

www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#111 OCTOBER 1999

EMBEDDED SOFTWARE

Open-Source Home Automation Program

Downloadable Web Server Software

How Does Windows CE Stack Up?

The Roots of Java



CIRCUIT CELLAR **ONLINE**

Double your technical pleasure each month. After you read *Circuit Cellar* magazine, get a second shot of engineering adrenaline with *Circuit Cellar Online*, hosted by ChipCenter.

— FEATURES —

PIC Remote Control with a Twist

Brian Millier

When he got his TV satellite dish, Brian discovered the freedoms of satellite network programming—and the limitations of the standard remote control. His '16F84-based IR remote puts him back in control.

Self-Help for Bugs in the Field: A Device-Initiated Upgrade Solution

Peter Gravestock

Upgrading firmware after a product is in the field can be challenging, especially when traditional methods can't be used. With a device-initiated architecture, Peter shows us how easy remote and automatic fixes can be.

Testing 1, 2

Part 3—Standards: Prepping Your Prototype

George Novacek

Coping with environmental challenges is a walk in the park in comparison to getting the power right. George explains the testing requirements as well as the changes you'll need to make for your product to succeed.

Ethernet Networking: Desktop to Device

William Peisel and Dick Caro

Afraid to venture into the gloomy tangle of Ethernet? William and Dick aren't telling tales when they say low costs and Internet integration capabilities will lead to Ethernet

networks at
all levels of industry
in the near future.

Resource Links

- **CAN-Controller Area Network**
 - **Digital Signal Processing**
- Benjamin Day*
- **Printed Circuit Board Software and Manufacturers**
 - **Pointers to Nanotech**
- Bob Paddock*

Test Your EQ

8 Additional Questions

— COLUMNS —

Considering the Details

I/O for Embedded Controllers—Part 1: Digital I/O

Bob Perrin

I/O, I/O, so off to work.... Designing generic controllers with only guesstimations of what the end-product I/O needs might be keeps Bob busy. Save yourself some work and look at the circuits covered in Part 1 of this series.

Lessons from the Trenches

Powering Your Memory

George Martin

Far too often, especially in programming, you can get away with what's less than best. Having personally tried all the less-than-best methods to write memory management routines, George invites you to listen in as he shares the best way to accomplish the task.

Silicon Update Online

Wire Wars

Tom Cantrell

The action's getting lively on the USB-versus-IEEE 1394 front as both sides fortify their legal defenses and rally

the troops.
Catch Tom's latest
report from the front lines (and
the 1394 Developers Conference).




WWW.CIRCUITCELLAR.COM/ONLINE
Table of Contents for September 1999

CONNECT YOUR PIC TO THE INTERNET

**INTERNET
PIC[®] 2000
CONTEST**

NOW, GETTING CONNECTED TO THE
INTERNET CAN EARN YOU CASH

www.circuitcellar.com/pic2000

- 12 MisterHouse**—An Open-Source Home Automation Program
Bruce Winter
- 20 The Java Virtual Machine**
Dave Lyons
- 24 A Versatile Timer/Synchronizer**
Brian Millier
- 30 Calling on the Standards**—Making Sure Your Modem Can Communicate
Arthur J. Carlson
- 56 Hands-On PIC Trainer**—Programming in Assembly
Jon Varteresian
- 60 IrDA Technology**—Part 1: An Overview
Hari Ramachandran
- 66**  **MicroSeries**
Rolling Your Own Microprocessor
Part 2: Design Application with the Rabbit-80
Monte Dalrymple
- 72**  **From the Bench**
Get SmartMedia—Part 2: Hands On
Jeff Bachiochi
- 78**  **Silicon Update**
'Net-in-a-Chip
Tom Cantrell

Task Manager Elizabeth Laurençot Open for Business	6
New Product News edited by Harv Weiner	8
Reader I/O	11
Test Your EQ	83
Advertiser's Index November Preview	95
Priority Interrupt Steve Ciarcia Spreading the Wealth of Knowledge	96

INSIDE ISSUE 111

EMBEDDED PC

- 38 Nouveau PC**
edited by Harv Weiner
- 40 What's in a Name?**
Windows CE vs. a Hard RTOS
Mal Raddalgoda
- 45** RPC **Real-Time PC**
Where in the World...
Part 3: Fighting the Wind with GPS
Ingo Cyliax
- 50** APC **Applied PCs**
GoAhead for Nothing—Getting the Server Started
Fred Eady

TASK MANAGER

Open for Business



When people are asked what they want to be remembered for, a common response is that they want to be remembered for having *contributed* something. There are countless ways this can happen.

Although you may be a philanthropist who donates a million dollars to a university so less-advantaged students can obtain a good education, or you may be a scientist who develops a vaccine that saves thousands of lives, most of us are going to be remembered for smaller gestures. I'm sure you've heard it said before, but let me repeat it here: it's the little things that count. The small kindnesses and efforts in day-to-day life are what truly matters.

It doesn't have to be entirely altruistic; you can benefit from your efforts now, whether it's on a financial or personal-satisfaction scale. And you don't have to contribute to everything everywhere. It's a good idea to choose your causes wisely and pick something that interests you. This month, I want to mention a couple options that may appeal to embedded software engineers.

In this issue, Bruce Winter presents an open-source home-automation program he developed. This Perl-based program can be accessed online, and Bruce encourages everyone to add to it (one programmer has already put together a GPS-based module that tracks car location from home) and to share the results. So, if you're interested in programming for home automation, check out www.misterhouse.net.

Speaking of open-source software, we've received a lot of great feedback about Ingo's Real-Time PC series on RT-Linux. There's so much interest these days in this POSIX-based OS! Linux operates under the GNU Public License, which means that the source code must be made available. Why would anyone want to give their source code away? Consider what Richard Stallman wrote in the GNU Manifesto (www.fsf.org/gnu/manifesto.html):

"I consider that the golden rule requires that if I like a program I must share it with other people who like it. ... GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace."

You may object, "Don't programmers deserve a reward for their creativity?" Richard's response: "If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results." So, don't say, "Oh, that GNU stuff is none of my business." Make it your business; find out how you can contribute!

