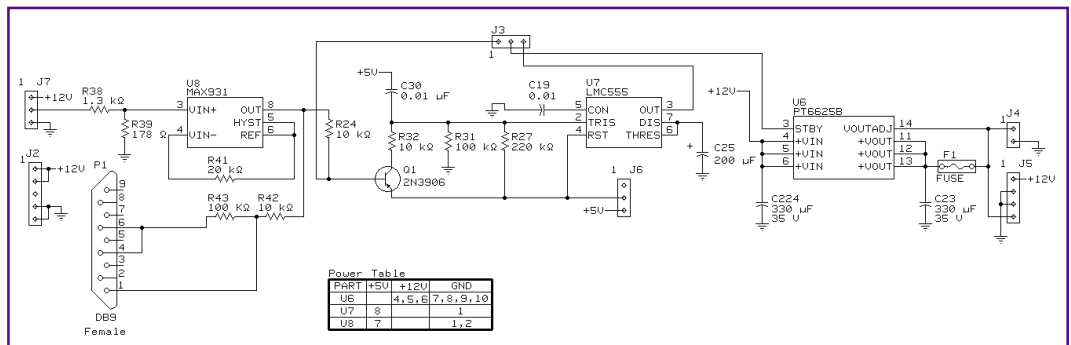
At 900 MHz, cell phones compete. At 410 and 433 MHz, you need FCC approval and local TV stations may interfere. Audio-video transmitter/receivers and other high-throughput applications tend to slice up the 2.4–2.83-GHz bandwidth.

That leaves little free space, even for radio Ethernet devices that employ frequency hopping to minimize interference. The less-expensive 900-MHz radio modems used by many university researchers were hopelessly overwhelmed.

The second reason the researchers and engineers desperately wanted an onboard PC was so they could analyze vision data in real time. Even 1.6-Mbps Ethernet moves data too slowly to handle real-time visual data efficiently.

An embedded PC's PCI bus, with up to 132-Mbps throughput, is a whole different story. That's two orders of magnitude improvement. The researchers processing images for shape identification and navigation purposes could now use our system plug-and-play (see Photo 1).

Figure 2—*If the Pioneer 2's battery voltage drops below 10 VDC, or when the computer power is switched off, an embedded circuit delays shutoff of the onboard computer's dedicated power supply (12:5 DC/DC voltage converter) for 2 min. Software operating in the background senses the state of the DCD line on the circuit's serial connector and gracefully shuts down the computer before power is removed.*



together successive sensor readings, with odometry giving a rough indication of how they align.

Because the Ethernet data-stream is not deterministic, packets for the different sensors would arrive at different times. The robot then requires a complicated timestamping system to sort things out when the packets finally arrive.

PICKING THE RIGHT PC

We had a short (but stringent) list of requirements when we set out to select an

embedded-PC technology. The list of requirements included:

- SBC readily available off-the-shelf
- small form-factor
- PCI expansion bus
- Ethernet port
- Linux and Windows support
- readily available expansion modules for I/O, sound, video capture, etc.
- reliable supplier

The wish list was also short, but it seemed a more difficult challenge:

- large number of serial I/O ports
- low power draw
- concomitant low heat generation
- high-speed CPU
- good onboard I/O capabilities
- expandability
- modest price
- good operating temp and shock/vibe specs
- high reliability

We settled first on the EBX form factor with PCI-enabled PC/104-Plus expansion bus as our desired configuration. The Ampro Little Board/P5e met all the requirements and also met many of our wish-list desires.

The availability of Linux drivers was a critical feature. Linux is important not just because the AI community prefers Unix-like OSs but also because of a unique property of robots mentioned earlier—robots move while you're debugging them!

One solution is to run the embedded computer as an X Windows terminal under Linux from an offboard computer during the software development process. This can also be done under WIN with Timbaktu. But, being integrated and free, X Windows is the preferred solution among academics.

An alternative debugging solution is to tether and dolly the robot and watch the screen and wheel motions. Or, you can plug a monitor and keyboard into the robot's control panel and follow it around. Obviously, terminal emulation saves shoe leather.

CHALLENGES GALORE

The Little Board/P5e also offered reasonable speeds, excellent I/O, expandability, and a reasonable cost. The only feature we failed to find at that point was low power drain and low heat, so we did our best to accommodate with battery capacity and mechanical design.

Robotics designers try to minimize space because the more space inside the robot, the more weight in its case. The more weight in the case, the larger the motors and number of batteries are required.

Placement of the embedded PC within the robot was a difficult decision. Where do we put the board so users can access it? How does it integrate with the rest of the robot electronics (see Figure 1)?

We wanted to allow as many PC/104 expansion modules as practical. This led to the decision to use a 2.5" hard drive, even though it was more expensive than a

standard 3.5". We needed the extra space to shock-mount the drive.

The Little Board/P5e was simple enough to cable to for power and to connect one of its serial ports to our robot's built-in micro. More difficult was the decision as to which other capabilities to bring to our control panel on the robot's exterior. We decided to bring out the serial mouse, CRT interface, Ethernet port, and reset button.

Robots have a tendency to be powered down suddenly if their batteries run low or if users arbitrarily turn them off without first shutting off the computer. So, we designed and embedded delay circuitry to help the robot power down gently (see Figure 2).

A TASTE OF TILLAMOOK

Although we didn't know it then, the solution to one of our compromises (power drain) was just around the corner: Ampro's Little Board/P5x, based on Intel's Tillamook mobile Pentium with MMX technology.

The P5x also improved the robot's vision-processing capabilities. With the Imagination PXC200 frame grabber, the Little Board/P5e could process 16-bit color at 30 frames per second.

But, with the P5x's faster PCI bus (based on the Intel TX chipset), we could offer full 32-bit color at 30 frames per second as well as faster processor speeds. All this, with lower power drain and less heat generation.

The newest generation of Pioneer 2s, with the internal EBX embedded PC option, was an instant hit. Because we used an off-the-shelf embedded PC, our research and development costs were low. And thanks to manufacturing changes, we brought in the new base platforms at a price below that of the Pioneer 1s! [EPC](#)

Jeanne Dietsch is founder and vice president of business development for ActivMedia Robotics, her third start-up, including a venture with Pat McGovern, chairman of IDG. She has spoken at Comdex, CES, and other events on new technologies and has worked in computer-related fields since 1981. You may reach her at jdietsch@activmedia.com.

William Kennedy is president and chief technologist for ActivMedia Robotics. He has been working in robotics for three years and handled circuitry and electronics design for the embedded computer in Pioneer 2. You may reach him at wkennedy@activmedia.com.

John Belanger is senior mechanical engineer for ActivMedia Robotics. He collaborated with Kurt Konolige on designs for all the new Pioneer 2s, including mechanical design for embedding the Ampro LB3-P5e board in Pioneer 2. John has 35 years of involvement in all aspects of electronic packaging. You may reach him at jbelanger@activmedia.com.

Kurt Konolige, designer of both the Pioneer 1 and Pioneer 2 robots, is a senior computer scientist at the Artificial Intelligence Center of SRI International and a

consulting professor of computer science at Stanford University. His current research interests are fuzzy control for reactive systems, real-time vision systems (especially stereo), and mapping and navigation for mobile robots. You may reach him at konolige@ai.sri.com.

SOURCE

Little Board/P5e
Ampro Computers, Inc.
(408) 360-0200
Fax: (408) 360-0222
www.ampro.com

Astronomical Issues

Part 4: Digital Radio Software

T minus four, three, two, one! It's time to complete this series on radio astronomy! In this final installment, Ingo focuses on the host software and FPGA configuration so you too can tune in to radio emissions from Jupiter.

OK, so here I am at the end of this series on a digital receiver for radio astronomy projects. Although this is the last part, the project is far from complete.

One reader pointed out that it would be nice to be able to duplicate projects like this. The current state of this digital receiver project uses mostly off-the-shelf components that were in my lab, including FPGA modules from Derivation Systems and a PC/104-compatible CPU from Versallogic. The A/D modules are Analog Devices and Burr-Brown evaluation modules.

About the only components custom designed for this project are the CORDIC and decimation filter implementation in an FPGA configuration (also called the bitstream) that can be downloaded to the FPGA module and the host

software to read out the data from the FPGA modules. This month, I talk about the host software. Both the FPGA configuration and examples of the host software are on the *Circuit Cellar* web site.

The base components for the current implementation of this project probably

aren't inexpensive enough for someone to go out and purchase them just for a project like this. I used stuff I had around the lab, but I realize that not everyone has these components in their parts box. In the near future, I plan to reimplement this project on a more available platform.

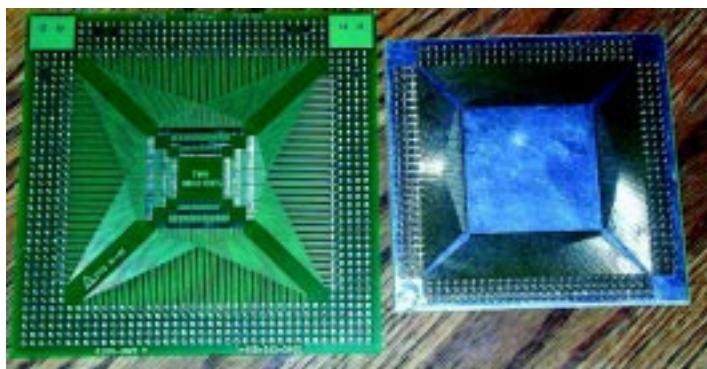


Photo 1—Many FPGAs come in quad flat pack (QFP) packages, and these surface-mount adapters allow you to use them in a prototype system where you may want to wire-wrap or solder the chip to other components that use through-hole mounting. The commercial adapter (on the left) fits different packages up to 240 pins. On the right is an adapter I designed for a 304-pin QFP package.

FPGA FOR PROTOTYPING

The new implementation will still use FPGAs because the technology acts as a sort of enabling technology for doing this kind of high-speed DSP. It would be impractical to implement this design using 74xxx TTL parts. Let me demonstrate by showing you the number of chips you'd need to implement a 12-bit pipelined CORDIC.

Each stage of the CORDIC would be constructed with nine 4-bit adders to implement,

three 12-bit adders, five 8-bit registers, and 14 total per stage. With 12 stages, that gives you 168 74xxx chips.

I believe that FPGAs and CPLDs are pretty much going to be the future for implementing prototypes and low-volume digital hardware. Of course, one of the stumbling blocks for using them is the software you need to implement your design. But, even the software is cheaper and much more available.

If you're a student or affiliated with a university, Xilinx will sell you a student edition of their software with a book for \$79. Otherwise, the entry-level price for their software is \$500.

By the time you buy the hundreds of TTL chips, the wire-wrap wire, and the prototype board to implement the equivalent of a single low-density FPGA, you'd have spent almost as much money as if you had bought the design software that enables you to implement circuits like this in FPGA. I agree that \$500 is a bit steep just to play, but both Xilinx and Altera offer free web-based access to their tools.

Another stumbling block for working with FPGA and CPLDs is the availability of parts in packages that you can easily solder. My favorite package is the 84-pin PLCC (J-head) package.

The package is inherently surface-mount technology (SMT), but reliable through-hole sockets (which can be plugged into wire-wrap sockets) on 0.1" centers are available. Most FPGA and CPLD vendors have low- and medium-density parts available in this package.

If you need more I/O pins than the 84-pin package provides, the jump to plastic quad flat pack (PQFP) isn't too scary. A common package is the PQFP-160. Several vendors make SMT-to-through-hole adapters like those in Photo 1.

But, you still need to solder the package on the adapter. Avoid the test-socket type adapters: they're unreliable and expensive.

Soldering these packages isn't so hard. First, align the chip on the pads. This is the hardest part and takes the most patience. Then, with a fine-pitch soldering iron, solder only the corner pins to the pads and don't worry about solder bridges.

Once the chip is aligned and tack soldered to the corner pads, apply a bunch of solder to the rest of the pins. Don't worry about shorting the pins but keep the solder near the pads on the pins.

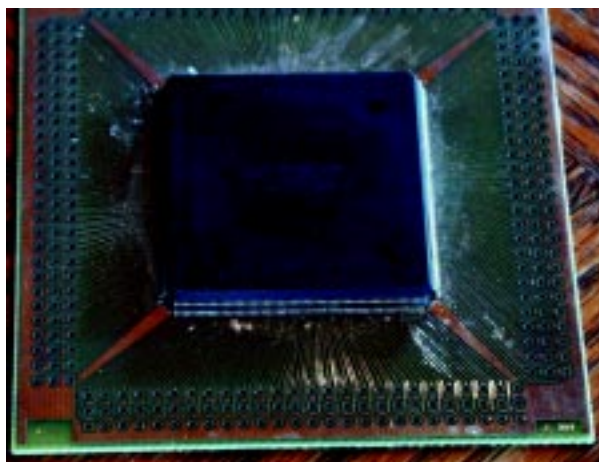


Photo 2—This dummy chip was mounted on my adapter using the solder-wick method. Even though it's not very hard, it pays to practice your technique with junk or dummy chips, especially using lower pin count and coarser pin spacing than this 304-pin monster.

Now that there is solder everywhere, show it to your boss. Just kidding. Use some solder wick to pull the solder away from between the pins and leave a perfect connection, like the one shown in Photo 2.

This package is a 304-pin QFP. Probably not the best package to start with, but it illustrates that it can be done. I'm pretty sure 304 pins is the largest PQFP package, with about 0.020" pin spacing. The finest pin spacing is in a 128-pin QFP where it is approximately 0.015". The 160-pin QFP has a pin spacing of around 0.025" and is a good package to start with.

Incidentally, there is a company that sells "dummy" chips for practicing your SMT soldering technique. Unfortunately, they only sell them in "trays." If any of you are interested in some single dummy chips, I can organize a group buy. By the way, I do have some more 304-pin chips....

By now, using an FPGA shouldn't seem as scary as it did at first. Chips are available; there's free software on the web. If you're a student, you can get cheap software. Chips are available in packages that can be used in prototyping, and the databooks and app notes from most vendors are available online.

Now you know why I use FPGAs in many of my designs, even if they end up in a digital-radio implementation on a PC/104-based PC. Let's look at how I designed and implemented the FPGA design for this project.

FPGA DESIGN QUICKIE

In general, I'm stingy. Unless I can get software and hardware at a low cost, I tend to roll my own. For example, the CPU board I used here is used for many other projects and serves as my hardware, software, and RTOS development system.

I did the PCB layout for my PC/104 FPGA board using a freely available PCB layout program (see Sources). It's even a two-layer board, so it's less expensive to fabricate.

Because I don't have the budget to buy high-end design software for doing VHDL or Verilog designs, I use an ad hoc way of designing for FPGAs. The "source" or design capture is done in a netlist language I developed for my own use. The language has a simple syntax that lets me specify networks conveniently.

Listing 1 shows how I specified the phase register in my CORDIC implementation. Notice that each line starts with a keyword like power or gate. The gate keyword means I want to instantiate a symbol, which can be implemented using another netlist file or a primitive in the FPGA design system. For example, in Listing 1, I invoked a 12-bit register reg12 and a 12-bit adder addsub12 symbol, as well as primitives and and or.

Each gate line has a list of signal pairs. The first is the name of the port on the subsymbol I want to connect to, and the second is the network or signal name I want to connect the port to. Signal names can be buses (e.g., a[11:0]) or simple signals.

The CORDIC routine is generated (synthesized) in this format with a program because it's too tedious and error prone to generate by hand. You specify the data word size of the CORDIC processor and the script generates a large netlist file with all of the signals and components in it.

An awk script converts these netlist descriptions into the native netlist format that the FPGA design implementation software uses. Awk is a pattern-processing language like Perl or Tcl that's available under Unix or Linux. By changing the translation, I can target different FPGA technologies, assuming I made some intelligent choices about the symbols I used.

Although this approach isn't the best, it has served me for many years and is fairly portable. Of course, over the years Xilinx has changed their netlist format from a readable text-based format (XNF) to a binary data-based format (NGD), which is not documented. They still provide conversion utilities from XNF and EDIF to their internal database format.

To implement my design after converting it to Xilinx specific netlists, I used the normal Xilinx FPGA implementation tool chain, which processes the netlist, maps it to the specific internal logic block representation, and then places and routes the design into a configuration file.

The board I built uses three FPGAs. Each FPGA has its own netlist files and is

implemented separately. The configuration files are concatenated into one large configuration file and saved in an ASCII-formatted hex file. These are then downloaded into the board.

I bundled up the Xilinx netlist files, the files needed to tell the Xilinx software how to assign the I/O pins for my FPGA board, and the program needed to download the design into the board, and put it all on *Circuit Cellar's* ftp site. You can use the Xilinx design implementation software (the commercial or student edition) to convert these to the FPGA configuration files. If you don't have access to the Xilinx implementation software, don't worry. I also included the final configuration file that you can directly download to the board.

Listing 1—This constant-phase register illustrates the netlist file format I use. At each clock tick, one gets added to the current phase. The back end makes sure that the phase is calculated correctly for the CORDIC implementation.

```
# 12-bit triangle wave generator between 3ff and c01
power 0 zero11
power 0 zero10
power 0 zero9
power 0 zero8
power 0 zero7
power 0 zero6
power 0 zero5
power 0 zero4
power 0 zero3
power 0 zero2
power 0 zero1
power 0 zero0
power 0 one13
power 0 one12
power 0 one11
power 0 one10
power 0 one9
power 0 one8
power 0 one7
power 0 one6
power 0 one5
power 0 one4
power 0 one3
power 0 one2
power 0 one1
power 1 one0
power 1 vcc
# here is the phase accumulator
gate addsub12 add vcc a[11:0] ang[12:1] b[11:0] one[11:0] o[11:0]
  n[12:1]
gate reg12 c c ce ce d[11:0] n[12:1] q[11:0] ang[12:1]
#
gate and 1 /ang12 2 /ang11 o quad1
gate and 1 /ang12 2 ang11 o quad2
gate and 1 ang12 2 /ang11 o quad3
gate and 1 ang12 2 ang11 o quad4
gate or 1 quad2 2 quad3 o neg
# correct up/down slope
power 0 ang0
# negate in quadrants 2 and 3
gate addsub12 add /neg a[11:0] zero[11:0] b[11:0] ang[11:0]
  o[11:0] o[11:0]
```

RADIO SOFTWARE

In my digital radio, the input from the antenna is amplified and low-pass filtered before digitizing with a high-speed ADC. The low-pass filter is an antialiasing filter, which ensures that only the Nyquist band we're interested in is sampled and converted. Last month, I explained that the low-pass filter can be replaced by a band-pass filter to convert other Nyquist bands by undersampling the signal.

After the signal is digitized, use a CORDIC implementation and phase register to implement a direct conversion receiver. The CORDIC multiplies the input signal by cosine and sine of a particular phase. The phase register is changed by a fixed phase angle on each clock cycle.