Finally, if you enjoy playing with web technologies, consider helping out at Linux Online (www.linux.org/about/assist.html). You'll even earn a coffee mug for your work! Perhaps you'll just get a coffee mug out of the experience, but I bet you'll get a whole lot more.

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Sue Skolnick

MANAGING EDITOR

Elizabeth Laurencot

CIRCULATION MANAGER

Rose Mansella

TECHNICAL EDITORS

Michael Palumbo

Rob Walker

CHIEF FINANCIAL OFFICER

Jeannette Ciarcia

CUSTOMER SERVICE

Elaine Johnston

WEST COAST EDITOR

Tom Cantrell

ART DIRECTOR

KC Zienka

CONTRIBUTING EDITORS

Mike Baptiste George Martin

Ingo Cyliax Bob Perrin

Fred Eady

GRAPHIC DESIGNER

Jessica Nutt

NEW PRODUCTS EDITOR

Harv Weiner

ENGINEERING STAFF

Jeff Bachiochi

Ken Davidson

John Gorsky

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

ADVERTISING CLERK

Sally Collins

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://ftp.circuitcellar.com)

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

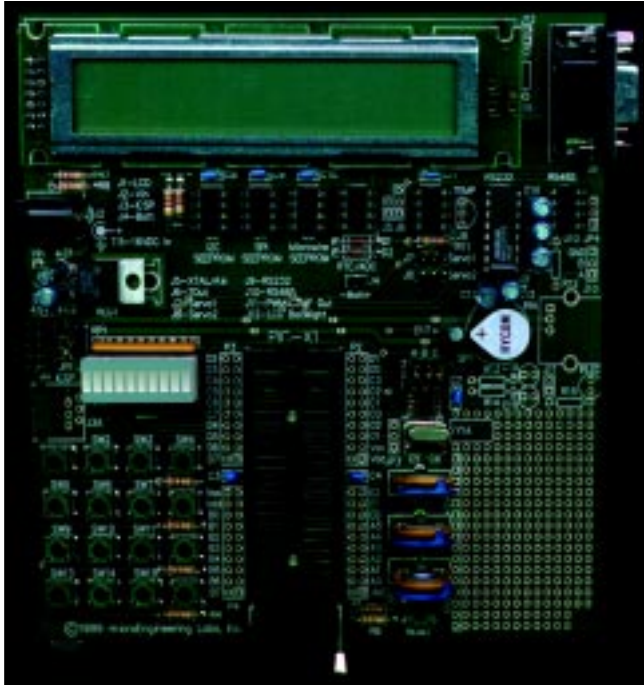
CIRCUIT CELLAR®, THE COMPUTER APPLICATIONS JOURNAL (ISSN 0896-8985) is published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (860) 875-2751. Periodical rates paid at Vernon, CT and additional offices. One-year (12 issues) subscription rate USA and possessions \$21.95, 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 EXPERIMENTER/LAB BOARD

The PIC-X1 is a testbed containing a prototyping area and most of the circuitry commonly used with PICmicros, including a 5-V power supply, crystal-controlled oscillator (adjustable from 4 to 20 MHz), and reset circuit. Application circuits include a switch matrix, potentiometers, LEDs, LCD module, serial EEPROM, real-time clock, temperature sensors, servo connectors, RS-232/-485 and IR interfaces, and speaker.

All of the I/O pins are brought out to headers next to a 40-pin ZIF socket. This allows connection to offboard circuits as well as allowing onboard circuits to be connected to other PICmicro pins, if desired. The PIC-X1 is designed to work with 40-pin PICmicros, but may be jumpered to work with smaller devices.

Projects such as calculators, LCD clocks, digital thermometers, LCD backpacks, tone dialers, TV remote controls, and so forth may be created using the PIC-X1. It includes in-circuit programming connectors so the resident PICmicro may be reprogrammed on-the-fly (requires flash device) using programmers that support this feature, like the EPIC PICmicro Programmer.

The PIC-X1 is priced at **\$199.95** for an assembled unit, **\$139.95** unassembled, or **\$49.95** for a bare PCB. A parts list, schematic, and example programs are also included.

microEngineering Labs, Inc.
(719) 520-5323 • Fax: (719) 520-1867
www.melabs.com

EMBEDDED CONTROLLER MODULE

Uni-Micro is a compact, 8051-compatible embedded controller module featuring a combination of memory, digital I/O, and analog I/O that is perfect for instrumentation, monitoring, and control applications. It provides enhanced JTAG-based serial programming that is six times faster than parallel programming. The module features 128 KB of flash memory, 32-KB boot/data flash, 256 bytes of RAM, 2-KB battery back-up scratchpad SRAM, 3000 gates programmable logic, and 54 programmable I/O pins.

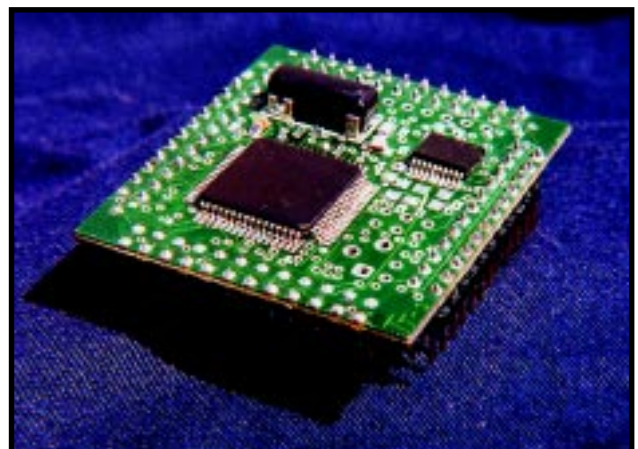
The eight I/O ports on the Uni-Micro can function as microcontroller I/O, CPLD I/O, address out, address in, latched address out, latched address in, data port, special function out, alternate function in, peripheral I/O, or open-drain outputs. Each I/O port is easily configurable by programming two sets of 8-bit registers, each bit controlling a single pin in the I/O port.

Serial programmability is required for first-time, in-system programming. Without it, first-time flash programming can only be accomplished using a conventional external EPROM programmer or boot code stored in the MCU ROM. Serial programming of the Uni-Micro module can be accomplished using a PC equipped with a JTAG board.

By integrating all the programmable logic, concurrent flash memory, data flash memory, SRAM, and I/O MCU core, the Uni-Micro Module has a power consumption of only 57 mA at 16 MHz.

Uni-Micro sells for **\$59**.

OS Systems Inc.
(908) 979-1885
Fax: (908) 979-3414
www.ossystems.net



NEW PRODUCT NEWS

FIBER-OPTIC MODEM

The **Model 232FLST Fiber-Optic Modem** provides the EMI/RFI and transient immunity of fiber optics in a transparent cable connection between PCs or other devices. It is ideal for data acquisition and other communication applications that are employed in electrically noisy environments.

A pair of the devices will enable any two pieces of asynchronous RS-232 equipment to communicate full or half duplex over two fibers up to 2.5 mi. RS-232 data signals at up to 115.2 kbps and RTS/CTS handshake lines are supported. Port powering allows operation without external power in most PC applications. Low-powered ports such as laptops may require an optional 12-V power supply.

The RS-232 connector is DB-25 female and the fiber cable connectors are ST style. The modem is only 4.3" × 2.3" × 1".

The modem sells for **\$154.95** (two are required to connect two components). An optional power supply is available for an additional **\$14.95**.



B&B Electronics
(515) 433-5100
Fax: (815) 433-5105
www.bb-elec.com

NEW PRODUCT NEWS

PHONE LINE SIMULATOR KIT

Ring-It! is a microprocessor-controlled telephone-line simulator that acts like a phone company central office and is used to test and demonstrate telephones, answering machines, fax units, voicemail systems, or modems. It supports E-911 training and caller-ID. An external audio jack is included for call-monitoring applications.

The simulator emulates an analog telephone line. For example, a connected telephone produces an authentic-sounding dial tone. Dialing a seven- or eleven-digit phone number with a touch-tone phone rings the device plugged into the test line. Busy signals and reorder tones can also be heard.

The caller-ID feature provides number-only or name/number messages. Five different test modes offer stan-

dard telephone-line emulation or special repetitive-cycle testing, including automatic ring-up. An LED readout

displays the DTMF digits that are dialed to verify use of touch-tone phones.

Ring-It! can be purchased assembled or as a kit that includes a PCB, electronic components, and a technical manual.

The factory-assembled unit (RI-001F) sells for **\$325**. The deluxe kit (RI-001D) is **\$205** and includes the caller-ID option and cus-

tom enclosure. Non-caller-ID kit versions (RI-001) are available starting at **\$149**.

Digital Products Co.
(916) 985-7219 • Fax: (916) 985-8460
www.digitalproductsco.com



READER I/O

KEEPING UP TO DATE

John Luo's article in the June issue (107) "Compact Optical Image Scanner" discussed a design built around the Texas Instruments TSL1401 linear photodiode array.

TI no longer supports this device, but Texas Advanced Optoelectronic Solutions (TAOS) has licensed the optoelectronic sensor product portfolio (which includes the TSL1401) from TI. Information can be found at www.taosinc.com or by calling (972) 673-0759.

Carl Strippoli

cstrippoli@taosinc.com

FINDING THE WAY AROUND SA

I enjoyed the GPS article by Ingo Cyliax in the August issue (109) and the earlier series by Do-While Jones (*Circuit Cellar* 77-78). Both did a great job of getting right to the meat of the matter. However, both articles may give readers a misunderstanding with regards to Selective Availability (SA) and position error.

The DoD implemented SA by diddling with the timing of the signals, among other things. This translates into a different range error for each satellite. These range errors, once crunched through the math, give the position errors noted in the articles.

Both articles gave the impression that the SA errors can be "subtracted out" by knowing the position error of a known fixed site. Here's the rub—the fixed site's error in position will only be the same as the observer's if both positions were computed from the same set of satellites. In the real world, any two GPS receivers may use different satellites if one particular satellite is locally obscured. In that case, the position errors are largely uncorrelated, and fixing the problem, as outlined in the articles, can actually make matters worse!

To fix SA, broadcast the range errors, not the position error, so the corrections can be made before the data is crunched. That's what the Coast Guard and other DGPS providers do. I hope this clarifies things.

John Wilson

Annandale, VA

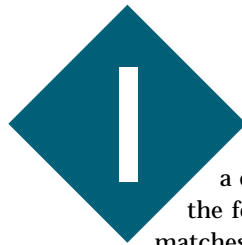
FEATURE ARTICLE

Bruce Winter

MisterHouse

An Open-Source Home Automation Program

If you thought that talking houses only existed in cartoons, it's time for you to meet MisterHouse. This Perl-based program can tell you who's on the phone, what room your kids are in, the latest weather forecast, and more.



Let's start with a quiz. Which of the following best matches your philosophy in home automation?

- Computers are too ornery and difficult to manage to be meddling in daily affairs of the house.
- Automating daily house chores with a computer is a fun and useful thing to do, but it can only reliably be done with a dedicated and specialized microprocessor.
- PCs are destined for more than just sending e-mail and playing golf; soon they will rule the world (or at least parts of your house)!

If you subscribe to the first philosophy, you're probably reading the wrong magazine. I bet a good number of you are members of group two. I used to be a member of this group when I started working with home automation (HA) with Steve's first Home Run HA system from the *BYTE* magazine days.

Over the years, as PCs got faster and cheaper, I've migrated to the third group. The nice thing about doing HA on a PC is that you have a much wider choice of programs and hardware to integrate. Every day it seems like there is some neat new program or device you can hook into a PC.

At my day job (designing chips at IBM), I've found great productivity in a programming language called Perl. About a year ago, I decided to write a Perl-based, open-source multiplatform HA program. I call it MisterHouse—which is easier than calling it a Perl-based, open-source multiplatform HA program.

This article gives a little background on MisterHouse and shows how I use it to log phone calls and announce who is calling over our house PA system.

WHY OPEN SOURCE

It used to be that when someone wrote a program and then made it available for free, it was called "free software." Often, only the binary for the program would be distributed.