In effect, you're multiplying the input signal with the sine and cosine of a numerically controlled oscillator (NCO). This process is called heterodyning, which, in the frequency domain, corresponds to shifting the input signal by the frequency of the NCO.

Two products exist after the multiplication—the sum and difference of the signal

and the NCO frequency. We're interested in the difference, and filter it crudely using two 16:1 decimation filters.

This process is implemented in an FPGA design on a PC/104 FPGA module. The result is a datastream of the in-phase $I(t)$ and quadrature signal $Q(t)$ at $\frac{1}{256}$ of the data rate used to sample the original data. We can use the frequency setting word (FSW) of the NCO to select the frequency of the signal shift and thus continuously tune the receiver.

To read the output datastream of the downsampled radio spectrum, use the 8-bit ISA-bus I/O mapped interface in Figure 1. The software can set the FSW in the FPGA using the control/status register (CSR). The FSW is implemented via a three-wire shift register (see Listing 2).

The CSR contains the interrupt-enable bit that enables the interrupts from the FPGA card. Reading the CSR lets you check for a pending interrupt and clears the interrupt if it was pending.

The data register is implemented using an 8-bit port into a 32-bit shift register,

Listing 2—This Tcl script sets the frequency of the radio. This script includes an external module that implements functions to access the PC/104 I/O space directly. The 16-bit quantity is shifted into the FSW using the CSR on the FPGA board.

```
#!/usr/bin/tclsh
load ./jvm-demo/lib/pf2kio.so
io_init 0x270
io_outb 0 0x270
if {$argc > 0} {
    set fsw [expr [lindex $argv 0] | 0x10000]
} else {
    set fsw 0x000
}
io_outb 0x04 0x270
for {set i 16} {$i >= 0} {incr i -1} {
    puts -nonewline [expr ($fsw >> $i) & 1]
    if {[expr ($fsw >> $i) & 1] == 1} {
        io_outb 0x6 0x270
        io_outb 0xe 0x270
        io_outb 0x6 0x270
    } else {
        io_outb 0x4 0x270
        io_outb 0xc 0x270
        io_outb 0x4 0x270
    }
}
puts ""
io_outb 1 0x270
```

Listing 3—This interrupt service routine reads the radio board. On each interrupt, we clear the interrupt-pending register and read the I and Q values, adding them to a couple of FIFOs for the upper level routines to read.

```
void intr_handler(void)
{
    short x[2];
    outb(inb(0x270),0x270); /* clears interrupt and pointer */
    insb(0x271,&x,4);
    rtf_put(0, &x[0], 2); /* I queue */
    rtf_put(1, &x[1], 2); /* Q queue */
}
```

Listing 4—This Tcl script controls the radio over the entire radio spectrum and collects a power sample at each dwell point. It was used to collect the data for the spectra in Part 2.

```

#!/usr/bin/tclsh
load ./jvm-demo/lib/pf2kio.so
io_init 0x270
// set the frequency word to the desired frequency
proc setfsw {fsw} {
    io_outb 0x04 0x270
    for {set i 16} {$i >= 0} {incr i -1} {
        if {[expr ($fsw >> $i) & 1] == 1} {
            io_outb 0x6 0x270
            io_outb 0xe 0x270
            io_outb 0x6 0x270
        } else {
            io_outb 0x4 0x270
            io_outb 0xc 0x270
            io_outb 0x4 0x270
        }
    }
}
// collect and time average 512 samples
proc collect { fdi fdq } {
    set power 0
    for {set i 0} {$i < 512} {incr i} {
        set di [read $fdi 2]
        set dq [read $fdq 2]
        binary scan $di "s" x
        binary scan $dq "s" y
        set power [expr $power + ($x*$x + $y*$y)]
    }
    return $power
}
// set up to read the I and Q FIFO channels
set fdi [open "/dev/rtf0" "r"]
set fdq [open "/dev/rtf1" "r"]
fconfigure $fdi -translation binary
fconfigure $fdq -translation binary

io_outb 0 0x270
// cycle through all the frequencies and collect power
puts 512
set fsw 0x10000
for {set i 0} {$i < 512} {incr i} {
    setfsw $fsw
    io_outb 1 0x270
    after 10
    io_outb 0 0x270
    puts [collect $fdi $fdq]
    incr fsw 0x40
}
// we're done
close $fdi
close $fdq

```

Listing 5—This routine implements a simple AM detector. It reads I and Q and computes the vector magnitude. This stand-alone program runs on a Linux machine with Soundblaster cards and pipes the output samples into the sound device /dev/audio.

```

#include <math.h>
#include <sys/file.h>
main()
{
    int fd1;
    int fd2;
    short i,q;
    int j;
    unsigned char amo;
    double sqrt();
    j = 0;
    while(1){
        read(0,&i,2);
        read(0,&q,2);
        amo = sqrt((double)(i*i)+(double)(q*q))/2;
        write(1,&amo,1);
    }
}

```

where the I and Q samples are stored as two 16-bit values.

To read the receiver card, set the FSW word, set up the interrupt vector for the IRQ line that has been selected via a jumper on the card to point to the interrupt service routine (ISR), and enable the interrupts on the card. When an interrupt request comes in, the OS dispatches the ISR.

Now write to the CRS to clear the interrupt and read four bytes from the data port. If more than one board shares this IRQ level, read the CSR and check the interrupt-pending bit to make sure this board was the one that fielded the interrupt.

Once the data is read, the ISR splits it into two FIFOs, which behave like circular buffers—one for I and one for Q samples. Then, get out of the ISR as fast as possible. Listing 3 shows the ISR for this card.

After the I and Q samples are read from the receiver, you can do more processing. To use the receiver as a spectrum analyzer, scan the receiver over the desired frequencies in tuning steps, measuring power along the way. To measure the power, add the squares of the I and Q to each other.

By averaging the readings, you can enhance the receiver's sensitivity because all random noise will average out over time. The longer the averaging time, the more sensitivity can be achieved. The code in Listing 4 was used to make the spectrum plot in Part 2.

Because you have I and Q signals, in theory you can implement almost any demodulator you want. The simplest, of course, is the AM demodulator or detector, which measures the amplitude of the signal by computing the vector magnitude. I implemented a crude AM receiver using the demodulator code in Listing 5.

Compared to the 10-kHz channel spacing you'd need to receive a standard AM broadcast station, our receiver still has

Listing 6—This ISR was used with the AM detector in Listing 5. The I and Q values are decimated some more and then sent over the FIFO. A simple program then reads the I and Q stream from the FIFO and sends them over the network to a machine with a sound card.

```
#define N 14
RT_TASK mytask;
int ints, ptr;
short buf[N];
short i, q;

void intr_handler(void)
{
    outb(inb(0x270), 0x270); /* clears interrupt and pointer */
    insb(0x271, &buf[ptr], 4); /* get i/q */
    ptr += 2;
    if(ptr == N){
        i = q = 0;
        for(ptr=0; ptr<N; ptr+=2){
            i += buf[ptr];
            q += buf[ptr+1];
        }
        ptr = 0;
        rtf_put(0, &i, 2);
        rtf_put(0, &q, 2);
    }
}
```

too much of a bandwidth. We decimated by 256:1, which gives a sampling rate of 8 MHz per 256 (i.e., 31.25 kHz) and a bandwidth of ~16 kHz. Listing 6 band-limits the signal before demodulation, which is done at interrupt time.

FURTHER PROJECTS

This receiver is adequate for observing Jupiter emissions using the undersampling technique and increasing the sampling rate. To increase the sampling rate, you need a faster bus interface; our 8-bit PC/104 bus interface is limited. At 31.25 kHz, you have to read four bytes plus reset the interrupt request, so the transfer rate is $5 \times 31.25 \text{ kHz} = 1.5 \text{ Mbps}$ (about the maximum for an 8-bit ISA bus).

One solution is to use 16-bit I/O or a PCI interface, like that in the PC/104-plus spec. If you only care about the power in the selected band, you can have the FPGA integrate power over some amount of time. Or, you can have one of the FPGAs automatically scan the frequency and collect power samples in its onboard SRAM frequency bins. This arrangement reduces the load on the host system and provides a near real-time power spectrum.

To monitor the 1428-MHz hydrogen band, you need to add a dish antenna, a low-noise amplifier, and a downconverter. You can use an existing receiver that provides the last IF as an output. Some commercial receivers provide a 10.7-MHz IF output, which can be fed into my digital receiver for wide-band digital processing.

I have some other projects brewing as well, such as a WWV time station receiver combined with a web server. So, don't go anywhere; this could get good. [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.

SOFTWARE

FPGA configuration and examples of the host software are available on the Circuit Cellar web site.

SOURCES

A/D modules

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

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

FPGAs

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

PC/104 CPU

Versallogic Corp.
(800) 824-3163
Fax: (541) 485-5712
www.versallogic.com

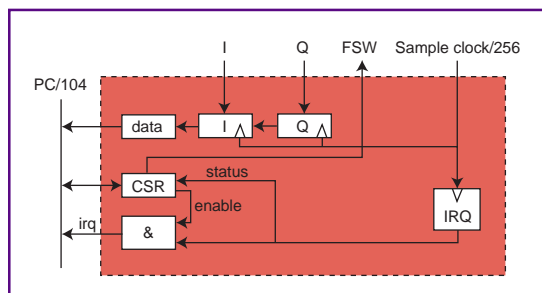


Figure 1—Here is the bus interface for the digital radio. The I and Q samples are stored in a shift register that can be read using an 8-bit port on the PC/104 bus. The command and state register (CSR) is used to program the frequency setting word (FSW) of the numerically controlled oscillator (NCO) and to manage the interrupt request (IRQ).

Applied PCs

Fred Eady

Easy DOS it

DOS is dead? Not according to Fred! In fact, he thought DOS was the most feasible solution to his recent project. Fred put some diagnostic and service routines behind a touchscreen interface, and DOS made it easy.

Like you, I read to keep up with the changes in technology. It seems that every embedded piece I see has to include a mathematical equation that only the Almighty and a Pentium XXI running a Romulan RTOS could solve.

On the other side of that equation, most embedded hardware ads use the word "serious." You've read them: "Can be programmed in C for the serious embedded application."

Ok, define "serious." To me, any embedded application that you take your time and thoughts to write is serious.

To quote a programmer friend, "It doesn't matter what language you use or what hardware is involved. The object is to make the software and the hardware work together to perform the desired task." Yep.

I was recently given a task along with some embedded hardware and told to code and deploy a solution in a short development window with minimal cost. The task involved putting diagnostic and service routines behind a touchscreen interface.

Being embedded oriented, I'm used to not having the everyday items like keyboards and mice, but one of the tasks was to totally prepare a hard disk with the touch of an onscreen button. Have you ever tried to automate the DOS FDISK command? How about FORMAT?

If you answered yes to both questions and made it work, stop reading and turn the page. If you answered yes and did not succeed, answered no to either question, or are just plain interested in finding out how I did it, read on. (You can tell this is going to be a software piece by the "if-then" and "or" sentences you just read.)



Photo 1—Touch the screen and completely prepare a DOS drive. Imagine that!

IN THE BEGINNING...

There was DOS. Some famous guy said that DOS would be dead by now. Some other famous guy quipped that the world of computing and information technology will become paperless. Well, DOS is alive and well, and I still use a printer.

You know where I stand on those points and you've figured out that the OS will be DOS for this project. DOS says simple to many of you, but as I pointed out, automating simple isn't so simple.

One of the most important tasks to be performed is a single-touch FDISK and FORMAT operation resulting in a bootable MS-DOS 6.22 partition on a hard disk. I investigated various scripting programs that could probably have done the job.

The problem was that I had to perform this operation on various-sized hard disks, which meant writing and keeping up with multiple scripts. That's not so bad, but I only had the space of a high-density diskette to fit all of the routines and scripts into. Multiple anything was not a good idea.

Listing 1—All of the *includes* are provided with the PB/Vision package. Those that are not included are automatically generated in Workshop.

```

$INCLUDE "COMM.BI"           ' add comm routines
$INCLUDE "WINDOW.BI"        ' add window routines
$INCLUDE "FORM.BI"          ' add form routines
$INCLUDE "POPMENU.BI"       ' add popmenu routines
$INCLUDE "MOUSE.BI"         ' add mouse routines
$INCLUDE "EVENT.BI"         ' add event routines
$INCLUDE "RESOURCE.BI"      ' add resource routines
$INCLUDE ".\CCINK.BI"       ' add project file

$IF %HELPCODE
$INCLUDE "HELP.BI"          ' add help system
$ENDIF

DIM frmCCINKData AS SHARED frmCCINKTYPE 'frmCCINK
DIM frmCCINKHandle AS SHARED INTEGER
DIM pullHandle AS SHARED INTEGER
DIM statusHandle AS SHARED INTEGER
DIM resourceFile AS SHARED STRING
DIM HelpFile AS SHARED STRING
DIM CtrlBox AS SHARED MenuColorTYPE
DIM MenuColor AS SHARED MenuColorTYPE

$INCLUDE ".\CCINK.INC"      ' add user file

CCINK.INIT                  ' initialize the interface
CCINK.RUN                   ' run the program
CCINK.DONE                  ' shut down the interface
END

```

The FDISK/FORMAT utility also had to work with the programming language selected for this project. Because the OS is DOS, 16-bit applications will be the rule. The programming language and utilities have to be able to display a GUI with buttons and controls corresponding to *x* and *y* coordinates on the touchscreen.

My first thought was to use Borland's C++ Builder. Good choice for Windows, but overkill for a simple DOS application. My next choice was Microsoft's C++ in a 16-bit flavor. Same as Borland and still no "built-in" DOS GUI.

You C brethren out there, don't get me wrong. Both of the C packages I mentioned are capable and could be applied in this application, but I really didn't have the time to construct a suitable GUI from scratch with any language. My final and most desperate thought was to browse the *Circuit Cellar* Florida Room for some long-lost off-the-shelf application construction package I had previously used.

Aha! Bill's prelude to Visual Basic—Visual DOS. I'll bet the younger of you didn't even know such a thing existed. Although quite unsupported, Visual DOS was designed to give a DOS application the look and feel of the Windows environment.

Visual DOS was similar to the latest Visual Basic packages except it was designed for DOS. Like today's Visual Basic,

a form is presented and you put controls and buttons on the form and write event-driven code behind them.

Visual DOS may be the ticket. It supports its own graphical interface and a mouse. Mouse support is important because my touchscreen pretends to be a mouse.

After a few hours of remembering how to make it work, I put Visual DOS back on the shelf. The functionality was there, but the GUI was not "today." I wasn't satisfied with the grainy "yesterday" look of the buttons and screen. The end user deserved a better-looking interface. With this discovery, the project ground to a halt.

FALL BACK AND REGROUP

At this point, I felt I'd given this thing an initial stab with the thought of keeping everything as simple as possible. But the problem wasn't solved. I had to select the right components to complete the job and I had to do it quickly.

In addition to FDISK and FORMAT duties, I needed to:

- send test messages to two types of serial printers
- provide a message area for any errors
- provide printed logs and instructions

- execute various DOS-based utilities
- execute a touchscreen calibration utility

First, I needed a suitable programming-language package that included a good-looking GUI that can easily be ported to DOS or works natively in DOS. The logical selection would be something like Bill's Visual DOS product, but better. The application-language package has to open files, write and read the standard PC I/O devices, and allow external programs to execute or be called to execute from within the executing program.

After another search of the bookshelf, I came away with PowerBASIC 3.5. This product is standard BASIC with a twist.

PowerBASIC 3.5 has all the functionality of standard BASIC and then some. Although I won't need it here, PowerBASIC 3.5 does C things like bit arithmetic and inline assembler. Most importantly, PowerBASIC 3.5 compiles into a compact machine-language .EXE and that makes it suitable to embed.

PowerBASIC 3.5 is the answer for the serial printer issue and the external program execution parts, but there's still the problem of getting a GUI to front the code. I remembered that PowerBASIC 3.5 had some add-on products that just might meet the needs of this project.

You know, it pays to read. I keep all the paper that comes with software products for just this kind of moment. A bright yellow flyer in the PowerBASIC 3.5 box described all the stuff that can extend the usefulness and power of the PowerBASIC 3.5 package.

I originally purchased this package because it can be used with a companion product to construct standard DLLs from

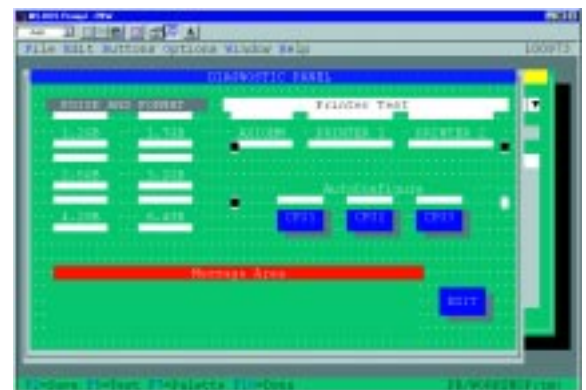
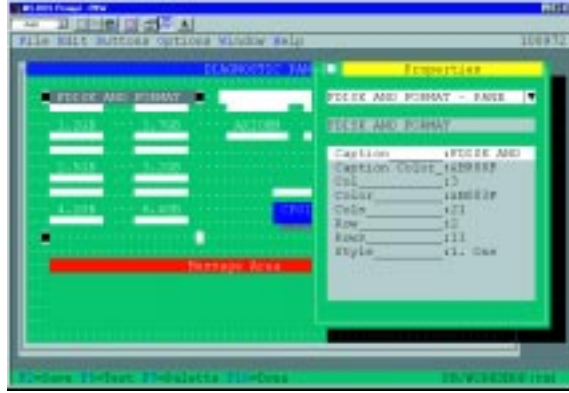


Photo 2—It's kind of rough looking, but it beats drawing all this from scratch.

Photo 3—
There aren't as many property choices as I'd like, but what's there works well.



Visual Basic 5 source. As the PowerBASIC 3.5 package is a native DOS product that can be used in a Windows environment, I read through the flyer hoping for a pot of gold at the end of the rainbow.

Apparently enough folks had this problem before me for a product to be offered. PB/Vision for DOS looked like the answer. The flyer read "Create visual text applications with windows, menus, buttons, and much more." Hmm....

Next morning, I assembled the PowerBASIC 3.5 toolset I needed to complete this project—PowerBASIC 3.5 Compiler for DOS and PB/Vision for DOS.

BUILDING THE DOS GUI

I had already experimented with "scripting" the FDISK and FORMAT commands with limited success. I could redirect the answers to the prompts via a file, but if anything bombed out in the meantime, there was no way to recover. In essence, the program would hang at the anomaly.

I decided to continue getting the main program frame assembled and solve the problems as they arose. A good-looking user interface was most important, so I went about it. Photo 1 is what the user sees.

Photo 2 is well past the starting point, but I think you get the idea. Just like VB5 and VB6, PB/Vision uses a form as the basis for the GUI. Photo 2 was built by placing controls and buttons on the initial form and naming them appropriately.

Photo 3 is an example of how the FDISK and FORMAT area was constructed. If you've ever done any of Bill's Visual stuff, this is similar. I was a little disappointed with the lack of "controls" that I could tweak to make the GUI behave like I wanted it to, but hey, this is DOS.

Although I wasn't happy with the way I made the GUI, I was happy with the professional look I ended up with. The GUI-under-DOS problem was solved.

PowerBASIC 3.5, like VB, provides a skeleton for the code based on the buttons

you put on the initial form. As you might imagine, PowerBASIC 3.5 isn't natively event driven or real-time oriented. In most applications, the touchscreen would be polled and x or y coordinates would be manipulated to determine which button was touched. The PB/Vision package does even better and provides an environment that mimics an event-driven system.

Three main modules make up the PB/Vision event-driven system: INIT, RUN, and DONE. Listing 1 shows all of the necessary run-time routines that are INCLUDED at the top of the compile phase.

Right below the INCLUDEs is a line that enables you to include a HELP system. Pretty strong stuff compared to the BASIC I learned in the Radio Shack showroom.

Moving on down the listing, you see where the SHARED resources are declared. These are pretty obvious as to what they are. Following the SHARES area is a line-include file that is application specific. The CCINK.INC include file in Listing 2 determines what the GUI looks like graphically.

The three main modules (actually sub-routines) I described earlier are declared following the CCINK include file. It's interesting that the PowerBASIC 3.5 developers enable you to manually edit any file in the source chain.

Although there are warnings all along the way about changing something here that affects something there, change is still allowed. I figure that's for you brave types. For us scared guys and gals that work under tight schedules and high pressures, the PB/Vision utility Workshop (included in the PB/Vision package) provides all of the switches programmatically that you can select manually. Guess which way I went.

Listing 2—This file is used as food for the appinit function.

```
APP.ATTR = &H8F
APP.PATTERN = 32
APP.ROWS = 25
APP.MENUATTR = &H0070
APP.STATUSATTR = &H0030

IF LEN(ENVIRON$("windir")) THEN
  APP.GRAPHICSMODE = 0
  APP.GRAPHICSMOUSE = 0
ELSE
  APP.GRAPHICSMODE = 1
  APP.GRAPHICSMOUSE = 1
END IF

'APPTITLE &H1F, "CCINK.BAS (Edit CCINK.INC to customize desktop)"

CtrlBox.kolor = &H7170
CtrlBox.borderattr = &H70
CtrlBox.titleattr = &H2E
CtrlBox.highlight = &H0B03
CtrlBox.sepbar = &H7B
CtrlBox.border = 1
CtrlBox.Flags = %DRAGBAR OR %CONTROL OR %SHADOW

$IF %HELPCODE
  Help.rows = 12
  Help.cols = 70
  Help.kolor = &H7F70
  Help.border = 1
  Help.borderKolor = &H70
  Help.Title$ = "Edit CCINK.INC to change title"
  Help.titleKolor = &H9F
  Help.textKolor = &H7170
  Help.style = 1
  Help.buttonStyle = 3
  Help.buttonKolor = &H7F70
  Help.buttonHighlight = &H707F?
  Help.flags = %SHADOW OR %DRAGBAR
$ENDIF
```

The INIT subroutine in Listing 3 takes care of initializing the environment. This includes finding a mouse, determining the type of memory support available, installing the necessary button driver code, and loading the main GUI. A function call to APPINIT uses the APP variable parameters found at the beginning of Listing 2 (APP.XXXXXX) to help start things up.

The GUI is in front of the user at this point and it would be nice to have something happen when the screen is touched. The RUN subroutine is responsible for this. As you see in Listing 4, there's not much to this routine except for the GETEVENT call.

I was disappointed that I couldn't get more in-depth information about GETEVENT. This call does a bunch of stuff under the covers. The same goes for APPINIT.

Anyway, GETEVENT is the means by which the event-driven mechanism is fueled. The program loops here, waiting for an event to occur. When it does, the GETEVENT routine provides user feedback by performing a "virtual button move" on the

screen. Then, by means unknown to me, it calls the function in Listing 5.

Here's where all the user code is inserted for each button defined on the GUI. I included separate BASIC modules for each instance of a button. It's not important here as to what's in each module because I'm pretty sure you grasp the logic of the idea.

I will show you the contents of the FD and FDB files as they are what drive the automated FDISK and FORMAT engines. I went ahead and included the DONE routine in Listing 4, because there isn't much to say about it except that it contains a call to APPCLOSE.

That's it. Use Workshop to create a GUI and use PowerBASIC 3.5 to add the code and compile a nifty-looking DOS application. Compared to working with its Windows counterpart, the PB/Vision product is somewhat limited. The redeeming factor is that although there are limitations, the PB/Vision-PowerBASIC 3.5 combo is a good programming tool for embedding DOS applications.

Listing 3—It sure would be nice to know what *appinit* really does here.

```
SUB CCINK.INIT                                ' program initialization code
  winUseUMB = 1                                ' add upper memory support
  IF BIT(pbvHost, 5) THEN                      ' test if in IDE
    resourceFile$ = "CCINK.RES"
    HelpFile$ = "CCINK.HLP"
  ELSE
    resourceFile$ = AppName$                   ' program is compiled, get name of EXE
    HelpFile$ = AppName$
  END IF
  APPINIT
  gottaMouse% = MOUSEINIT(mouseButtons%)
  MOUSECURSORON
  REGCOMMANDBUTTON                            ' install driver
  REGPANELBUTTON                              ' install driver
  frmCCINKHandle% = FRMCCINK.INIT             ' Load "frmCCINK" object
END SUB
```

Listing 4—Although it doesn't look like much, the *getevent* call is pretty powerful.

```
SUB CCINK.RUN                                ' program execution loop
  DO
    eventNo% = GETEVENT(0)
    SELECT CASE eventNo%
      CASE 102
        EXIT LOOP
      CASE 103
        $IF %HELPCODE
          HELPSHOW HelpContext%, HelpFile$, Help
        $ENDIF
      CASE ELSE
        END SELECT
    LOOP
  END SUB

SUB CCINK.DONE PUBLIC                          ' program termination code
  MOUSECURSOROFF
  APPCLOSE
  END
END SUB
```

AUTOMATING FDISK AND FORMAT

We're down to the big one. I searched all over the Internet attempting to find someone smarter than me who had figured out a way to do the FDISK/FORMAT thing without a keyboard. I consulted with Mr. Norton and a Ghost. I even studied Unix scripts hoping to port some knowledge to my little DOS app.

Finally, as I was reading through a thread on formatting, some Joe shouted, "It's all in PartitionMagic from PowerQuest." Duh. I have that. For those of you scratching your head and mumbling "Who the heck is PowerQuest?" you may know them for products like DriveCopy and Drivelmage.

Well, after I picked myself up off the floor, I pulled out PartitionMagic and looked at its feature set. Seems that the Enterprise version comes with a scripting capability.

This package does everything, including creating, deleting, moving, and resizing FAT, NTFS, and HPFS partitions. Some of these commands sound like FDISK and FORMAT to me. I'm on the way, but first I've got some housekeeping to do.

I needed a way to determine if a DOS partition was already on the target drive. In the beginning of the diagnostic program, I attempt to read and write the active partition. Depending on the results, I set a DOS environment flag as Y or N.

When the FDISK/FORMAT button is touched, I read the environment flag into a string variable and determine how I want to initialize the target drive. I put some fancy user output around the operation, but I know you're only interested in the facts shown in Listing 6.

The code at the top of Listing 6 is the PowerBASIC 3.5 code that invokes PartitionMagic via the command line. N in the DOS environment variable signifies that a new or unpartitioned drive is to be prepared. An environment variable with the value Y is already partitioned and possibly formatted.

The error log and result files give the user some feedback as to what is happening with the FDISK/FORMAT process. Also, the log and result files are used as logical flags to help determine if the newly partitioned and formatted drive can be made bootable via the diagnostic program.

Let's look at the TXT files called in the PartitionMagic command line. OLDFD.TXT prepares drives that are determined

Listing 5—This approach isn't as elegant as Bill's, but it serves the purpose.

```
FUNCTION FRMCCINK.ROUTINE% (BYVAL handle%, BYVAL eventNo%, BYVAL
  parm1%, BYVAL parm2%) PUBLIC

  SELECT CASE eventNo%
    CASE 101          ' <CR>
    CASE 102          ' <ESC>
    CASE %cmaxiohmclick
      $INCLUDE "C:\POFILES\AXIOHM.BAS"
    CASE %cmprt1click
      $INCLUDE "C:\POFILES\PRIN1.BAS"
    CASE %cmprt2click
      $INCLUDE "C:\POFILES\PRIN2.BAS"
    CASE %cmsetcnfg1
      $INCLUDE "C:\POFILES\SET1.BAS"
    CASE %cmsetcnfg2
      $INCLUDE "C:\POFILES\SET2.BAS"
    CASE %cmsetcnfg3
      $INCLUDE "C:\POFILES\SET3.BAS"
    CASE %cmdexit
      close
      system
    CASE %cmfd12
      FD
    CASE %cmfd17
      FD
    CASE %cmfd25
      FDB
    CASE %cmfd32
      FDB
    CASE %cmfd42
      FDB
    CASE %cmfd64
      FDB
    CASE ELSE
  END SELECT
  FRMCCINK.ROUTINE% = eventNo%
END FUNCTION
```

Listing 6—The execute commands turn control totally over to PartitionMagic. This helped conserve precious DOS memory.

```
*** CODE SNIPPET FROM FD.BAS
if f$ = "N" then
  execute "A:\PQMAGICE /CMD=NEWFD.TXT /LOG=RESULTS.FIL /
  ERR=ERROR.FIL"
end if

if f$ = "Y" then
  execute "A:\PQMAGICE /CMD=OLDFD.TXT /LOG=RESULTS.FIL /
  ERR=ERROR.FIL "
end if

*** OLDFD.TXT
Select Drive 1
Select Partition 1
Check
Label /SetLabel1=""
Delete "NO NAME"
Create /FS=FAT
Format "NO NAME" /FS=FAT
Set Active

*** NEWFD.TXT
Select Drive 1
Select Partition 1
Create /FS=FAT
Format "NO NAME" /FS=FAT
Set Active

*** FOR DRIVES > 2 GB
Create /FS=FAT /SIZE=2040
```

to be partitioned and formatted. The first line of this script selects what is equivalent to the C: drive. The next step selects the first (and in this case, the only) partition.

Check is a PartitionMagic command that checks the drive for errors. If the drive doesn't pass this step, the PartitionMagic program ends and returns an error code.

Remember, in Bill's FDISK utility you must enter the partition name to delete the partition. That's one of the keyboard-needed gotchas I was contending with before finding out I could use PartitionMagic.

Using Label, I can set the partition label name to anything I desire. Here, I nulled it out so the next command (a Delete Partition with a NO NAME flag) could be issued and completed successfully.

After the old partition is deleted, a new partition can be built using Create Partition. The /FS=FAT flag tells PartitionMagic to make a standard FAT partition. Create assumes you want to use the entire drive unless you enter the /SIZE parameter as shown at the bottom of Listing 6. For drives greater than 2 GB, I used /SIZE to ensure that the drive size didn't exceed the DOS limits.

The final step is to issue FORMAT Partition and set the newly created and formatted partition active. At this point, PartitionMagic automatically reboots the embedded PC and the AUTOEXEC.BAT file inspects the error log and results file. If all is well, a DOS SYS command is issued and Bill's COMMAND.COM and a couple hidden files are transferred to the new partition.

A simple BOOTPC executable is invoked to return the user to the GUI for another operation. As you see in Listing 6, the only difference in the OLDFD and NEWFD script files are the commands and flags needed to negate the partition label.

WHAT HARDWARE?

That's how it's done—just like you'd do it with a keyboard answering the prompts. This solution can be ported to just about any embedded platform that runs DOS. In fact, the only hurdles I had to overcome were DOS induced.

There's no reason this concept can't be implemented with any other programming language with or without a GUI. The idea was to introduce you to another aspect of embedded design using software objects

that aren't often thought of as "embeddable."

By the way, the diagnostic package size, including all of the modules and PartitionMagic, was a little less than 1.18 MB. What started out as complicated, is now "serious" and 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.

SOURCES

PowerBASIC 3.5, PB/Vision for DOS

PowerBASIC, Inc.
(813) 659-8000
Fax: (831) 659-8008
www.powerbasic.com

PartitionMagic

PowerQuest Corp.
(801) 437-8900
Fax: (801) 226-8941
www.powerquest.com

Touchscreen

MicroTouch Systems, Inc.
(978) 659-9000
Fax: (978) 659-9105
www.microtouch.com

FEATURE ARTICLE

Michael Bading

Low-Cost Data Acquisition System

Michael uses a single-board 8-bit computer complete with its own OS and BASIC interpreter to create a data-acquisition system for under \$1000. He did it in three months, too. And the best part is the one it lacks: no PC required.



I was working late one Friday when the phone rang. It was my friend Paul, a design engineer in the solar division of the local power company. He was asking me to build a data-acquisition system for him.

“Sam Easley’s lights went out about half an hour ago while he was sitting down to dinner and he was steamed! I’m tired of getting these calls. I’ve got to do something about it.”

It turns out the company was investing heavily in a solar resources program. Part of the program was supporting off-grid customers (e.g., ranchers like Sam Easley) by providing them with solar power and servicing their systems.

Because the program was fairly new, it also involved working with customers who already had solar systems. Each system was different so it was quite a challenge to diagnose problems among the various systems.

“I need some type of data-gathering system that can be easily configured to diagnose each customer’s system without having to send a programmer into the field every time one of my field technicians goes on a service call,” explained Paul.

“And I need it quick. It has to be in production in less than three months. We need to be able to

diagnose where the energy losses are and how to improve each system. I have some room in the budget for developing a data-acquisition system, but the retail price has to be less than a \$1000 and it needs to work without a PC.”

I held my breath and listened. I wouldn’t mind having a system like that for myself, I thought. But just about every data-acquisition system I’d seen required some programming, none were less than \$2000, and most of them required a PC to operate.

After getting off the phone, I wondered whether I could design a system that would take accurate data, was easily configurable, and was still inexpensive. I also thought about how quickly he wanted the system developed and what the best approach to the problem would be.

SOLAR SYSTEM BASICS

I considered the typical parts of a modern solar power system and what data had to be gathered to solve Paul’s problem of determining what and where the losses are in each system. At the heart of each solar electrical system are the four basic components shown in Figure 1—a solar array, a charge controller, a battery bank, and some type of inverter to convert the DC power to usable AC electricity.

He also wanted to be able to monitor each customer’s major electrical devices. If no problems could be found with their solar electrical system, he could tell them which device or appliance was causing the outages.

Any first-year engineering student knows that before you can determine the losses in a system, you have to know what’s going into it. The fact that a photovoltaic (PV) array’s volt-

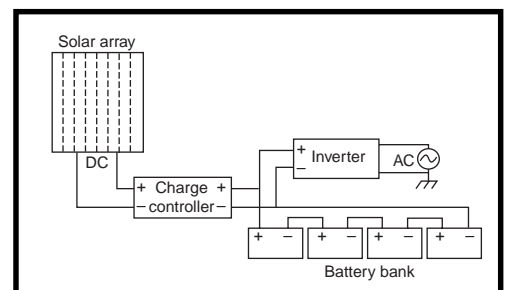


Figure 1—In a basic solar energy system, energy is converted from sunlight to electricity by the solar array and is then stored in the battery bank for later use to be converted directly to AC.

age and current varies throughout the day (as well as from day to day) makes loss assessment difficult.

The importance of getting an accurate base of what's being generated is directly related to the accuracy of assessing the losses. Problem is, there are nearly as many configurations and sizes of PV arrays as there are people. Also, the voltage and current measurements for the PV array needed to be easily scalable to work with arrays ranging from 300 W to 5 kW.

To determine the losses of each component, current and voltage measurements are needed before and after each system component. Data has to be gathered at regular intervals to determine where in the system the problems may be.

For the front end of the solar-powered system, all of the voltage and current measurements would be DC because only the final stage (the inverter) would be AC. If no problems were found with the solar electrical system, the remaining sensor inputs would measure AC voltage and current at each major appliance.

KEEPING IT SIMPLE

I know data-acquisition systems all require some programming as well as knowledge of conversion from digital values to a real-world measurement or a shunt resistor to convert current to a measurable voltage. I also knew any existing user-friendly system was at least double Paul's ballpark budget (just for the basic model without options).

How was I going to design a system that could be used by someone with a limited technical background and keep it under a grand? What exactly is a data-acquisition system, and why are so many data-acquisition systems so complex? Ultimately, I decided that only four parts are needed for any data-acquisition system like the one diagrammed in Figure 2:

- a sensor or set of sensors
- a measurement device
- a way of switching or multiplexing between sensors and the measurement device

- a means of storing the measured values

Next, I began to throw out what I knew wouldn't work. The application engine needed to be a preexisting device that was available at a low cost and had enough computing power to handle the job.

An 8-bit controller would be more cost effective than a 16-bit one. But if I went the 8-bit route, it needed to have the accuracy and speed necessary to manipulate the data and still perform math functions.

Because assembly-language programming and debugging is usually slow and arduous, coding in a higher level language was a must, which meant I had to use a controller with a robust set of peripherals and a developed set of routines. The peripheral set, at a minimum, had to include a real-time clock, two RS-232 serial ports, and a 12-bit ADC onboard.

STARTING A FIRE

With the hundreds of controllers on the market, how would I find such a device? A single-board 8-bit computer complete with its own OS and a BASIC interpreter, the Firecard in Photo 1 was an easy choice to start with (see Figure 3). But would it be right for this application? How much additional hardware and software were necessary?

The Firecard 24/20 is a 24-MHz 8-bit Z180-based SBC system with a

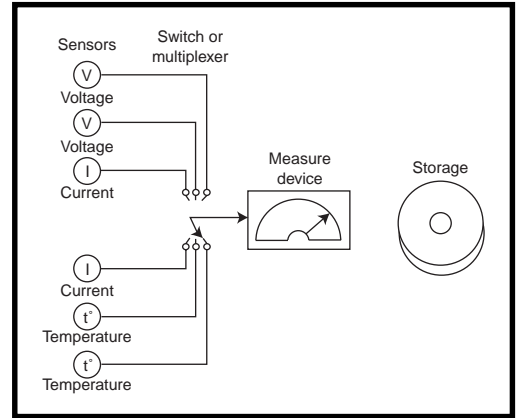


Figure 2—In a basic data-acquisition system, sensors convert physical properties into electrical signals, a switch or multiplexer then determines which signal is measured by the device (ADC), and the corresponding value is stored in memory or some other digital medium.

disk operating system (Fire-DOS) and a BASIC interpreter (Fire-BASIC). The system software takes up 2.5 KB of memory, boots in 2–3 s, and has fast program execution (typically several thousand lines of BASIC code per second). All other programs supplied with the Firecard are less than 64 KB.