Recently, the term "open source" has been promoted. There's an official definition at www.opensource.org/osd.html, but the basic idea is that the source of the software is posted on the Internet so that anyone can help work on it.

With new hardware becoming available daily and the amount of information on the Internet growing exponentially, the number of HA possibilities is growing faster than any one person can possibly keep up with. Using the open-source idea, if the program doesn't do what you need done, you can extend it and then share it for others to use.

In the few months that I have had MisterHouse (MH) posted online, I have reaped not only lots of good ideas from other users, but also bug fixes, code extensions, and completely new applications.

THE OS DECISION

Here's another quiz: Which is the better OS to do HA on? Windows and the great wealth of fun software available on it, or a Unix OS with its improved reliability and multitasking?

Can't decide? With Perl you can run on either OS! In fact, Perl is available on several different OSs, but I've only tested MisterHouse on Windows 95, 98, NT 4.0, Linux, and AIX.

By using the same underlying interface subroutines, you can keep the higher-level user event code platform-

Listing 1—This code will turn on an X-10 module set to address O5 and speak the time 30 min. after sunset. Note how the `time_now` function can handle time offsets and the `$Time_Now` variable will get substituted on-the-fly.

```
$pedestal_light = new X10_Item("O5");
if (time_now "$Time_Sunset + 0:30")
{
  set $pedestal_light ON;
  speak "I just turned the pedestal light on at $Time_Now";
}
```

independent. For example, on Windows, a serial port item will use the `SerialPort` module to read and write serial data using Win32 API calls to a serial port DLL. On a POSIX-compatible Unix system (Linux and most others), that call is translated by the `SerialPort` module to use POSIX `TERMIOS` calls.

Here's another example. When you ask MH to run a background process on Windows, the Perl `Win32::Process` module is used; on Unix, the `fork` function is used. The user code is the same, regardless of the platform.

VOICE IN, VOICE OUT

Much of the fun in HA comes when your computer talks and listens. Perl doesn't come with built-in voice software, but it talks to programs that do.

On Unix, you can implement text to speech (TTS) with the Festival speech synthesis system. Festival provides a server that MH can talk to via TCP/IP

sockets, and you can plug in different voices and languages.

On Windows, you can do both TTS and voice recognition (VR) using freely downloadable engines from Microsoft. These engines provide access via OLE methods, which Perl also supports. Several different voices are available, including some digitized voices that sound amazingly humanlike.

The VR is in a command-and-control mode (i.e., it will not recognize any arbitrary set of words; only the ones from a set of phrases you specify using the `MH_Voice_Cmd` object). This mode improves reliability over what you might get with a dictation mode, where any word can be expected.

PICK YOUR INTERFACES

All of the voice command objects you specify will be available with VR, as well as from the command line, a GUI Tk interface, and a web interface.

The Tk interface is built using Perl Tk widgets. These widgets enable us to create GUI objects with just a few lines of code. Photo 1 shows the default Tk interface.

The web interface can be used from any Internet browser, whether it's in your house or not. The interface uses a web-authentication password to allow remote control (e.g., go to `misterhouse.net` and you can take a look at the one on my house).

Using HTML templates, the web interface can be easily changed. The default web interface is shown in Photo 2.

Other programs communicate with MH through TCP/IP sockets or via commands in a file. The "Hardware Interfaces" sidebar shows the various interfaces that MH currently supports.

A QUICK LESSON IN PERL

All HA programs have some sort of programming language built in for event programming. Instead of implementing a new event-control language, I used Perl.

Perl is a powerful language and it can take quite a while to learn all of its tricks. But, by using object-oriented programming and extending it with HA-related functions, MH hides much of the complexity. Listing 1 is an event for turning on an X-10-controlled light half an hour after sunset.

Before I go on, I want to cover a few basic Perl syntax rules. All variables start with a `$`. Strings are quoted with either single or double quotes. Use double quotes if you want variables within the string to be substituted.

Commands begin with `#` and commands end with a semicolon. Do loops and conditional blocks start and end with `{ }`. There are two ways to do `if` statements—`if (test) {action}` and `action if test`. If tests are `==` (or `!=`) for numeric data and `eq` (or `ne`) for string data.

Local variables are declared with the function called `my`. Object methods (e.g., `set`) can be specified in two ways:

```
set $dishwasher ON; # Indirect
  object form
$dishwasher->set(ON); # Classic
  'object oriented' form
```

Listing 2—This code shows how incoming caller-ID data and outgoing phone numbers are spoken and logged.

```
$callerid = new Serial_Item('I');
$phonetone = new Serial_Item('O');

if ($caller_id_data = state_now $callerid)
{
  # On startup, old caller-id strings might be sent...ignore them
  if (time > ($Time_Startup_time + 15)){
    speak("rooms=all " .
      &Caller_ID::make_speakable($caller_id_data));
  }
  my ($cid_date, $cid_number, $cid_name) = unpack("A13A13A15",
    caller_id_data);
  logit("$Pgm_Path/../data/phone/logs/
    callerid.$Year_Month_Now.log", "$cid_number $cid_name");
  logit_dbm("$Pgm_Path/../data/phone/callerid.dbm", $cid_number,
    "$Time_Now $Date_Now $Year name=$cid_name");
}
# Log outgoing phone numbers
if ($phonetone_data = state_now $phonetone)
{
  logit("$Pgm_Path/../data/phone/logs/phone.$Year_Month_Now.log",
    $phonetone_data)
}
```

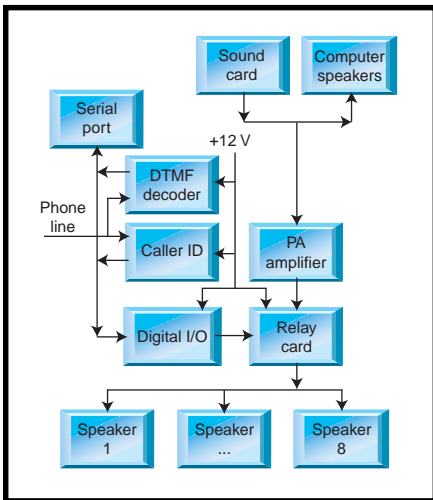


Figure 1—The computer's sound card drives the house PA speakers through a relay card controlled by a digital I/O interface. The phone line is monitored with DTMF and caller-ID cards.

By looking at the examples supplied with the MH package, you can learn what you need to know about Perl for its use in MH. Another good starting point is www.perl.com. The reference book of choice is *Programming Perl*. If you want to know how to do something in particular, *The Perl Cookbook* is excellent.

THE TALKING PHONE

Enough background, it's time for an example! If you're going to have your house talk to you effectively, you can either give it a really loud voice or you can use a house PA system with a speaker in key rooms.

But what if you want to have a conversation with your house in the living room while your spouse is trying to sleep in the bedroom? You can either get him/her a nice set of earplugs or you can insert a computer-controlled relay card between the PA amplifier and the speakers, as shown in Figure 1.

The digital I/O card is an inexpensive (\$30) PIC-based card available from Weeder Technologies. The relay card is a Universal Relay Card from Jameco (\$100). By modifying `pa_control.pl`, you can define which room is controlled by which relay.

Now you can direct speech to the room of your choice, using either of the following formats:

```
speak "rooms=bedroom Time to wake up";
speak(rooms => "all", text => "The laundry clothes are dry");
```

The other two Weeder cards shown in Figure 1 let you monitor outgoing and incoming phone numbers. The DTMF card detects any pressed phone keys, and the caller-ID card decodes the name and number of incoming calls. If you have a caller-ID-capable modem, you can use it instead. By sensing when the modem is not in use, MH can share this modem with other programs.

The code in Listing 2 monitors the data coming from these two cards.

The Weeder kits have a simple protocol that uses the first character of a string to indicate which card is sending data: "I" is for the caller-ID card and "O" is for the DTMF card. The MH code detects these codes and stores the incoming data in the `$callerid` and `$phonetone` objects.

The `$caller_id_data` string is parsed by the `Caller_ID::make_speakable` function, which changes the phonebook-formatted name into a more pronounceable name. For example, it changes "WINTER BRUCE LA" into "Bruce Winter" by swapping fields and dropping initials and abbreviations.

If the caller's area code differs from yours, this code uses a lookup table to add the city or state that the call is from. Optionally, it will use a user-defined rule to replace the received name with a more phonetically correct name or a prerecorded WAV file.

`$phonetone_data` and `$caller_id_data` are logged into monthly log files that can be retrieved and displayed by `display_calls`. The Tk interface for this feature is shown in Photo 3.

The caller-ID signal is sent between the first and second rings. In order to give MH a chance to decode and speak the caller-ID information before the phones start ringing, a ring morpher is inserted between the external phone line and the other phones in the house.

The ring morpher has the side effect of delaying the ring by one

Hardware Interfaces

CM11—The ActiveHome kit (\$50) includes a two-way X-10 CM11 interface, lamp module, and a couple of remote controllers.

Weeder kits—Weeder Technologies offers the following PIC kits, priced from \$30 to \$50 each:

- two-way X-10 interface
- 12-bit digital I/O (can be input or output, switch or button)
- 8-port analog I/O, 10-bit resolution
- caller ID: name and number
- outgoing DTMF phone monitor

These kits are good for PCs with a limited number of serial ports because they can all share the same serial port.

JDS—Currently, only the two-way X-10 interface has been tested.

WX200/198 weather stations—These weather stations are available from Radio Shack and mail-order catalogs, ranging in price from \$200 to \$300. They monitor indoor/outdoor temperature, humidity, barometric pressure, wind speed and direction, wind chill, and rainfall amounts.

Modem—Using the `is_available` method, a caller-ID-capable modem can be shared with other programs. It can also be used to send messages to pagers.

Ham radio TNC modem—Using a ham receiver attached to a radio modem (terminal node controller), MH can monitor packet radio traffic. This information includes locations and speed of cars equipped with GPS devices and weather information from home weather stations.

Other serial devices—The `Serial_Item` object can read and write to any serially connect device, such as an IR controller or robotic interface.

cycle. This delay allows us to hear the caller-ID announcement (e.g., “phone call from Bill Clinton”) at the same time the phones start to ring.

OTHER EXAMPLE EVENTS

Here are a few of the events we use to control our house. You can find them online (see the Software section).

I have window quilt curtains hooked up to pulleys attached to 24-V DC motors and a set of relays. Each curtain has two relays—on/off and up/down. These relays are driven either by local switches or by Weeder DIO outputs. `curtains.pl` monitors the outside temperature and sunlight levels using a Weeder analog kit, and opens or closes the south-facing curtains to maximize solar heat gain in the winter months.

The `deep_thought.pl` `trivia.pl` modules display and/or speak fun facts from various databases. It certainly makes breakfast entertaining!

`door_monitor.pl` monitors door openings and closings using magnetic reed relays connected to Weeder DIO

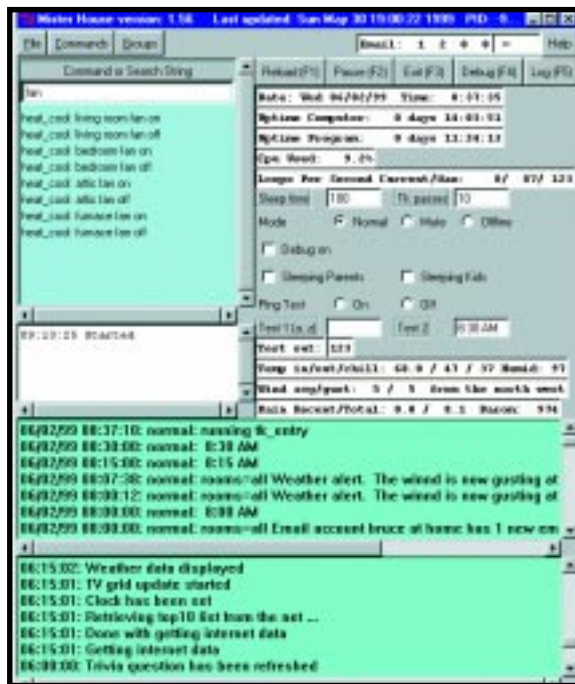


Photo 1—The bottom two frames are logs of what has been spoken and logged. The upper left frame is a searchable list of commands.

inputs. Do your kids leave outside doors open? This code issues a friendly reminder to close the door using the outside PA speaker. It will also close the garage door at night, but only after it senses no movement in the garage.