The command set is reliable and easy to learn. Because the programming language is interpreted BASIC, a compiler isn't needed and debugging is fast.

The hardware includes 1-MB fixed-flash memory for data and program storage, 64-KB SRAM, and 8-KB EEPROM. It has two asynchronous serial ports with RS-232 drivers and clocked serial output port for printer.

It also features an 8-channel 12-bit ADC and two PWM 8-bit DACs with buffers. It has a real-time clock and 32 bytes of RAM with battery backup.

As well, the I/O configuration is customizable through a Xilinx FPGA.

With the 1-MB fixed-flash memory set up as a hard disk, the 1-MB removable flash memory set up as a bootable floppy, and a clocked serial port configured as a printer interface, the system architecture is transparent to the user and can be viewed almost like a PC. The system calls to the peripherals are straightforward.

The Firecard's edge connector can be inserted into a 30-pin SIMM socket to allow external access to the system peripherals.

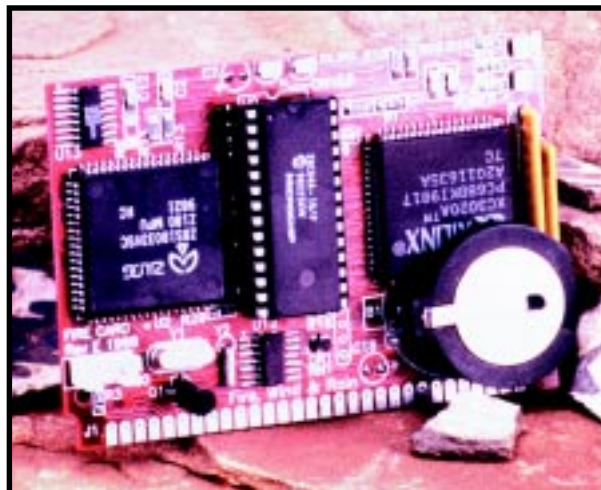


Photo 1—The Firecard 24-MHz single-board computer offers lots of power for many applications.

INITIAL PARAMETERS

Paul and I agreed on some initial parameters. The data-acquisition system should be able to measure up to 48 channels at a speed of at least once per second or greater. The system should be able to measure DC voltage up to 100 V (both single-ended and common-mode), DC current, AC voltage (peak and RMS), AC current, temperature, and contact closures.

Next, I had to come up with a plan for designing the data-acquisition system, what hardware would be used, and how to make the user interface and system setup and configuration easy to understand. Last but not least, I had to provide something inherent to all authentic data-acquisition systems—calibration.

I presented the requirements to my design team and advised them to keep one idea in mind—simplicity. I showed them my data-acquisition block diagram and instructed them that any changes that increased the complexity of the system would have to either directly make the system easier to use or help reduce cost.

With low cost as a key requirement, we couldn't use all the fancy goodies out there. We had to focus on the basics. Yet, at the same time, it had to be an actual data-acquisition system—not a PC card that pretends to perform data-acquisition.

Out of the initial brainstorming session, four concepts developed that would eventually form the basis of our system, which we call the DAS-100.

The first of these concepts involved a simple user interface in which all

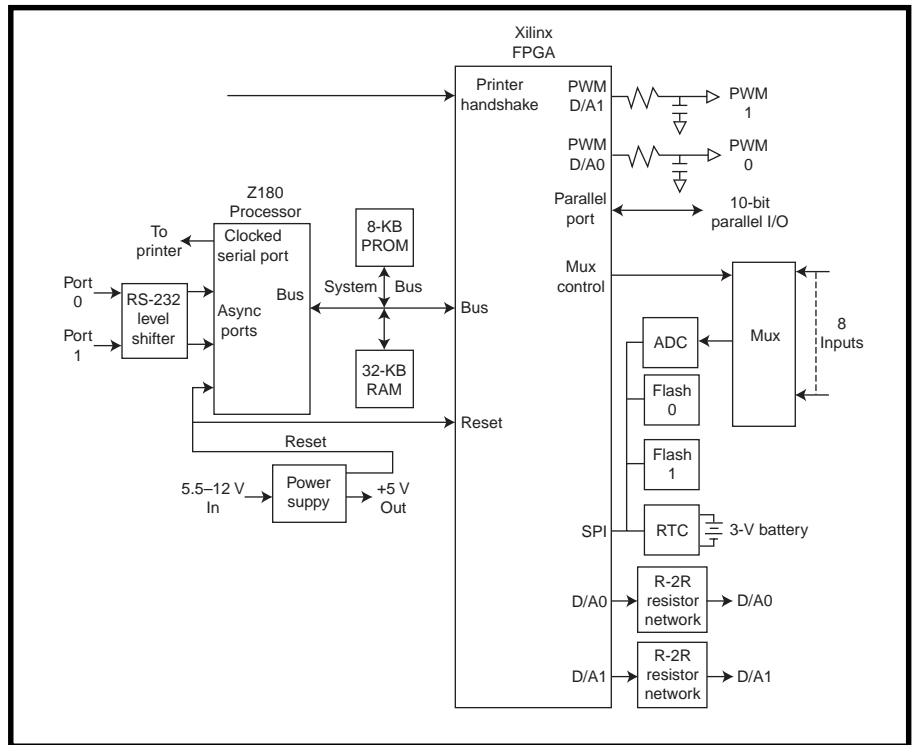


Figure 3—Here you see the block diagram of the Firecard. The Xilinx FPGA handles the addressing of all onboard peripherals and is easily reconfigurable at bootup.

data calculations and conversion are automatically performed by the system. System setup was menu driven. The user would only be prompted for card information (which cards were installed in the system), which channels would be used for which measurements (including what units the information should be displayed in), and two simple calibration values (gain and offset), if needed, for each sensor. During actual data gathering, a 16 × 4 LCD with four menu buttons would be used to real-time monitor the data being collected.

To perform measurements with a large common-mode voltage, we needed

an isolated ADC separate from the Firecard. Although the Firecard has an ADC built in, its measurement could only be referenced to ground. Also, the system might be required to measure subtle differences at an elevated voltage so a differential input was needed to measure small voltage changes between inputs.

One example of this would be the need to measure small changes of 10–20 mV from a shunt resistor (to measure current) on a 60-V PV array. If we were unable to use a common-mode method of measurement, the resolution would be quite large.

The third concept was to attenuate the data signal coming in and then use a programmable gain amplifier (PGA). This arrangement would inherently protect the system from being damaged in an overvoltage situation and would give the widest range of data scaling at the lowest cost.

The final concept was the minibus. Because the Firecard already had an edge connector that enabled external peripheral access, the minibus was developed to allow the greatest compatibility with the Firecard yet still lend itself toward the specific application at hand.

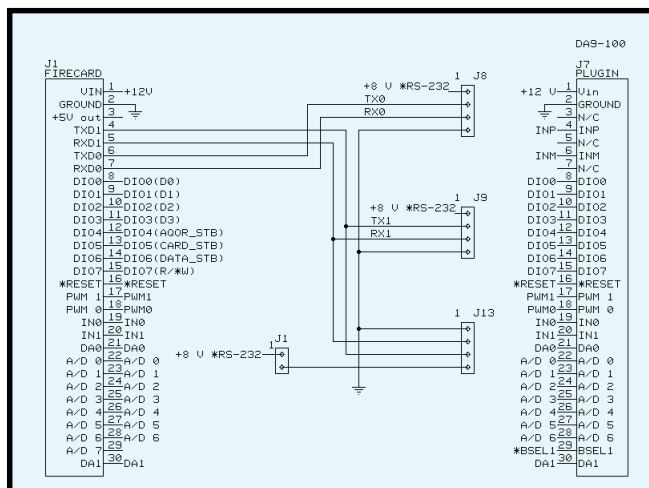


Figure 4—The DAS-100's minibus is largely based on the Firecard's edge connector pinout. This sped up both hardware and software development for the project.

Photo 2—By using current sensors with toroid cores with Hall-effect devices and RJ-11 connections, shunt resistors and circuitry are not needed. Installation of the sensor is a breeze.



RIDING THE MINIBUS

In Figure 4, compare the Firecard pinout with the diagram of the minibus. They're similar, but with some important changes. The Firecard's RS-232 signals (pins 4–7) were routed to a user interface on the motherboard, but on all the other cards in the system, these pins were used for either the single-ended or differential analog inputs to the measurement device, or not used at all.

Digital signals are routed to one of the two digital input bits (pins 19–20). The 8-bit digital I/O bus (pins 8–15) were broken up into a 4-bit Address/Data bus, three strobes, and a Read/Write signal. All other signals were left unchanged so they would be directly compatible with the Firecard's connector pinout.

Although the 4-bit bus permits addressing for up to 16 input option cards, only eight devices are addressed. This setup leaves room for expansion

in possible future designs or room for more complex addressing schemes.

Each device, including the measurement device (consisting of the attenuator, PGA, and an isolated 12-bit ADC), has an individual address and is accessed through the 4-bit bus and three active-low strobes—card, address, and data.

Addressing a particular channel on an option card and acquiring data from that channel into the 12-bit ADC is a five-part process. First, the card strobe is held low and a 4-bit address is sent and decoded on the motherboard to the corresponding option slot.

After the correct card is selected, the data or address strobe (along with

a 4-bit code) is used to select the desired channel. Once the channel is selected, it's inherently connected to the 12-bit ADC by means of multiplexing or relays.

For the third step, the ADC is signaled to begin sampling. Once the sample is made, the data is mathematically adjusted (based on the type of measurement and units desired) and stored.

Finally, the data channel is deselected. The Firecard coordinates all the timing of the measurement process, performs the calculations, and stores the data.

The PGA and attenuator pair present a flexible method for preconditioning the signal before it's sampled by the ADC. The PGA circuit in Figure 5 offers two modes to permit either unipolar or bipolar measurements by the ADC. For unipolar measurements, the incoming signal is referenced to ground. For bipolar measurements, the incoming signal is reference to midscale of the ADC or +2.5 V.

The PGA also has four scales— $\times 1$, $\times 10$, $\times 100$, and $\times 1000$. Along with the divide-by-100 attenuator, it enables the 12-bit ADC to measure incoming voltages from a low range of 0–0.5 V to a high range of 0–500 V, with each

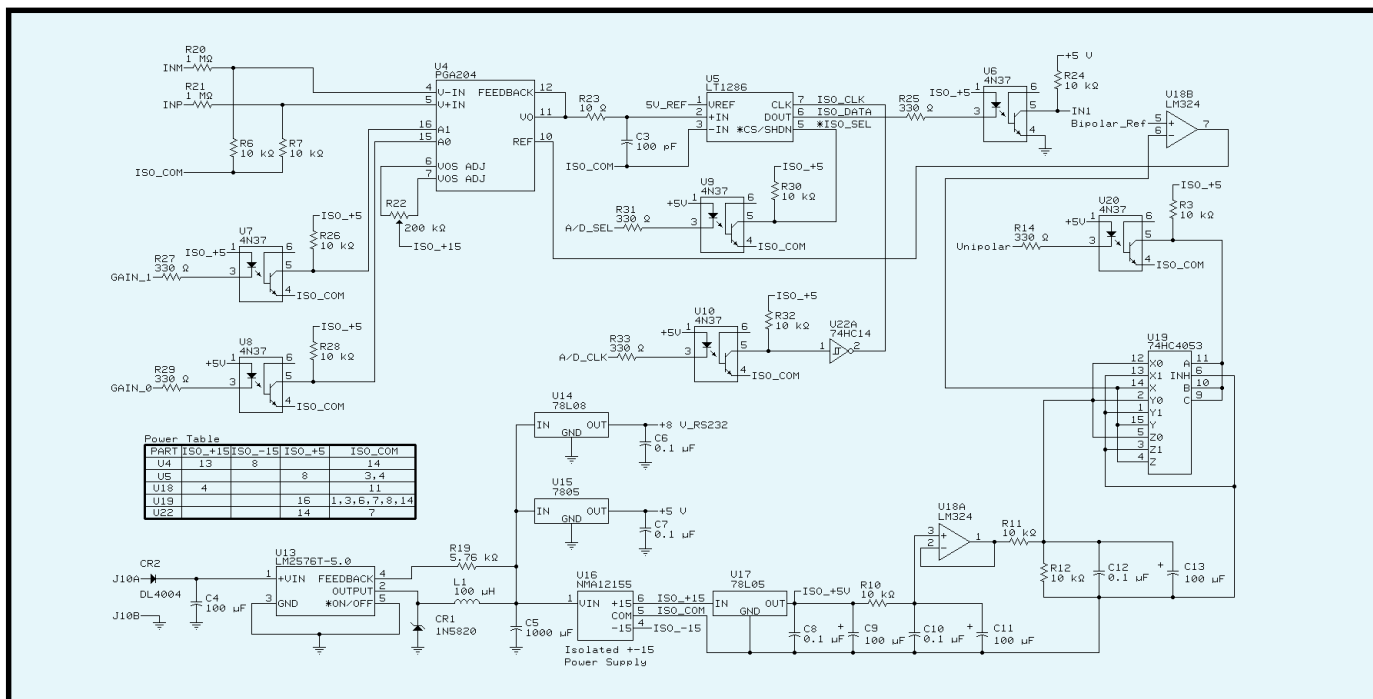


Figure 5—This circuit accommodates both unipolar and bipolar differential measurements in four different ranges. The divide-by-100 attenuator before the PGA protects the circuit from being overdriven.

range having 12-bit resolution (corresponding to 4096 steps). Even though the input voltage can theoretically go to 500 V, for safety reasons we decided to specify only 100 V.

CHOOSING OPTIONS

Now that a method of measuring signals is established, let's focus on the types of data to be measured. The team came up with two types of option cards that can be used in two different configurations. Because we agreed to only address eight system devices, one of which was the measurement device (address 000, device 0), that left seven devices that we could still address.

Six option slots (addressed 001–110, or devices 1–6) on the motherboard permit the system to be configured and changed, depending on the option card used. Two input cards (DC and AC) and one sensor card were created to handle the bulk of measurements used in a typical application.

The final device group (consisting of three relays) was added to the motherboard (address 111, device 7). Through software, these relays can be used to control external components while data is being acquired. A simple latch (74HC174) and a transistor control the coil of each relay.

Because the system's measurement device is set up to measure voltage in a variety of ranges, the DC input card design in Figure 6 consists of a latch and relays that select the channel to be measured. For measuring DC voltages, no additional hardware is needed. DC current measurement is converted to a voltage using either Hall-effect DC current sensors or shunt resistors.

The simplicity of the DC input card design results in a low production cost. This card can also be used as a relay-switching card to expand the capability of controlling external devices beyond the three relays included on the motherboard.

For AC measurements, an AC sensor interface—in addition to the AC input card—allows the measurements to be digitized at the source, solving several problems. Sensors can be placed farther away from the data-acquisition unit and noise immunity increases.



Photo 3—The completed DAS-100 in this waterproof enclosure is ready for action!

serially through another multiplexer on the AC input card and read into the Firecard through the digital input pins on the minibus. This card can also be used purely as a digital input card because the design is identical to what would be needed to read digital inputs.

RELATING TO THE REAL WORLD

All connections are done with four-conductor RJ-11 connectors (phone plugs) and have the identical pinout (see Photo 2). The only requirement for any of the sensors is that their measurements are converted to a DC voltage or digital value. Because the DC voltage measurements require no special interface, the isolated ADC in the system is already

The AC input card is a digital input card that is designed to work in conjunction with the AC sensor interface. Only an 8-bit latch and buffers are needed to select the channel.

Data from the AC sensor interface card (see Figure 7) is transmitted back

set up to evaluate a wide range of single-ended and differential voltages.

DC current measurements are made using a Hall-effect DC current sensor or shunt resistor. Using Hall-effect DC current sensors enables current measurements to be read as a voltage with no need for shunt resistors. Although shunt resistors can give accurate readings of current, they are usually more difficult to set up and aren't recommended because of the inherent IR losses associated with them. Sensor calibration is performed once during system setup via software.

AC measurements are made with either Hall-effect current sensors (to measure AC current) or transformers (to measure AC voltage). They are then fed into the AC sensor interface, which is housed in a box near where the measurements are made. Both of these sensors output a voltage that is offset at 2.5 V to achieve a true bipolar measurement of the AC. Again, these sensors are calibrated during system setup.

The AC sensor interface measures two channels of AC (usually voltage

and current) and transmits a digital signal back to the AC input card. The AC sensor interface in Figure 8 consists of an analog multiplexer, a 12-bit ADC, and two 68HC705 microcontrollers.

The front-end microcontroller switches the analog multiplexer between the two AC input channels and clocks the ADC to allow 50x oversampling of each channel for a typical 60-Hz AC signal. It also reads the

corresponding digital value, and transmits each value to the back-end micro.

The back-end microcontroller then evaluates the peak and sum-of-squares value (used to calculate RMS) of each channel read and converted by the front-end microcontroller over the data cycle (set by the data-acquisition unit).

An LM335 precision temperature sensor makes temperature measurements that can be directly calibrated in kelvin. Operating as a two-terminal zener, the LM335 devices have a break-down voltage directly proportional to absolute temperature at +10 mV per kelvin. This linear relationship corresponds to 2.73 V at 0°C. The LM335 operates from -40° to +100°C.

SYSTEM SOFTWARE

The system software has three parts—configuration and calibration, the data-acquisition run-time software, and the user-interface software.

System configuration and calibration consists of a menu-driven user interface. During setup, the user is prompted to enter the type of card in each option slot, which channels are used, calibration values of each sensor, and the units of measure to be displayed.