It also monitors movements in hallways and stairways using X-10

movement sensors. When doors open and close or people are sensed in key areas, I have MH play short, unobtrusive sounds so I can tell where people are going—sort of like a high-tech squeaky door!

`internet_data.pl` retrieves and processes pages off the web (e.g., Letterman’s top 10 list, local weather conditions). It also sets the clock according to an Internet-connected atomic clock.

The `internet_mail.pl` event uses a background process to call the `get_email` script, which checks who has sent you mail in specified e-mail boxes. After the process has finished, MH announces what new e-mail you received and from whom.

Using a user-specified IP socket port, `telnet.pl` lets you use a telnet program to log in, specify an optional password, and issue commands.

Like a kitchen timer, `timer_seconds` `timer_minutes` `timer_hours` periodically tells how many seconds, minutes, and hours are left on your timer.

A ham-radio enthusiast wrote the `tracking.pl` code. Using a GPS unit, a pair of radios, and TNCs (terminal node controllers), this code tracks and announces his car’s position and speed. Using an optional position file, it announces position with respect to various landmarks (e.g., “Dad’s car has just left the IBM parking lot” or “Junior’s car is parked at lover’s lane”).

This code does other nifty things with packet radio (e.g., monitor current weather conditions from the nearest packet radio-connected weather station).

Using the `get_tv_grid` script in a background process, `tv_grid.pl` collects customized TV programming from the Internet to local files. It adds a “program the VCR” button to the TV grid entry, so you can not only see what’s on but also instruct MH to create an event to start and stop the VCR at a certain time for the specified channel.

`weather_monitor.pl` `weather_wx200.pl` reads data from a WX200 or WM918 weather station. It then logs, displays, and optionally speaks the data.

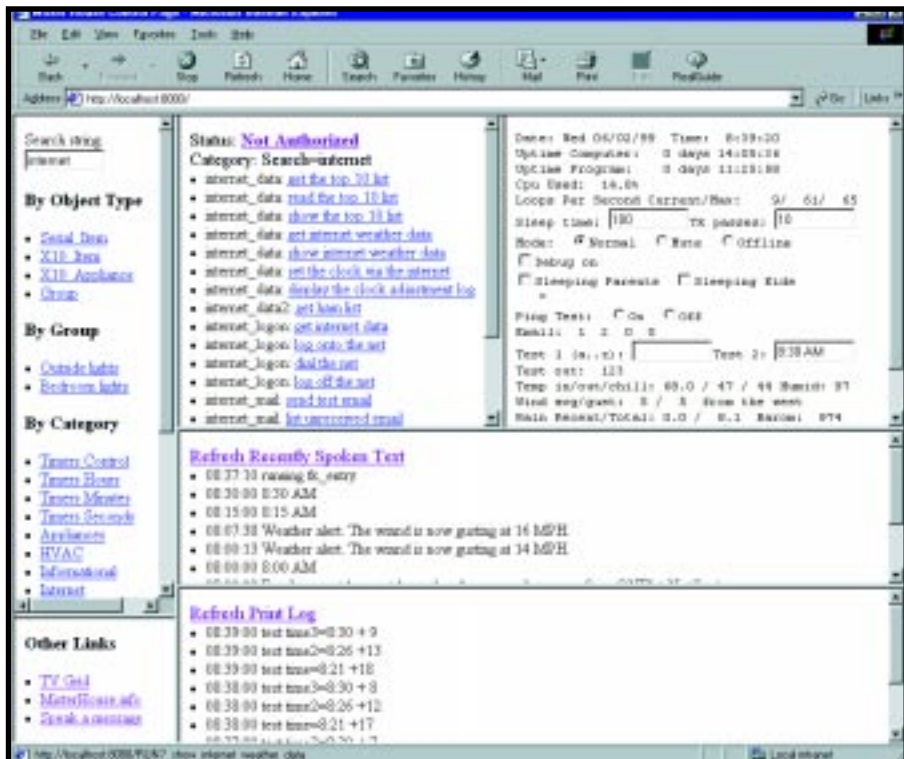


Photo 2—This mirrors the Tk interface, but includes an authorization password to allow secure access over the Internet.



Photo 3—The `display_callers` function shows incoming calls, outgoing calls, a list of all past callers, and a pick list of the logs for each month.

JOIN THE FUN!

Even if you don't have any of the hardware currently supported, you can still run MH and play around with the web, Tk, VR, and TTS interfaces.

The `install.html` file has the installation instructions. `mh.html` has documentation on the various objects, methods, and functions, and updates. `html` contains a list of updates.

`mh_src_###.zip` has all the source code, support files, and documentation. If you don't have Perl installed, you'll need `mh_win_###.zip` or `mh_linux_###.zip`, which have the platform-dependent compiled versions of MH.

Depending on how many events you load and if you have the Tk interface turned on or not, MH will use from 5 to 15 MB of memory. At ten passes per second, it uses about 30% of the CPU cycles of a 100-MHz Pentium CPU.

So, dust off that old PC, give your house a little personality, and come join the open-source movement and make the world a better place! Or at least have your house train your kids to keep the back door closed. ☺

Bruce Winter designs integrated circuits as a senior engineer at IBM in Rochester, MN. This has nothing to do with home automation, but it sounds impressive in a bio. You may reach him at bruce@misterhouse.net.

SOFTWARE

Software for the MisterHouse application is available via the *Circuit Cellular* web site, misterhouse.net, and misterhouse.webjump.com.

RESOURCES

T. Christiansen and N. Torkington, *The Perl Cookbook*, O'Reilly, Sebastopol, CA, 1998.

MH events for author's house, misterhouse.net/mh/code/Bruce
Mailing list, www.onelist.com/subscribe.cgi/misterhouse, www.onelist.com/archives.cgi/misterhouse

L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl*, O'Reilly, Sebastopol, CA, 1996.

SOURCES

Digital I/O card

Weeder Technologies
(850) 863-5723
www.weedtech.com

Universal Relay Card

Jameco
(800) 536-4316
(415) 592-8097
Fax: (415) 592-2503
www.jameco.com

TTS and VR engines

www.microsoft.com/iit/download/speechengines.htm

Text-to-speech with Festival

www.cstr.ed.ac.uk/projects/festival

ActiveHome kit

X10 (USA), Inc.
(800) 675-3044
www.x10.com

Interfaces

JDS Technologies
(858) 486-8787
Fax: (858) 486-8789
www.jdstechologies.com

FEATURE ARTICLE

Dave Lyons

The Java Virtual Machine

Is dependability more valuable than speed? When James Gosling created Java, he thought so. Dave agrees and wants to show us the Java Virtual Machine and how easy and how convenient working with Java can be.



In 1991, James Gosling was developing a new programming language for use in intelligent consumer devices. He started with C++ but realized that no number of extensions to the language would meet the requirements. Because intelligent devices need a wide variety of OSs and processor families, the new language needed to be machine independent.

Gosling designed a language where the source code was compiled into machine code targeted at an abstract processor. Applications wouldn't have to be recoded or even recompiled. Only the code that implemented the abstract processor would have to be ported. Gosling named his new language Oak after the tree outside his window.

Oak was object oriented and programs were composed of classes and interfaces. An interface defines a set of methods that a class can implement. This approach allows much of the functionality of C++'s multiple inheritance without the runtime overhead.

The Oak syntax was similar to C and C++. A handful of basic types were provided including byte, short, int, long, char (an unsigned 16-bit value), float, double, and boolean.

The sizes of these basic types were explicitly defined so that Oak applications would behave consistently across

various platforms. Floating-point functionality was based on the IEEE-754 standard and was explicitly defined so programmers could precisely predict behavior.

In developing Oak, Gosling determined that reliability was more important than speed. Consumers expect appliances such as VCRs, telephones, and toasters to work. They wouldn't tolerate a product that needed to be rebooted. Several aspects of Oak's design reflected the reliability issue.

Objects in Oak were allocated using the `new` operator but were only returned to the heap when the garbage collector determined that there were no longer any references to the object.

Oak also avoided the problem of using variables before they were initialized by rigorously checking to be sure that all variables inside an Oak method were set beforehand. A used-before-set condition resulted in a fatal compiler error. And, perhaps the most significant reliability-related change—there were no pointers in Oak.

OAK BECOMES JAVA

As time passed, Sun's Green project (later spun off as FirstPerson, Inc.) was shut down when no market could be found for the technology. At a group meeting, Gosling, along with other Sun notables such as Bill Joy and John Gage, recognized that the Internet had recently been made more accessible by way of web browsers and that it was a good fit for the technology embodied by Oak.

The Internet moved media such as text, audio, and graphics to various platforms. With Oak, application code could be moved in the same way.

The goals Gosling had set out to accomplish fit perfectly with the way applications were being written, delivered, and used on the Internet. Although the discovery was quite accidental, it appeared that the Oak/Internet combination had potential.

Oak was eventually renamed Java and a web browser called HotJava was written in the language. This browser, besides being able to display standard web pages containing text and graphics, also permitted small applications (applets) to be embedded in pages.

Sun decided to make Java and Hot-Java freely available and in doing so, set in motion a new style of computing that seems to be gaining momentum.

ABSTRACT PROCESSOR

From the beginning, a number of companies licensed the source to Java and began porting it to their platforms. This effort involved more than simply porting the abstract processor, known as the Java Virtual Machine (JVM). Although the term JVM describes the Java interpreter, it also describes the entire platform.

The comprehensive set of APIs defined as part of Java also needed to be ported to the new platform. These APIs defined standard mechanisms for graphics and windowing, file storage, and network access.

Figure 1 illustrates the Java runtime environment. User programs and the bulk of the Java API reside in class files which the class loader checks for correctness before use. The execution engine (also called the JVM) interprets the code loaded from the class files.

The JVM is stack-based and defines four 32-bit-wide registers—the program counter, top of stack, current stack frame, and local variables pointer. Instructions for the JVM consist of a one-byte opcode followed by zero or more operands. The instructions vary in complexity from simply pushing a constant onto the stack to allocating a multidimensional array of objects.

The JVM is also able to make calls to native methods, which are implemented in a native system language, typically C or C++. Native methods are used to interface to the underlying OS, but they can be used in other situations as well.

For example, native code might be used by an application for computationally intense code such as decoding motion video. Native methods can also be used to create a Java wrapper for large, existing code bases.

Although a certain number of native methods are needed for almost all Java implementations, they should be used with discretion. A Java application that relies directly on the native methods can no longer be run on all Java platforms without porting the native code.

This portability (or as Sun calls it, “Write once, run anywhere”) is key to Java’s appeal, so Sun created a program called 100% Pure Java. To be certified 100% Pure, an application can be made up of only Java bytecodes.

THE PERFORMANCE KEY

It’s not hard to imagine that the key to executing Java code quickly lies in the bytecode interpreter. There’s a good deal of supporting code involved in things like garbage collection, class loading, and interfacing with the underlying operating system, but most of the time is spent interpreting the bytecodes. Because of this, most efforts to improve Java performance have focused on speeding up the interpreting process.

The simplest way of speeding up the bytecode interpreter is to hand-code the interpreter loop in native assembly language. This approach often results in a smaller interpreter, which takes advantage of modern microprocessor cache as well as instruction pipelining. Writing the bytecode interpreter in assembly language can increase the speed twofold.

A larger performance gain can be achieved by using a just-in-time compiler (JIT). A JIT converts the Java bytecodes to native machine code at runtime, resulting in speed increases on the order of 10×, which is helpful for computationally intense code.

However, a JIT can result in a speed decrease when there’s a large amount of code that isn’t executed frequently. This setback is because of the overhead of converting the Java bytecodes to native code.