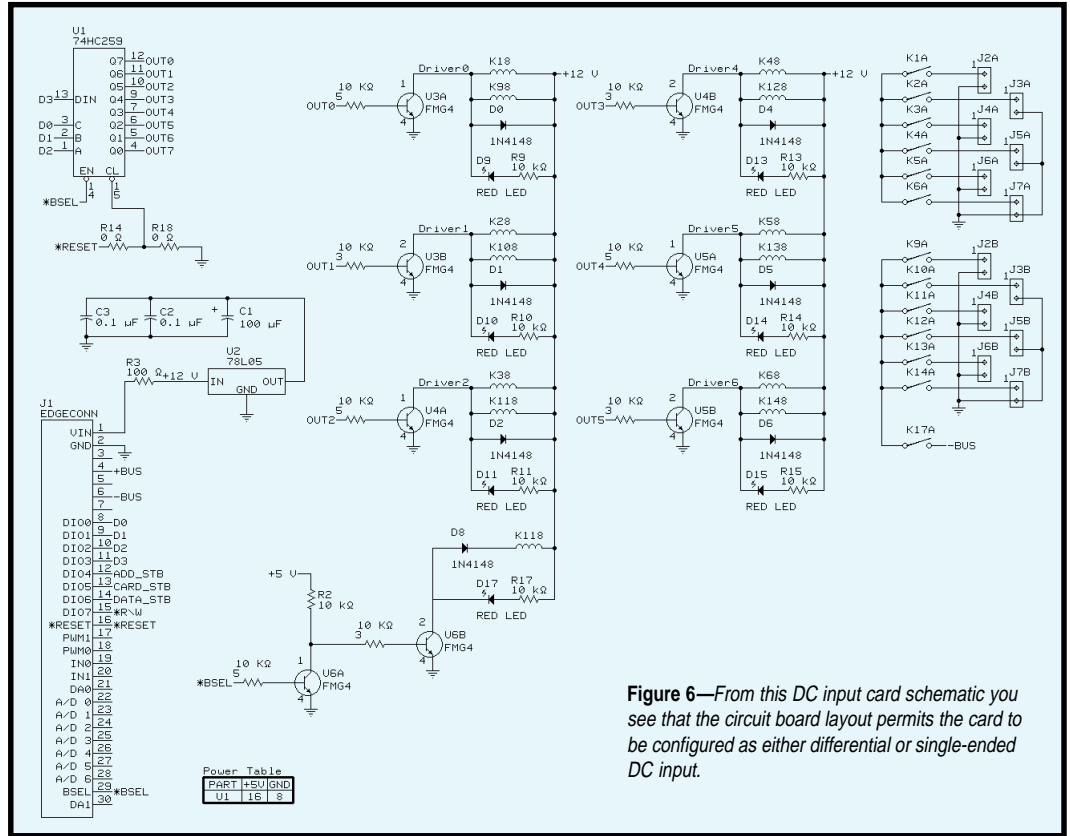


Figure 6—From this DC input card schematic you see that the circuit board layout permits the card to be configured as either differential or single-ended DC input.

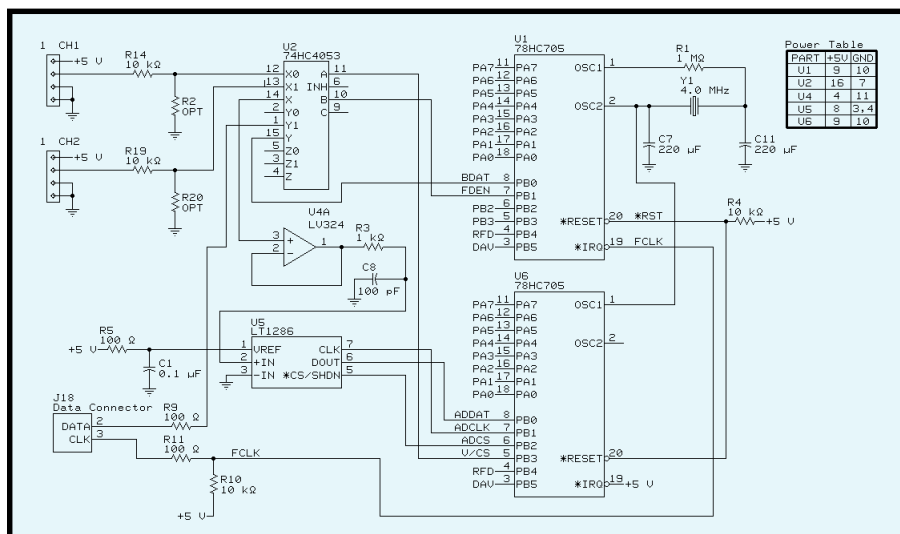


Figure 7—By using two 68HC107J1A processors and an LT1268 12-bit ADC, the digital conversion as well as the V_{p-p} and V_{RMS} calculations are performed before being sent to the AC input card.

All sensors are factory-tested and the calibration values (gain and offset) are stamped on the side of each sensor (if needed). During setup, the user is prompted for each value. Once these values are entered, calibration is complete. The user can also set up as many as 10 different screens on the LCD.

The run-time routines work in a simple polling method for each channel. A look-up table of the card type in each slot and what is to be measured from each channel determines how the data is to be scaled and processed.

Scaling is based on the type of units the user selects during setup. Typical data processing may include the average value, minimum value, maximum value, and temperature.

The AC sensor interface calculates the peak and sum-of-squares values, so only the square-root function is needed to calculate the V_{RMS} of the AC measurements. Once the data is processed, it is saved in an array until it is logged.

The user interface can be approached as a menu-driven program with a PC or as a screen-driven program with the 16×4 LCD and four menu buttons (no PC required) included with the system. The LCD panel displays up to 10 screens, each showing two parameters that monitor while the system is operating.

ON THE PATH TO A QUIET DINNER

Almost three months have passed since Paul told me about his problem, and now the DAS-100, shown in Photo 3, is about ready to go into production. When I first heard about Paul's dilemma, I didn't think a system like the DAS-100 could be built with such a short development time and still fulfill all of the requirements.

Before too long, Sam Easley may be able to sit down to a quiet dinner on a Friday night without his lights going out, thanks to the DAS-100. Unfortunately, it can't stop the telemarketers from calling during his meal. ☹

Michael Bading is the senior applications engineer at Fire Wind & Rain Technologies. During his 12 years in the computer and electronics industry he has worked with embedded controller design, embedded systems program-

ming, PC peripheral development, and semiconductor testing. You may reach him at mbading@firewindandrain.com.

SOFTWARE

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

SOURCES

Firecard

Fire Wind & Rain Technologies LLC
(800) 588-9816

(520) 526-1133
Fax: (520) 527-4664
www.firewindandrain.com

68HC705

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

LM335

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

Turn the Page

FEATURE ARTICLE

Ingo Cyliax

Pager Technology in Embedded Control

Low-bandwidth one-way communication that's wireless and cost effective is the big reason Ingo sees pagers being applied to embedded control applications. Tune in as he uses pager technology to monitor and send alarm conditions to pagers.



think it's safe to say that everyone's comfortable with the person-to-person aspect of pager technology. Some of us, though, wish that pagers and cell phones were never invented.

However, there are some interesting applications, like pager-based news services. Both ESPN and CNN offer a subscription service that sends news blurbs to your alphanumeric pager.

There are also communication applications that don't involve people, such as a TV program listing service. The paging receiver is embedded in a set-top box, and the service sends listings to be stored in the box. When the viewer wants to see the TV listing, the information is recalled and displayed.

I can already envision other uses. You could embed a paging receiver in

a remote controller. To activate or switch something, you'd send a code to the device. In an agricultural application, that device could control a pump for an irrigation system.

Those are just a few services that could implement a paging receiver in an embedded system. But, consider an application that involves the sending end of the paging service. The application involves monitoring and sending alarm conditions to pagers—obviously an embedded paging application. The thing is, there aren't a whole lot of embedded solutions out there.

I became involved with the application of alarm and monitoring systems a few years ago when the Internet first started to take off. I ended up consulting for several local Internet service providers (ISPs).

One common problem is that servers and network devices sometimes crash or hang, usually at odd hours. Because ISPs offer service 24-7, long downtimes mean losing accounts. And, most small ISPs can't afford to have operators on duty 24 hours a day.

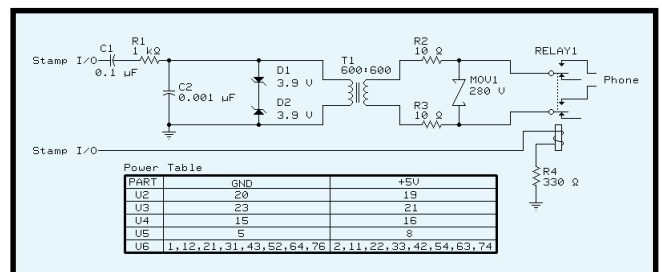
The standard solution is to have software that monitors critical services—for example, a script that tries to retrieve web pages from a server.

When the script detects a failure of a service (usually when it times out), it sends an e-mail to an address that gateways the e-mail to a paging service. If the operator on call has an alphanumeric pager, messages like "server www.superhot.com not responding" or "UPS on battery backup" can be sent.

This arrangement works fine most of the time. Also, tools that gateway e-mail to pagers are easily obtainable for some OSs. They're even freely available for Linux.

Notice I said "most of the time." What if the machine that scans the services hangs as well? Anyone who

Figure 1—In this simple phone line interface, the processor generates a one-bit pulse-modulated signal, which is low-pass filtered and sent over the phone line. The transformer is for DC isolation between the circuit and phone line, and the sidactor protects against high-voltage spikes and surges.



uses computers on a LAN knows that a network problem often locks up every computer on the LAN segment, even if it's not involved in any of the traffic or using the LAN at the time.

Or, what if the UPS that handles a critical service dies as soon as the power glitches and the machine that sends the e-mail SOS happens to be on that UPS. These are the kinds of situations in which you want your operator to be notified.

In some situations, adding a PC to monitor critical service isn't practical. Imagine an independent device that can monitor events, dial a paging service, and send a page to alert the on-call operator that something is amiss, or perhaps that everything is OK.

Before I show you my prototype, let's go over some basics on the protocols used by paging services. These protocols are called central office protocols, and one of them is telelocator alphanumeric protocol (TAP).

CENTRAL OFFICE PROTOCOLS

There are two protocols for communicating with paging service providers—one for numeric and one for alphanumeric paging.

Numeric paging is the simplest. You just use standard touch-tone tones to tell the paging service which numbers to send to the pager. The hardware is relatively easy, too. Figure 1 shows a touch-tone telephone interface that might be used for a BASIC Stamp.

The relay switches the dialer in and out of the phone line, and the transformer isolates the Stamp circuit from the phone line. The Stamp generates a pulse-coded signal that, when low-pass filtered with an RC filter, approximates the touch-tone frequencies needed. The circuit also has some spike protection.

With a simple dialer, dialing a numeric pager is as easy as:

- pick up the phone and wait for the dial tone
- dial the pager number and wait
- enter an optional PIN and wait
- dial the number to send a message and terminate with a #
- hang up

Device	Central Office
<CR>	"ID="<CR><LF>}
<ESC>"PG1"{password}<CR>	<ESC>"[p"<CR>
<STX>	
PagerID<CR>	
Message<CR>	
<ETX>	
checksum	
<CR>	Message sequence
	<CR><ACK> <NAK>
	<RS><ESC><EOT><CR>

Table 1—Here's the basic syntax that occurs between the device and the central office.

The tricky part is knowing how long to wait. Certain call-progress tones tell you if the line is available or busy, or when to send the PIN and message.

To decode these signals, you need a telephone interface that can decode the tones and give you digital signals. Many modem chips have this capability.

But if you don't have a way to decode call-progress indicators, you can still use numeric paging. You have to determine the appropriate timeouts and wait periods empirically. Just dial the paging service and use a stopwatch to get some upper and lower bounds for the wait times.

Because there's no guarantee that the message will get through and be received, be persistent and keep trying every 5–15 min. during the alarm.

To use TAP, you need a 300- or 1200-bps modem. Many central paging services don't accept connects from auto-training modems. If you have a "V.everything" modem, turn off the protocol negotiation and tell it to use standard 300- and 1200-bps modulation schemes. This also means no error control/correction/compression protocols.

The protocol is fairly simple, but you would think otherwise if you read through the actual definition. Table 1 shows the basic syntax of the messages.

Perhaps the hardest part of the process is obtaining the

telephone number of the central office for the paging service. These numbers differ from the pager number assigned to you, and you have to get them from your pager service.

Once you have the paging service central-office number, you can dial it and the modem pool will answer with a 300- or 1200-bps modem. Remember to turn off any compression or error-correction protocols on your modem.

After connecting, the central office determines the transfer rate you need. Seems redundant, but remember, these standards were designed to be robust.

The transfer rate is determined by sending a carriage return (<CR>) every 2 s until the service answers with the system prompt, ID =. The prompt may be followed by a <CR> or a line-feed (<LF>) <CR>.

Now you can log in. You need to identify the protocol you wish to contact (in our case, PG1). Some paging systems require a password, which is added to the protocol identifier. If the password is 00000, then the complete string is PG10000 followed by a <CR>.

The central office then sends some text that is not important to the protocol. It acknowledges the login by sending <CR><ACK><CR>.

If the login is wrong, the protocol sends <CR><NAK><CR> and you need to try again. If the system is too busy or otherwise not available, it may ask that you disconnect by sending <CR><ESC><EOT><CR>.

When the system is ready to accept messages, it sends <ESC>[p <CR>.

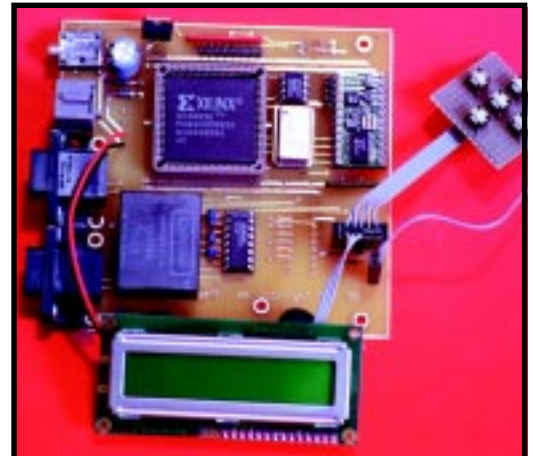


Photo 1—Here's the prototype PCB, with LCD and button matrix. The power is provided by an external 9-VDC 500-mA wallwart-type power supply.

which means you can start sending requests. The paging requests are made up of transaction blocks. Each block is initiated with a start-of-text (<STX>) character and terminated with an end-of-text (<ETX>) character followed by a three-digit checksum and a <CR>.

Each block must be less than 250 bytes. If it needs to be longer, use <ETB> instead of <ETX> to end the block. The next block will then continue where this one left off.

If <ETX> or <ETB> is used, a checksum is sent. Long blocks are usually not encountered because most pagers can't deal with messages longer than 80 or 120 characters.

The blocks consist of fields, each one terminated with a <CR>. For the alphanumeric paging protocol, there are two fields. The first is the pager ID, and the second is the message. Using <CR> as the field terminator implies that <CR> is not a legal character in an alphanumeric pager message.

When a block with the pager ID and message has been sent, the service responds with an optional message and either acknowledges or requests a

resend of the block with the <CR><ACK><CR> or <CR><NAK><CR> response.

It may also tell you that something in the block was wrong but not to retry, by sending a <CR><RS> <CR>. This response might be sent if the pager ID was invalid.

When the service wants to disconnect, it sends <CR><ESC><EOT><CR>.

Most services let you send more than one message, so sending pages to several pagers is efficient. But, some services may ask you to disconnect after each message is sent.

After your messages are accepted by the paging system, it queues them for transmission. If the service is very busy, it may take several minutes to transmit the message.

The service is supposed to reject new messages if it can't handle them. In theory, this means that once the service accepts the message, it's committed to delivering it to the pager. Of course, the pager may be turned off or out of range. Remember, be persistent.

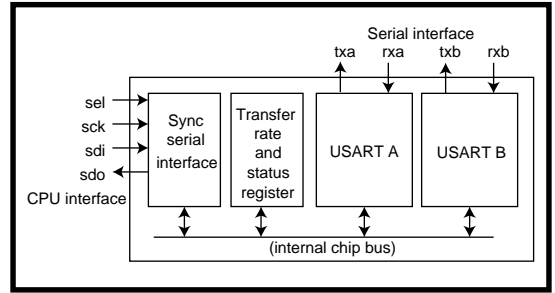


Figure 2—In the FPGA design, there is a three-wire serial interface for talking to the Stamp and several USARTs, as well as data-rate generators for the USARTs.

MAKING IT HAPPEN

To make it easy to add paging to a monitor or alarm application, I decided to build a paging device. This device does a little more than wait for alarm events. It can also be used to arbitrate redundant resources depending on their status. This ability is important, for example, if you have two web servers (one primary and one spare).

Normally, you want both web servers to field requests to balance the load. But if one fails, the other server needs to know about it.

When a web server acts as a backup, it needs to configure a host-address

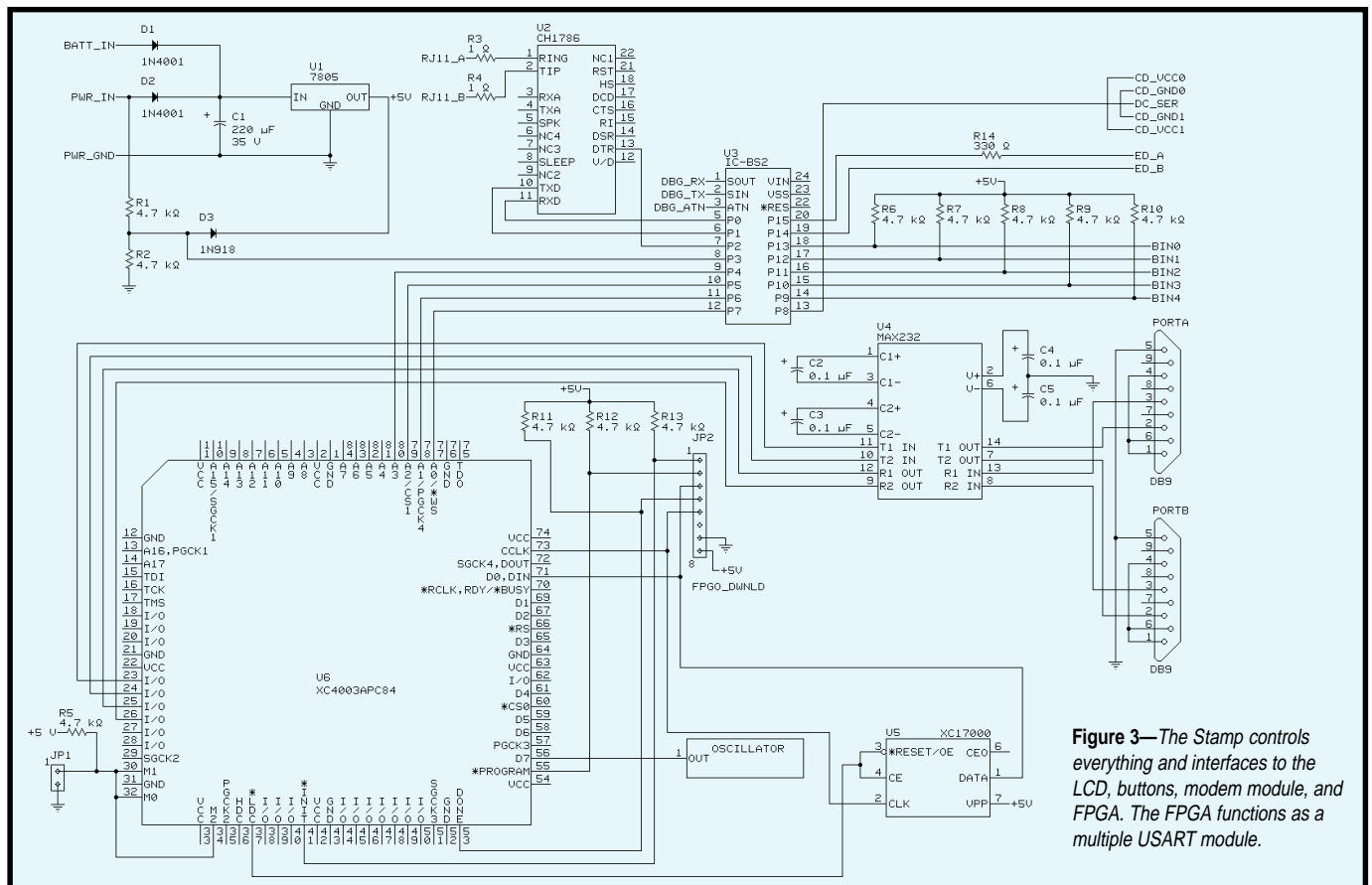


Figure 3—The Stamp controls everything and interfaces to the LCD, buttons, modem module, and FPGA. The FPGA functions as a multiple USART module.

alias so it can pretend to be the other web server. When the failed server comes back online, the backup has to be able to hand off the alias without dropping any connections in progress.

The primary function of the paging device is to page an operator when something is not normal (e.g., if one or both of the servers or ports have failed or if the AC power has gone out).

If any of these situations happen, the device pages up to three pager numbers and sends a message to each. The message is a short numeric string that contains the status of the device and a programmable device ID.

To monitor these external devices, I used two serial ports so that I can use two different schemes to sense external devices. In the simplest form, you can wire the transmit and receive connection of the serial port together to form a normally closed alarm circuit. If the circuit opens, a fault is detected.

In this mode, you can attach simple devices like thermostats, water detectors, or any kind of alarm device that has a relay output with a no-connect. Because RS-232 signaling is used over this loop, the distance can be quite far—thousands of feet, if necessary.

You can attach devices that have RS-232 ports to the paging device. In this mode, the protocol is simple. The paging device sends a single ASCII character to the external device.

If the character is echoed in less than a second or so, everything is OK. If the character is not echoed or an F character is returned, it's considered a failure and the pager device goes into calling mode.

In addition to sensing the state of the two ports (A and B), it also senses if the AC power goes away. This circuit is not very sophisticated because it only senses if there's enough power to power the device. Its 9-V backup battery lets it tell someone about the power loss.

For a user interface, it uses a 2×16 LCD and several buttons. You can program the pager to call up to three numbers and a system ID.

The system ID helps you to identify which device sent the message. When the LCD isn't used to interact with the system, it displays the status of the two ports and the AC power.

I used a BASIC Stamp II for this project, but other PIC processors can be used as well. I'm currently porting this project to a Micromint PicStic.

The Stamp handles the entire user interface with the LCD and buttons. The LCD interfaces to the Stamp with a serial interface that uses only one I/O pin. The five push buttons are muxed using three I/O pins, and debouncing is handled in software.

To permit asynchronous communication with a host via the serial port for the arbitration function, the serial port function is implemented in an FPGA. Implementing the UART externally to the Stamp lets it receive serial data at the same time that it is transmitting—a situation that occurs when you use the port in an alarm circuit. The FPGA interfaces with the Stamp via a three-wire serial interface.

Implementing the UART in an FPGA is overkill, but having an FPGA onboard enables you to implement other interfaces besides standard serial protocols. For example, you can implement the Dallas 1-wire protocol used with their temperature sensors in the FPGA.

To adapt the TTL serial levels to RS-232 levels (necessary for communicating with a server) I used a MAX232 RS-232 line driver/receiver. The MAX-232 generates ± 10 V using a flying capacitor converter from a 5-V supply.

The line drivers in the package use the ± 10 -V supply to provide RS-232 line levels. This setup can be a bit noisy, so a bypass capacitor across the V_{CC} and ground pins is necessary.

The TX/RX signals then just go to a DE-9 connector. To interface an alarm circuit, all you have to do is get a DE-9 plug and some wire to connect it to a no-connect contact. The RS-232 levels ensure that there's sufficient drive to overcome some rather long cable runs.

Finally, I used a Cermetek modem module that interfaces with the telephone circuit. The modem module is a complete 1200-bps modem with a Hayes-compatible command set in an extra-wide (0.700") 24-pin DIP package.

The interface to the Basic Stamp is via TTL-level signals, RX/TX, and DTR/DCD. Because I only implemented numeric paging, I could get away with using the simple circuit in Figure 1.

But, designing in a 1200-bps modem will enable me to add alphanumeric paging by simply changing the software.

BASIC Stamps are programmed in PicBasic. A small BASIC interpreter lives in the PIC chip in the BASIC Stamp and interprets BASIC tokens read from a serial EEPROM.

The PicStic, however, is a native PIC implementation and a PicBasic compiler can be used to program it. The compiler is compatible with the BASIC dialect in the BASIC Stamp I, which is similar to that in the Stamp II. So, porting this code to a PicStic shouldn't be too hard and it will run faster.

After initializing the FPGA interface and the modem, the program enters a big main commutator loop. This step is necessary because the BASIC Stamp doesn't have threads or interrupts.

The main loop calls out the arbitration state machine, which is responsible for implementing the logic. The state machine drives the two serial ports and receives input from them.

When it decides that an alarm condition has occurred, another state machine dials the numbers and implements the required timeouts. It's important that the dialer state machine and the arbitration state machine can both be run in parallel.

Along with the state machines, there's a user interface loop that looks at the button states and enables the user to view and change the three phone numbers and the ID number. These parameters are stored in the BASIC Stamp's serial EEPROM so they can be remembered even when the power goes out.

The power monitor is a diode-protected high-impedance voltage divider network with an input to one of the Stamp's I/O ports. If the unregulated voltage falls below the threshold, the program considers this an alarm condition and signals the dialer state machine.

The Stamp I/O pins are protected with diodes so an overvoltage isn't harmful as long as it doesn't inject too much current. The unregulated input can be as high as 12 V, so I just added an in-line resistor to limit the current.

When the main supply fails, a diode network transfers input power to a 9-V

transistor battery. Because I'm only using a linear regulator, I have about 30–45 min. to get off the alarm pages. A more expensive switching regulator would extend the battery life, especially if the sleep mode of the BASIC Stamp was used to conserve power.

The wired prototype in Photo 1 has been implemented in PCBs, and several units are available. I'm also working on an upgraded version of this device.

So, now I've shown you some uses for pagers in embedded-control applications. Using pagers is a cost-effective means for wireless low-bandwidth one-way communication from machine to machine or machine to human. ☐

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.

SOURCES

Modem, telephone interface

Cermetek Microelectronics, Inc.
(408) 752-5000
Fax: (408) 752-5004
www.cermetek.com

PCB

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

Basic Stamp

Parallax, Inc.
(916) 624-8333
Fax: (916) 624-8003
www.parallaxinc.com

PicStic

Micromint, Inc.
(860) 871-6170
Fax: (860) 872-2204
www.micromint.com

MAX232

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

Glen Reuschling

USB Primer Low-Speed USB Host Controller



Almost two years ago, I read a trade-publication announcement for a new 8-bit USB microcontroller from Cypress Semiconductor. Coming from a background in PID and PLC controllers, I immediately saw the potential for such a chip.

At \$1 apiece, you could create families of USB-smart motors, sensors, and actuators, all plugged into a USB-based microcontroller. Imagine not having to worry whether a thermocouple is J, K, or T type because the calibration and correction factors are already taken care of by the USB microcontroller.

A standard digital interface eliminates the need to configure special inputs and outputs before shipment to a particular customer. The tiered star arrangement of the USB interface lends itself nicely to the wiring structure of many industrial machines and eliminates the rat's nest of wires that converges at the back of an embedded multiloop controller.

The resulting collection of USB smart peripherals forms a simple distributed processing network, a potentially useful feature. The hot-swap and plug-and-play features of the USB offer the possibility of being able to do service and maintenance on a machine without costly shutdowns.

Not only does a digital signal path offer tremendous noise immunity

over millivolt-level analog signals in the industrial environment, but it also allows for the use of optoisolators for electrical isolation—a must in an environment where 220 and 440 VAC are standard working voltages.

Finally, the ability to put a USB microcontroller to sleep means low power consumption for those situations where that's a consideration. With all of this in mind, I ordered the Cypress USB Development Kit ("USB Micro," *Circuit Cellar* 88).

When it arrived, I fired it up, wrote, downloaded, and ran a few simple programs, but that was it. My older workbench computer doesn't have USB ports, nor were there any USB interface plug-in cards available.

On top of that, the only USB software driver available was a beta version for Windows 95. All my development software is DOS- and Windows 3.1-based, and I wasn't ready to buy a new Pentium motherboard with USB support for my workbench and then install Windows 95 on it.

Not to be deterred, I looked into the availability of a chip or chipset that I could use to implement a USB host controller. The few I could find were all targeted to the PCI bus. None were targeted to the embedded microcontroller market.

I checked into the possibility of obtaining a USB host controller as intellectual property in HDL format, but price tags were five and six figures in size. Another dead end.

After downloading and reviewing the Universal Serial Bus Specification, Rev. 1.0, it was apparent that the low-speed USB specifications were a somewhat reduced and doable subset of the full-speed USB specifications. At this point, I decided that I could and would build my own low-speed USB host controller from scratch.

Eight months and many dead ends later, I finally put together the four-port USB host controller shown in Photo 1, tied it to an 8051-type microcontroller, and got it to talk to the CY7C3650 USB development card. So, what started out as a personal challenge to build my own USB host controller from scratch turned into a comprehensive USB learning experience.

Part
3
of
4

To wrap up our series on the Universal Serial Bus, Glen presents a two-part discussion on building USB hardware from scratch. This month, he lays the groundwork for putting together a USB host controller.

This article recounts the work I did in building my low-speed USB host controller. By describing the problems I encountered and the solutions I came up with, my goal is to fill in the gaps in the standard USB documentation and make the USB more accessible.

This article is not a comprehensive review of the USB. I assume you're already familiar with the basics of the USB operation. For those of you who need to brush up on it, there are many good introductory articles to be found in various hardware-oriented publications, starting with Parts 1 and 2 of this *Circuit Cellar* MicroSeries.

Other excellent sources of information on the USB interface can be found in the datasheets and application notes published by the various chip manufacturers that offer USB products. A good example is Anchor Chips' web site, where a number of documents are posted that contain a wealth of tutorial-level information.

HOST CONTROLLER'S ROLE

Within the USB operation, the host controller has a distinguished role in contrast to those devices that are referred to as USB microcontrollers. The USB is a half-duplex serial bus with only one bus master (i.e., the host controller), but all of the USB microcontrollers that are commercially available are (by design) bus slaves and cannot perform the host-controller functions.

The advantage of the USB is that by offering a single standard interface for all low- and medium-speed peripherals, it eliminates the hassles of installation and configuration. It also offers true plug-and-play and hot-swap capabilities—features which, until now, have not been generally available to desktop PC users.

Although funneling all the different peripheral device communications through a single interface may make life easy for the PC user, it poses a major

challenge to the hardware and software developer. The problem with this arrangement is that a single interface must now do the work of all the various plug-in cards and software device drivers that currently reside in the desktop PC.

Now, all the functions that could previously be spread out over a number of pieces of hardware and software must all be incorporated into a single combined hardware/software interface—the host controller.

To deal with the range of demands that the different kinds peripheral devices put on the USB interface, the USB Specification calls for four different modes of data transfer (control, interrupt, isochronous, and bulk transactions) and two different bus speeds (full and low speeds with data rates of 12 and 1.5 MBps, respectively).

Applications pass data through the USB by supplying the host controller with pointers to memory locations where data is to be moved from or to. In turn, the host controller keeps track of everything by using linked lists of linked lists of these pointers.

It is this complexity of the host-controller function (i.e., its need for fast memory bus access and a fast CPU to handle the computational

overhead of each peripheral's associated device driver) that have restricted the commercially available full-speed USB host controllers to machines with either a PCI or other fast Local bus and OSs like Windows 98.

FULL SPEED VS. LOW SPEED

The full-speed USB protocol supports all four modes of data transfer with a maximum bandwidth of 1.5 MBps and a maximum data payload per transfer of 1023 bytes per packet per 1-ms frame for isochronous transactions [1, p. 55].

The host controller is required to prioritize and schedule all individual transfers, track the timing on isochronous transactions, and do error checking for bulk transactions. That's why full-speed USB host controller functions will always be beyond the capacity of the small embedded microcontroller.

On the other hand, low-speed USB supports only control and interrupt transactions, the two modes that entail the least computational overhead. Although the theoretical maximum bandwidth for low-speed transactions is 187.5 KBps, the maximum packet size per transaction is eight bytes.

This fact, combined with the maximum bus-access frequency per device of once every 10 frames, translates to a maximum data payload of only 800 Bps per low-speed endpoint channel (i.e., a doable data rate, even for a small 8-bit micro) [1, p. 57].

If you intend to interface exclusively to low-speed USB devices, a USB host controller becomes a possibility for small embedded micros. The requirements to support low-speed host controller functions are well within the capabilities of any fast 8-bit controller.

Regarding low-speed data rates, the bus access rate of once every 10 frames is a specification, not a hardware limit. It may be possible (depending on the low-speed device being used) to exceed this 800-Bps data rate by accessing it more often than once every 10 frames.

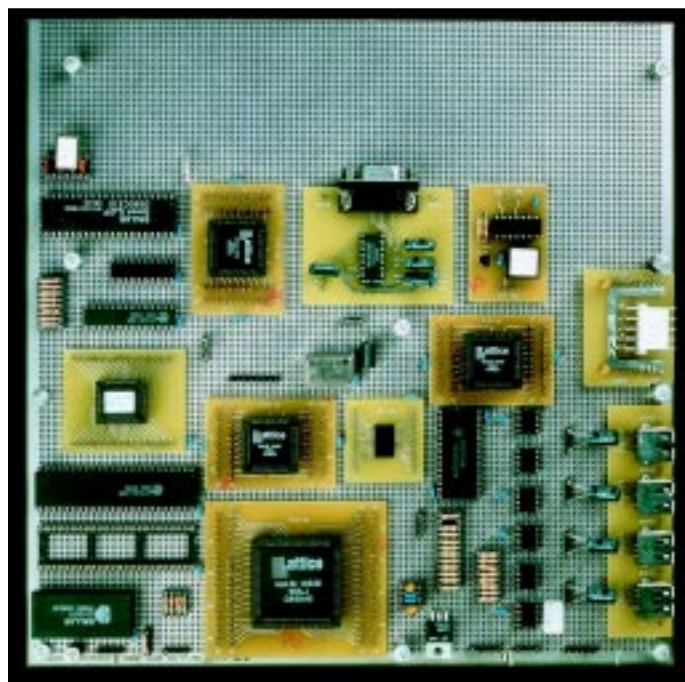


Photo 1—This is it: the final version of my host controller project. The parts along the top and left of the photo include a standard 8051-type microcontroller, and the parts filling in the bottom right form the μ SIE.

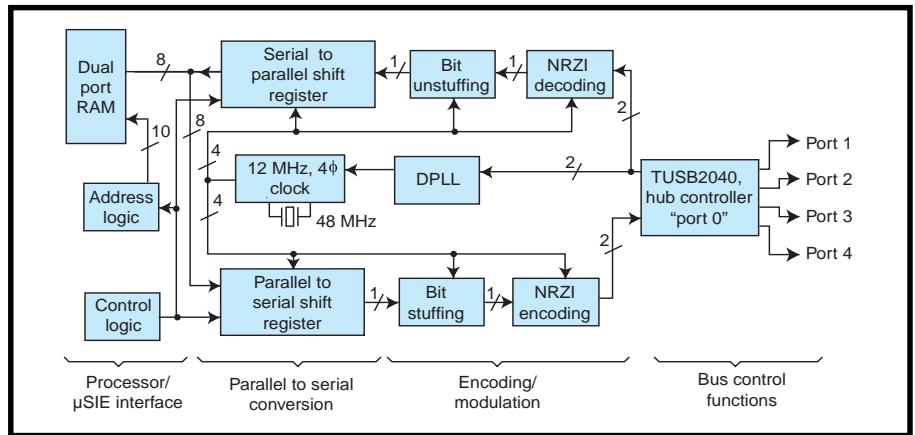


Figure 1—These are the basic functional building blocks that compose my low-speed host controller's μ SIE.

A second way to multiply data rates to an USB device is to access multiple endpoints within a single device. However, low-speed devices are limited to only two endpoints besides the default endpoint 0 [1, p. 47].

On the mechanical side, full-speed USB specifies a shielded twisted-pair cable for the signal lines, but for low speed, neither twisted pair nor shielding is specified [1, p. 86]. Maximum specified lengths are 5 m for full-speed and 3 m for low-speed cables [1, p. 89].

But, for practical purposes, signal integrity and the bus turnaround time set the upper usable length limits. So, despite the specifications, with proper cable selection, you should be able to wire a house-sized area with low-speed USB peripherals.

No doubt you've already encountered the term serial interface engine (SIE). The SIE is to the USB what the UART is to the RS-232 interface. The only functional difference between the two (from the processor's point of view) is that a UART passes data on a FIFO buffer (possibly only one byte deep), while the SIE passes data directly to and from the processor's memory via pointers.

For the sake of discussion, I'll use the term "micro serial interface engine" (μ SIE) to refer to the minimum hardware necessary to implement completely and correctly all low-speed USB host-controller functions. Interestingly enough, the μ SIE hardware requirements for the host controller are less than those for a USB slave SIE.

Because the USB protocol encompasses both hardware and software

issues, it makes sense that a host controller is partly hardware and partly software in its makeup. For this reason, there are many different ways to build a host controller, depending on which functions are implemented in hardware and which ones are implemented in software.

HARDWARE INTRODUCTION

The first step in building a USB host controller is to visit the USB Implementers Forum web page and download:

- "Universal Serial Bus Specification," Rev.1.0 and the white papers
- "Cyclic Redundancy Checks in USB"
- "Designing a Robust USB Serial Interface Engine (SIE)"

For a membership fee of \$2500 per year, you can join the USB Implementers Forum and have extended access to documents and tech support. But, this membership fee is out of the range of a lot of people, including me.

If you choose to download and read the USB Specification, two notes of warning are in order. First, the full USB specifications encompass issues of software, hardware, and communications protocol.

Unfortunately, the authors of the document use the same words at different points to reference alternately hardware, software, and communications protocol issues. Although this overloading of definitions may not bother a C++ programmer, it blurs the lines between the different aspects of the USB specifications and makes

them confusing to read from a hardware builder's point of view.

Second, the USB Specification is not and was never written to be a systematic guide for building USB hardware. With patience, most of the information you need to guide you in building a USB interface can be found in the USB Specification. As for details not explicitly stated in the Specification, be prepared to resort to some trial and error work to find them.

To help you locate information within the USB Specification and elsewhere, I included a list of references that source specific page numbers and sections.

HOST CONTROLLER HARDWARE

Figure 1 shows the minimum hardware configuration necessary to implement a USB host controller. A short guided tour of this basic μ SIE starts with the output section, which is the easiest to implement.

USB uses 8-bit bytes as its basic unit of data. So, as in all serial interfaces, the signal path begins with a parallel to serial conversion. Data is loaded one byte at a time and shifted out starting with the least significant bit.

The next step in the signal path is encoding and modulation, which includes bit stuffing and NRZI encoding [1, pp. 121–122]. Extra bits are added to the signal stream to ensure adequate transitions for syncing and clock separation. Then, data information is encoded as a differential signal and control information is encoded as DC level signals [1, p. 115].

The serial signal stream is then passed to the output buffer section. This section is responsible for line driving, slew rate control, hot-swap power management functions, and low- and full-speed device detect.

The return signal path becomes more complicated. Because there's no separate clock line in the USB, the receive clock must be derived from the incoming signal. This function is performed by a digital phase lock loop (DPLL) [2, p. 2].

DC level control signals must be detected, bus time-out intervals must be monitored, and this information passed on to the control section of the

μ SIE. Once it is past this input section, the return serial signal path is just the reverse of the outgoing path.

To avoid using memory pointers, the μ SIE uses a dual-port RAM as the processor/ μ SIE interface. Rather than passing pointers, the microcontroller's software is responsible for placing data to be sent or received only at specific memory locations. This is an example of a hardware/software tradeoff you can make when building a host controller.

Although the control logic is shown as a small block in Figure 1, it represents a major portion of the μ SIE. Also, not shown is the 1-ms frame clock and the functions of CRC generation and checking that can be implemented quite easily in software by the microcontroller.

JUST GETTING STARTED

Now you've got a USB background and some resources to check out for more information. Next month, I'll introduce my low-speed USB host controller and the lessons I learned while putting it all together. My goal is to show you that there's no reason USB should remain only the province of desktop PCs. ☒

Glen Reuschling is a manufacturing engineer by day and a serious hobbyist by night. His most recent home project resulted in this article. You may reach him at wildiris@cruzio.com.

REFERENCES

- [1] USB Implementers Forum, *Universal Serial Bus Specification*, Rev. 1.0, www.usb.org, 1996.
- [2] USB Implementers Forum, *Designing a Robust USB Serial Interface Engine (SIE)*, www.usb.org.

SOURCES

USB Development Kit
Cypress Semiconductor
(408) 943-2600
Fax: (408) 943-6848
www.cypress.com

Anchor Chips, Inc.
(619) 613-7900
Fax: (619) 676-6896
www.anchorchips.com

FROM THE BENCH

Jeff Bachiochi

Demystifying LCD Muxing



As the number of LCD segments increases,

so do connection problems. This month, Jeff takes an up-close look at multiplexing LCDs and the problems we're likely to encounter working on such a task.



How many times have you begun a project where, for it to work correctly, the output must be nothing? That's the price you pay for working with LCDs.

I'm not talking about those LCD modules where you speak serial or parallel and its built-in processor handles all the liquid-crystal control interfacing. I'm talking about handling those signals on your own.

For an LCD to remain functional, its drive must average 0 V. Refer back to my column in *Circuit Cellar* 78 to get the basics on the physical make-up of LCDs.

This month, I explain how multiplexed drivers create signal levels that enable some segments and not others. In *Circuit Cellar* 78, I talked about static displays where there's only a single common to every LCD segment. If the common is grounded, as in Figures 1a

and 1b, then each segment can be turned on by applying an alternating positive and negative voltage to each segment. Or to prevent the necessity of a negative voltage, an alternating voltage is applied to the common, as in Figure 2.

If the same waveform is applied to a segment, its potential is 0 V and the average is 0 V. But if the waveform is inverted and applied to a segment, it now has a voltage potential across it, but the average voltage remains 0 V.

WHY MULTIPLEX?

It seems a shame to complicate the static method of twisting those liquid crystals. A single common with multiple segments is simple in design and implementation.

But, somewhere down the road, adding more segments isn't feasible. Take a simple time-of-day clock, for example. A segment for the tens digit, one for the colon, plus seven for each of three other digits equals 23 segments. That's a lot of I/O to just run the LCD segments. What happens if you don't have that much I/O available? Multiplexing is the answer.

For an LCD to be multiplexed, it must be manufactured with that in mind. Static displays can't be multiplexed, nor can a multiplexed display be completely run in a static mode. A multiplexed LCD has segment connections shared among multiple commons.

Let's look at the clock model to see how it might be manufactured in both arrangements. Figure 3a shows 26 connections necessary to produce a static display. Figure 3b shows how the number of connections can be reduced to 11 by using four common lines.

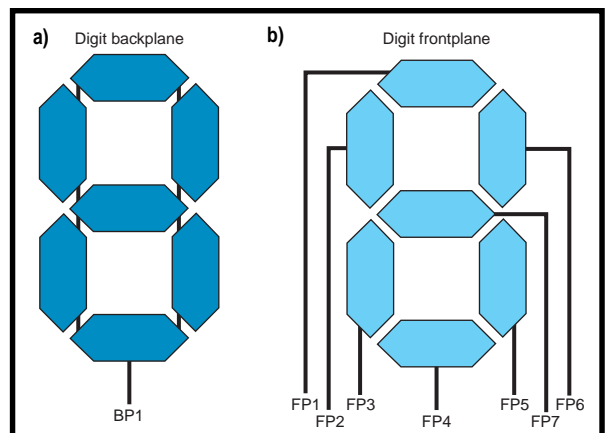


Figure 1a—A single-digit static LCD may have all backplane segments tied to one common pin while the front-plane segments each have their own connection (b).

NOT FOR FREE

Getting back all of those I/O lines isn't free. In fact, it'll cost you dearly. First of all, there's the display itself. If a segment connection is sharing signals with more than one common, then you only get to use a portion ($\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, etc.) of the signal for each segment.

With four commons, each of the four segments associated with the segment connection is enabled by one of the commons for one quarter of the frame time. The frame time is the time to cycle through each common once. If a particular segment is enabled for only one quarter of the frame rate, two things change.

Because the overall frame rate is reduced (four commons must now be individually cycled), the display seems to flicker because it is now effectively one quarter the static rate. Even if the frame rate is increased to compensate, each segment's on time is now only one quarter of its original on time. Thus the contrast (and viewing angle) is reduced.

The control signals for multiplexing are another story. The trick again is twofold. Simple logic-level control is out the window because of LCD rule number one—only AC signals are allowed as the potential across each segment that must average 0 V. With the logic-level controls, there's no way to share signals and get an AC potential of 0 V. Enter bias-level multiplexing.

BIAS LEVELS

Bias levels are voltages other than just V_{CC} and V_{SS} that drive the segment and common lines of a multiplexed LCD. Figure 4 shows how four segment inputs and four commons use a one-third bias to control 16 entirely separate segments. Note how each of the control signals interacts, yet the voltages across each segment remain symmetrical AC signals.

When a backplane or common like BP1 is in its active portion of a time frame, it switches between V_{lcd} and 0 V. Although other backplanes are active, BP1 disables its segments by switching between $\frac{2}{3} V_{lcd}$ and $\frac{1}{3} V_{lcd}$. Segments such as SEG1 behave similarly. During a time frame, the segment is enabled in turn by each of the backplanes.

When a segment backplane pair needs to turn a segment on, the segment presents voltages of V_{lcd} and 0 V directly opposing that of the enabling backplane. To turn off a segment, the segment offers voltages of $\frac{2}{3}$ and $\frac{1}{3} V_{lcd}$ in step with that of the enabling backplane.

On segments have a potential of $2 \times V_{lcd}$ across them, and off segments have only $2 \times \frac{1}{3} V_{lcd}$ across them. The magic happens because of the difference between V_{off} and V_{on} for the liquid crystal fluid.

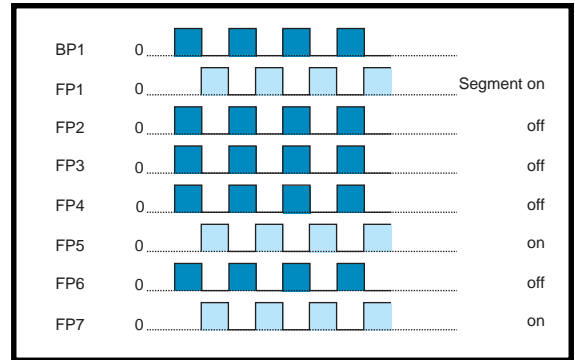


Figure 2—Static designs offer simple, straightforward interfacing.

BIAS ORIGINS

The simplest approach for providing analog voltage levels (bias) to the LCD is through a resistive voltage divider network. Other methods such as DACs or charge pumps may create discrete analog levels. Imagine connecting these discrete levels to each backplane and segment connection.

This process would require a large number of CMOS switches (four for each connection). Luckily, much of this is handled within backplane and segment driver chips.

Unlike the static situation where an on segment had AC voltage across it and an off segment didn't, multiplexed segments always have some voltage across them. Because of this, the V_{lcd} becomes critical. If the V_{lcd} is too high, the off voltage ($\frac{1}{3} V_{lcd}$ for $\frac{1}{3}$ bias) won't be low enough to turn off the segment and all segments will appear on.

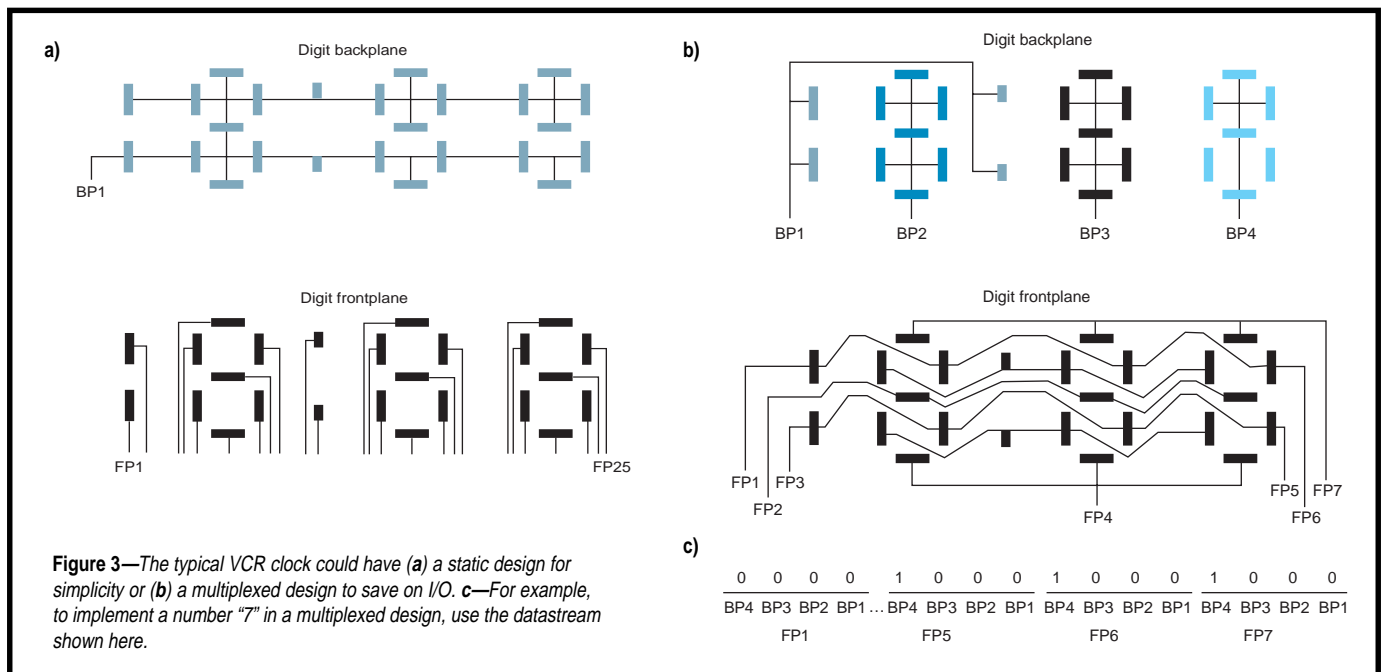


Figure 3—The typical VCR clock could have (a) a static design for simplicity or (b) a multiplexed design to save on I/O. c—For example, to implement a number "7" in a multiplexed design, use the datastream shown here.

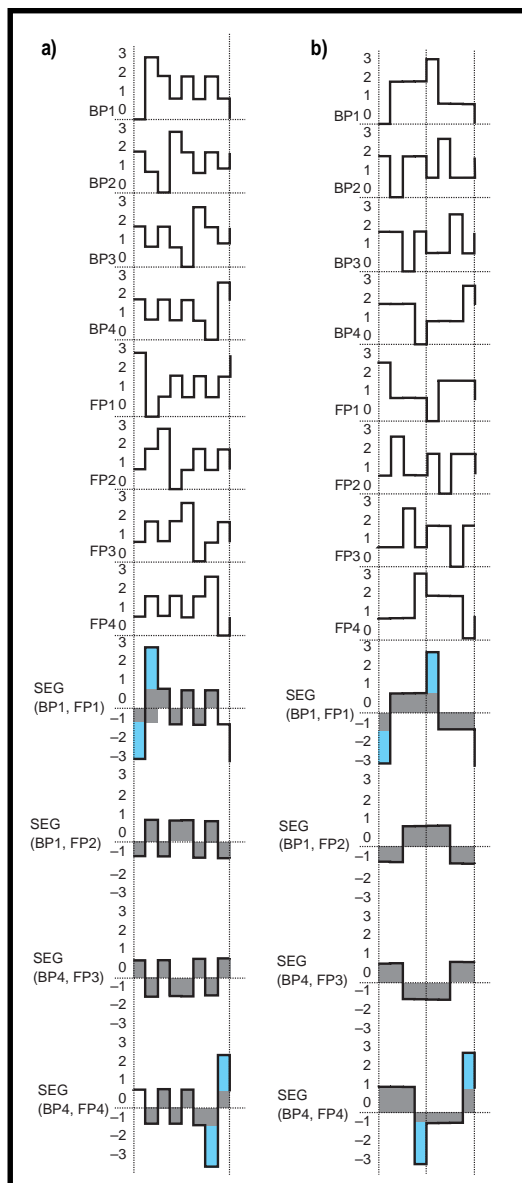


Figure 4—Multiplexing can be accomplished by sequencing through each backplane either by backplane (a) or by polarity (b).

The opposite also happens. Too low a voltage will fail to exceed the saturation voltage necessary to turn on a segment. To allow for some adjustment of the V_{lcd} , a potentiometer is used between V_{dd} and V_{lcd} (or V_{ref}).

In environments where temperature is not constant, thermal compensation is necessary. Negative temperature coefficients of liquid crystal fluid create contrast problems as the temperature falls and ghosting as temperature rises.

To compensate, the V_{ss} must be supplied from a compensation circuit. A temperature sensor and op-amp can supply the optimum drive voltage based on the manufacturer's specifications. This spec is based both on the fluid

type and the number of multiplexed backplanes.

Keep in mind that there are tradeoffs for increasing the number of backplanes. As the number increases, the contrast, viewing angle, and effective temperature range all decrease.

To prevent the on segments from flickering, the frame time must be kept under 33 ms ($1/30$ s) and generally greater than 11 ms ($1/90$ s). A higher frame rate means a higher load on the drivers.

Only AC is allowed across any segment. DC voltages higher than 50 mV cause permanent damage. Although the waveforms placed on the segments have been carefully designed to average 0 V, the bias network must be accurate to assure cancellation to within 50 mV.

If a resistor divider network is used, pick a good 1% metal film resistor. Even 5% resistors can produce over 100-mV bias due to tolerance issues.

PREPACKAGED DRIVERS

There are many manufacturers of LCD drivers. Except for some of the smaller segment and common count drivers, package styles are all SMT, mainly because of the high density attainable.

Some larger display drivers support more than 100 commons and 100 segments. These are

often split into separate driver chips, one each for the segments and commons.

Each manufacturer favors a slightly different interfacing scheme. Sharp includes both parallel and serial interface options with very high pin counts. Philips uses their I²C serial interface with the smaller drivers available in 28- and 40-pin DIP packages. Rohm offers serial interfaces for their SMT drivers, as does Motorola.

Most drivers have an onboard oscillator for automatic waveform generation and bias generators for the multiple voltage sources. Let's look at a common/segment driver from Motorola.

As you see in Figure 5, the 52-pin TQFP (tiny quad flat pack) MC14LC-

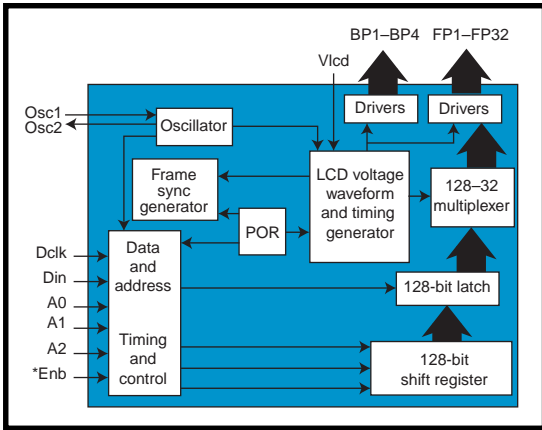


Figure 5—Many backplane/segment drivers contain all the essentials for training those pesky little crystals to sit, stand, and otherwise perform on cue.

500x supports four backplanes (commons) and up to 32 segments. Separate V_{dd} and V_{lcd} inputs allow special compensation circuitry to be used when necessary for the LCD bias drive.

Although an external oscillator can be applied to OSC1, a simple resistor can be used across OSC1 and OSC2. The internal oscillator is divided by 1024 to bring the frequency down to the 30–100-Hz range.

The synchronous serial interface is available in SPI or I²C flavors. SPI is a slightly simpler interface consisting of control word followed by the 128 bits of pixel data in the form BP4, BP3, BP2, and BP1 for each of the 32 segments FP1–FP32.

The data clocked into the 128-bit shift register via DCLK is transferred into the data latch after the last input bit is clocked. The data bits are read from the latch (by the timing generator) to build segment (FPx) outputs with the unique bias voltage necessary to enable or disable LCD segments.

The I²C format requires the driver to respond with a low (acknowledge) bit on the data line after every byte is clocked. In addition to writing data to the driver, a read can also be performed to retrieve data back from the driver. Besides this slightly different interface, the internal operations are the same as the SPI device.

THE FINAL SEGMENT

The number of LCD segments in today's products continues to grow, as does our lust for more information. But, much of the grunt work in designing a system to use multiplexed LCDs has been done. App notes abound to help you complete the task. Good luck! 🍀

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

RESOURCES

LCD information, www.lxdinc.com, www.crystaloid.com, www.sharpmeg.com, www.eio.com.

SOURCE

MC14LC500x
 Motorola
 (602) 952-4103
 Fax: (602) 952-4067
www.mot.com

of CCDs (e.g., blooming, streaking, and smearing) are less problematic for CMOS designs.

Of course, using the CMOS process means all the support logic (timing control, A/D conversion, and all manner of signal processing), which requires extra ICs in CCD-based designs, can be integrated.

Thanks to CMOS, the imaging IC business is no longer just the province of heavyweights that can afford specialized CCD fab lines. The door is open for a lot of competition—a surefire way to hasten innovation.

PC PICTURE PERFECT

Though mega-pixel models are within reach, the real action is centered around lower resolution units up to and including the ubiquitous 640 × 480 VGA class. In addition to Photobit, there are now a number of other suppliers both large (HP and Hyundai) and small (OmniVision and VLSI Vision) targeting the niche. Volume prices for these units are in the \$20 range, which should capture the attention and imagination of designers.

The OmniVision OV7610 is an example of a color imager with digital outputs. Like other suppliers, OmniVision also offers even lower cost monochrome units as well as versions with NTSC/PAL video outputs for consumer products.

As shown in Figure 2, the '7610 combines the image array with video timing generator, signal processing, A/D conversion, and digital video output formatting. Setup and control is via a clocked serial I²C interface.

Just connect a 27-MHz crystal and have at it. Each 8-bit ADC runs at 13.5 MHz, and the output can be formatted in a number of ways (e.g., YUV 4:2:2 or GBR 4:2:2, interlaced or progressive) according to various digital video standards (e.g., CCIR601 and CCIR656).

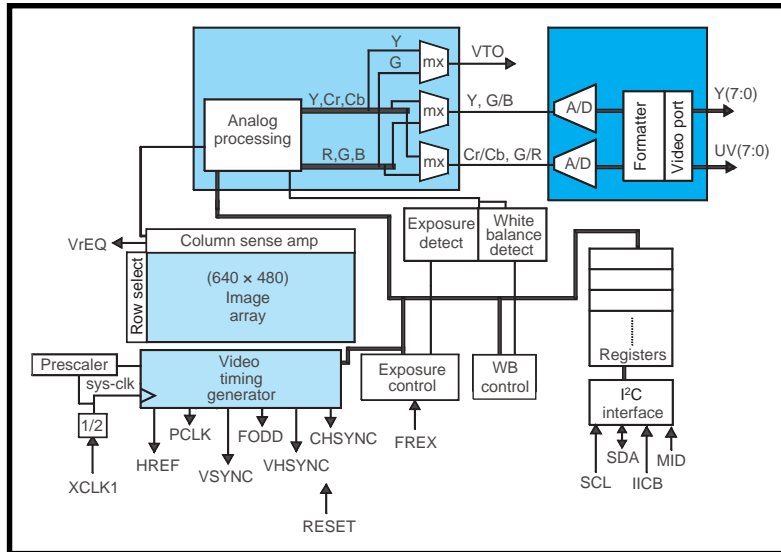


Figure 2—The OmniVision 7610 illustrates the power of CMOS to integrate the entire signal chain from photodetector to 1s-and-0s.

The '7601 handles a lot of the cumbersome (and speaking for myself, error prone) details of getting a good shot, including automatic exposure control, automatic gain control, white balancing, and such. Being a 1s-and-0s man, my first thought would be to crunch the digital output with a DSP. But according to OmniVision, the initial round of signal processing is best performed pre-ADC using analog circuits.

Nevertheless, all the automatic adjustments can be overridden to permit external control. To make it easier, the '7610 keeps track of parameters such as line and field/frame average luminance and color levels. This enables an external controller to diagnose the overall scene and make any desired adjustments to the 50+ on-chip control registers via the I²C bus.

When it comes to hooking cameras to PCs, USB has (finally!) come of age.

To make it easy, OmniVision offers a companion chip (the OV511) that, with the addition of little more than a 256 K × 16 (4 Mb) DRAM, implements a complete USB camera (see Photo 1).

Speaking of companion chips, OEM designers with a big enough PO in their pocket (manufacturing licenses are \$85k) should check out the Clarity 2.0 ASIC design from Sound Vision, shown in Figure 3. It's not only adaptable to a variety of CCD and CMOS imagers (most

recently, Sound Vision announced support for the new HP chip cams), but it also deals with all the other pieces of the puzzle, including LCD viewfinder, CompactFlash card image storage, and computer (USB and RS-232) and TV (NTSC and PAL) interfaces.

Incorporating an ARM720 32-bit CPU core with cache and two-clock multiplier, Clarity 2.0 performs speedy JPEG compression by processing close to a million pixels per second. To allow back-to-back shots, the firmware implements a hierarchical storage scheme that includes DRAM and CompactFlash memory. Compression runs as a background task, so another shot can be taken immediately, even before the previous shot is processed.

CMOS EYE SEES

The promise of imaging goes beyond digital cameras. Consider the M64282FP



Photo 1—USB is finally on the move (ironically, thanks to blessing by the iMacs), and OmniVision follows suit with a highly integrated USB camera design.

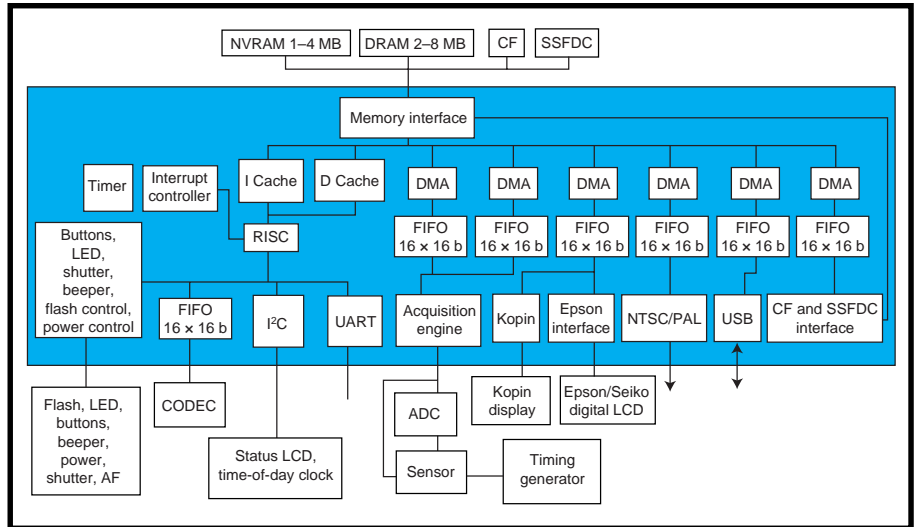


Figure 3—The Sound Vision Clarity 2.0 ASIC includes an ARM CPU core and all the interfaces needed to turn an imaging chip into a complete digital camera.

Artificial Retina (AR) from Mitsubishi. Like the other chips, it relies on a photodiode array (128×128) built with CMOS process, but the similarity stops there. The artificial retina (like the human eye) is designed to perform a variety of local preprocessing before sending the visual input on to a higher intelligence.

As you see in Figure 4, the chip is quite simple and requires only 16 pins. Basically, after the configuration is set up by the clock serial interface, the host uses the START and READ control lines for handshaking and shifts out the analog image data with XCK at up to 500 kHz. Because each frame is roughly 16k pixels, that turns out to be about 30 frames per second.

Yes, you can hang an ADC and micro outside to create a simple camera, but there's more to the AR than that. Besides handing over the raw image, the chip can perform autonomous processing on the data—notably, edge extraction and enhancement.

The on-chip processing is carried out in the analog domain via current-mode calculations between pixels and varying the addressing pattern when scanning the array. For instance, edge extraction subtracts each pixel from its two (1D) or four (2D) neighbors. Furthermore, the relative weight of a pixel and its neighbors can be programmed to implement edge enhancement.

Putting more intelligence in the sensor means less silicon is needed at the other end of the wire to perform higher level functions such as pattern matching, orientation detection, and motion estimation.

Mitsubishi sees opportunities for a new human-machine interface paradigm based on capturing gestures, notably for PC and video gaming. Although the success of such new application concepts remains to be seen, there's no doubt the basic technology is consumer-capable. The AR chip has already been designed in at Nintendo and, by the time you read this, should be shipping to the tune of a million units per month.

PIXEL POWER

As you can see, the CMOS imaging frenzy is just kicking into high gear. No doubt, we'll see units with more resolution and more intelligence on-

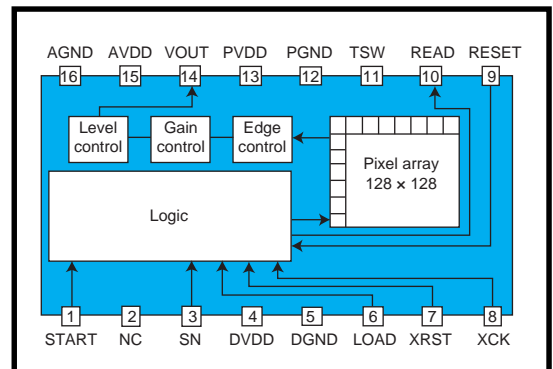


Figure 4—Mitsubishi calls the M64282FP an Artificial Retina because, like a human retina, it preprocesses the data by performing edge detection and enhancement.

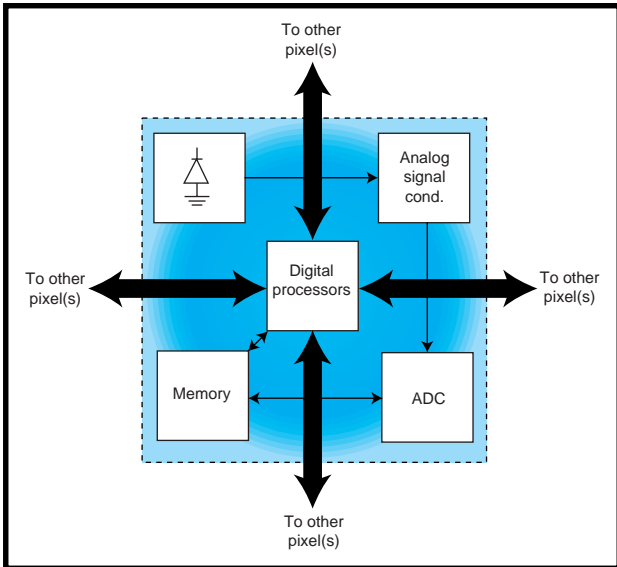


Figure 5—According to researchers at Stanford University, in the future, not only A/D conversion but even memory and processing will be replicated in each pixel.

chip, and aggressive pricing as well. Beyond that, the next major technology shift promises to pack more logic at each pixel.

Recently, I attended a session organized by the local chapter of the IEEE Signal Processing Society titled “CMOS Image Sensors: State of the Art and

For instance, performing the conversion right at the pixel achieves maximum SNR, by avoiding noise introduced when shuffling the analog output of the photodiode to an ADC on the other side of the chip. Also, because each pixel carries its own baggage, the design is inherently scalable.

Future Trends.” In the presentation, David Yang described the CMOS imaging research currently underway at Stanford University.

A major goal of the Stanford effort is to find a way to proliferate A/D conversion to each pixel. Current designs either funnel all pixels through a single ADC or, at best, use a separate ADC for each column.

Putting an ADC at each pixel offers a number of advantages.

Because each pixel performs A/D conversion in parallel, overall imager bandwidth can be quite high even though the individual ADCs can be slow. For instance, a 1-M pixel unit could deliver a billion pixels per second with just a pokey 1-kHz ADC at each pixel.

In turn, the extra bandwidth can be used to extend dynamic range by multiple sampling and other pixel-parallel real-time processing techniques. For instance, by sampling the image at exponentially increasing exposure times, both light and dark areas of the image are captured and the dynamic range can be enhanced by a factor of two.

It all sounds grand, but there is the minor problem of where to cram all the extra transistors. Turns out there’s a bit of a real-estate shortage with technology squeezed between the rock of resolution and the hard place of sensitivity.

The essence of the problem is described by the so-called fill factor (i.e., what proportion of the chip is actually performing image gathering). If the fill factor, typically 30–40%, becomes too

small, resolution (given practical chip-size limits) or sensitivity must suffer.

Fortunately, the march of silicon offers hope. The minimum pixel size is about $4 \times 4 \mu\text{m}$, reflecting the realities of cost-effective lenses and signal quality. As IC geometry shrinks, a relatively larger proportion of the area allotted each pixel will be available for extra transistors.

Still, it's easy to bite off more than even the best fab can chew. To that end, the Stanford researchers have come up with a simple bit serial successive approximation A/D technique that requires adding only a comparator and 1-bit latch at each pixel.


Putting their research into practice, they've fabricated a 640×512 imager using a commercial grade $0.35\text{-}\mu\text{m}$ CMOS process with $10.5 \times 10.5 \mu\text{m}$ pixels that achieves a 29% fill factor and delivers up to 250 frames per second with 8-bit A/D resolution.

The team predicts that pixel-level A/D conversion will become common as IC processes shrink to $0.15 \mu\text{m}$ and beyond. At that point, they'll turn

their attention to the next challenge. Once each pixel has an ADC, why not add the memory to store data at each pixel, and ultimately a processor to crunch it as well (see Figure 5)?

HAVE CAM, WILL CRAM

I suspect the fast pace of development in CMOS imaging technology over the last couple years means you can expect to see chip cams popping up in all sorts of applications (e.g., games, communication, robotics, pattern recognition).

Beyond just swiping market share from their CCD- and film-based counterparts, the most interesting applications will take advantage of the low prices and high-integration opportunities uniquely offered by CMOS imaging technology. 

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

PB-300
Photobit
(626) 683-2200
Fax: (626) 683-2220
www.photobit.com

M64282FP
Mitsubishi Electronics
(408) 730-5900
Fax: (408) 732-9382
www.mitsubishichips.com/ar

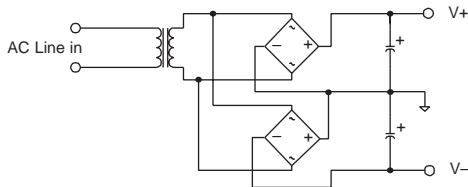
Camera chips
OmniVision Technologies, Inc.
(408) 733-3030
Fax: (408) 733-3061
www.ovt.com

VLSI Vision Ltd.
(408) 556-1550
Fax: (408) 556-1564
www.vvl.co.uk

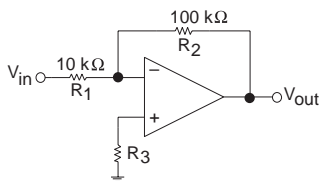
Hewlett-Packard
(408) 435-4303
Fax: (408) 435-4303
www.hp.com

CIRCUIT CELLAR Test Your EQ

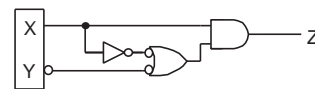
Problem 1—The circuit shown below was designed with the intention of providing both positive and negative voltage rails with respect to a common ground. Why doesn't this circuit work?



Problem 2—The circuit shown below is an inverting amplifier. What is the purpose of R3? What is the optimal value of R3?



Problem 3—Simplify this circuit. Assume zero propagation delay associated with the gates. The signals X and Z are active high. The signal Y is active low.



Problem 4—What is the output of the following C code?

```
void main(void)
{
    int i = (int)0xffff;
    if (i<0)
        printf("i is negative\n");
    else
        printf("i is positive\n");
}
```


PRIORITY INTERRUPT

Circuit Cellar Online



encourage reader feedback about my editorials but I must have really hit a nerve with my June '99 "Servings Per Issue" piece. In fact, I received more e-mails and reader reaction from that editorial than any other in our 11-year history. So what did I say that was so significant? Who was I dumping on this time?

For those of you who don't remember, I was discussing the defined-gratification aspects of the Internet and whether *Circuit Cellar* should evolve to be more Internet-like. This paragraph pretty much summed up my opinion:

Certainly the continued growth and change of technology has prompted a wide variety of new topics that the editors feel they must cover. Unfortunately, the reality of print-magazine economics dictates that something has to give if they expand coverage of new topics in the same size magazine. In light of the fact that I am presented with the same technology-coverage dilemma, I find that weighing the needs of the readers with the realities of being publisher leaves me with the same questions and no answers. It's not that I'm going kicking and screaming into the next millenium. I just need to feel that any editorial reorientation occurs for reasons other than being trendy.

A whole bunch of people felt the need to tell me to stick with my gut feelings and don't get trendy.

"I like Circuit Cellar as it is. I get information from the Internet but I get insight from your magazine." — R. Fellows

"There are very few magazines that offer any technical content.... Please do not take the path of BYTE." — R. Pryor

"...it's the only magazine I'm willing to pay for.... Mark me in the "don't change the format" category." — B. Raymond

"I need the kind of in-depth understanding that is actually useful.... That is why I buy your magazine." — T. Cantlon

This list could be considerably longer. There were many more comments. Invariably, the universal opinion was that you like the way we present things and "if it ain't broke. Don't fix it." OK, I get the message. But, I also know that technology is evolving and I can't be oblivious to it. There are lots of new topics I want to cover and many traditional ones I've been ignoring too long. I guess the only solution is for us to start another magazine!

Circuit Cellar Online magazine starts this month (visit www.circuitcellar.com/online for details). Unlike most print publications (or the trade magazines), *Circuit Cellar Online* is not a copy of our print publication. It's a completely new magazine (about the same size as the print magazine) with more of the high-quality editorial that you've come to expect from *Circuit Cellar*. We have new columnists with lots of great tips and tidbits. We have new departments and many more feature articles. Best of all, it's now twice as much *Circuit Cellar*.

Having another publication (albeit a virtual sister) gives me the pages to expand editorial coverage for the many things we've discussed like embedded Internet, signal interfacing, and design tips. You can expect a sampling of all these with the very first online issue. At the same time, having more pages allows me to get back to some of the things I like to call "traditional values." Yes, they've always been my pet interests, but I want to see more home automation, robotics, and basic tutorial articles in our pages again.

I also want you to know that all this couldn't happen without help. *Circuit Cellar Online* is hosted by ChipCenter (www.chipcenter.com). ChipCenter is an engineering portal owned by some very big players in the publishing, software, and distribution industries. Hosting *Circuit Cellar Online* provides them with instant technically-credible traffic. What we get is a super Internet connection and the security of not having to risk this venture alone. Of course, the success and continued availability of *Circuit Cellar Online* depends on its readers and the traffic you create while reading it. I trust that you will spread the word.

Finally, I don't look at this as accommodating evolving technology. I view it simply as the next step. The Internet can offer a world of enhancements and opportunities to magazines that know how to use them properly. *Circuit Cellar Online* is a virtual extension of the print magazine—not a replacement and not a copy. I recognize the loyalty among *Circuit Cellar* readers and I respect it. Rest assured, anything I ever do to change *Circuit Cellar* will always follow our prime directive.



steve.ciarcia@circuitcellar.com