A refinement on the JIT approach is Sun’s HotSpot technology. The HotSpot interpreter identifies areas of code that are frequently used and focuses its attention there.

Methods that are frequently called can be inlined and the code inside methods can be optimized because much more information about a program is available at runtime than at compile time, which is where optimi-

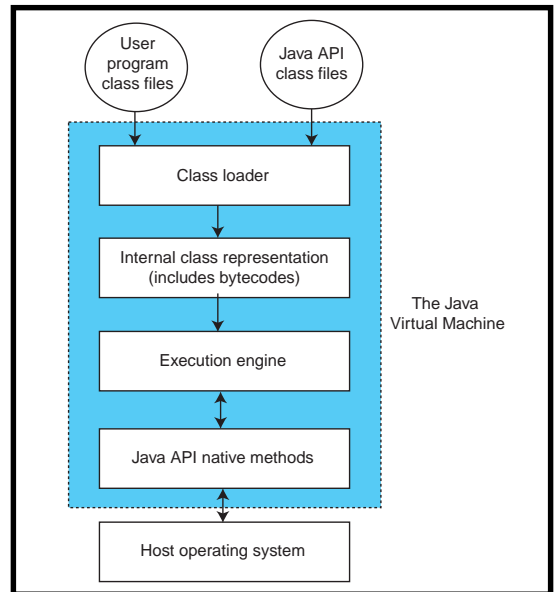


Figure 1—The class loader converts Java code files into a form usable by the execution engine. The underlying operating system services such as networking and graphics are accessed through native methods in the Java API.

zations have traditionally been done. Although no concrete performance numbers have been published, Sun hopes HotSpot will enable Java performance to approach that of C++.

HARDWARE CONSIDERATIONS

The bytecode interpreter plays a large role in determining the performance of Java on a particular platform, but the underlying hardware also plays a significant role. A processor with a high clock rate may not perform as well as a slower processor with a larger cache.

Although true in general, executing Java code magnifies the difference because so much time is spent in the relatively small bytecode interpreter. The effect would not be as pronounced if a JIT or HotSpot-based interpreter was used. A bytecode interpreter that fits entirely into cache could dramatically affect Java performance.

Floating-point support can also be significant in determining Java performance on a platform. Processors without floating-point hardware process the Java floating-point instructions up to 10× slower than similar processors with hardware floating-point support.

The problem is compounded by the fact that processors lacking floating-point support are typically lower horsepower chips aimed at consumer devices. Application programmers should take

note of this situation. If a piece of code is targeted at consumer devices such as those based on the new PersonalJava platform, floating-point usage should be kept to a minimum.

ULTIMATE JAVA PLATFORM?

It may seem that the obvious answer to the Java performance question is to build a JVM in hardware. Sun has done just that with their picoJava architecture, which uses Java bytecodes as its native language.

The architecture is also designed to optimize access to the Java stack. Even though these Java chips may seem like the obvious choice, many consumer electronics manufacturers that are building Java into their devices are choosing other microprocessor families.

In the consumer electronics business, a difference of a few dollars in the price of a microprocessor can make a huge impact on the economic viability of a consumer product. If one processor includes built-in peripheral devices such as serial ports or an LCD driver at a lower price than a competing chip, a difference in execution speed may become inconsequential in comparison to the difference in price.

Java chips may offer the best performance, but when a technology like HotSpot narrows the gap between hardware and software implementations, choosing a microprocessor for a Java-based device becomes complicated. In the end, the success of Java-based devices may be determined by factors besides just the technical aspects. ☒

Dave Lyons is a software architect at Microware Systems. During his 12 years there, he has worked with file systems, booting mechanisms, and development tools. Currently, Dave is focused on the company's port of PersonalJava for OS-9. You may reach him at davel@microware.com

SOURCE

Java Virtual Machine

Sun Microsystems
(800) 786-7638
(408) 276-5200
Fax: (408) 276-0633
www.java.sun.com

FEATURE ARTICLE

Brian Millier

A Versatile Timer/ Synchronizer

You may not be able to teach an old dog new tricks, but you can use Visual Basic to train a group of counter/timer devices. Brian shows us how to make a freestanding device that can read, write, load, and even learn new tricks.



Since the '50s, science fiction has featured robots that look human or exhibit human traits. Such devices rarely existed outside of Disneyland but are now becoming commonplace on the factory floor. These specialized robots can be trained by human operators to do assigned tasks with little outside intervention.

This project follows the same philosophy as the industrial robot, just on a smaller scale. At its heart is a group of counter/timer devices controlled by a PIC16F84. Training the device (i.e., setting the configuration and timing parameters of all the timer modules) is done with a Windows-based program running on a PC.

After you choose a configuration, you download the data via a serial port to the PIC16F84. The device is now a freestanding unit, producing all the timing signals, synchronized triggers, and everything else needed for the application. Only if you needed to

change the timing parameters or configuration must the device be reconnected to the PC.

The CTS9513 was designed to provide a master timing synchronizer and pulse generator for a laser-based research instrument. Like most custom research equipment, the actual requirements weren't well-known in advance. In the past, similar situations have led to time-consuming redesigns. This time, I built a general device using two CTS9513 counter/timer devices (see Figure 1).

Each CTS9513 contains a five-stage scaler and five 16-bit timing modules that can be programmed to handle divide-by- n , delay, or PWM functions. Many of the clocking and gating interconnections among the timing modules can be configured in software and many additional interconnection permutations can be handled by jumpers.

In addition to the two scalable quartz oscillator clock sources is a power-line zero-crossing trigger (60 or 120 Hz), as well as provision for an external clock source. Eight of the 10 timing modules are buffered and routed to BNC outputs that can be set to produce either active-high or active-low outputs (see Photo 1).

Although the CTS9513 is rated at a maximum count rate of 20 MHz, I chose a 16-MHz quartz oscillator, providing 0.0625- μ s resolution (adequate for my application). By substituting an AMD AM9513, you can use a quartz crystal instead of an oscillator module, but the maximum clock frequency is 7 MHz.

In *Circuit Cellar 78*, I described a portable pulse generator based on an AM9513 that generated various user-

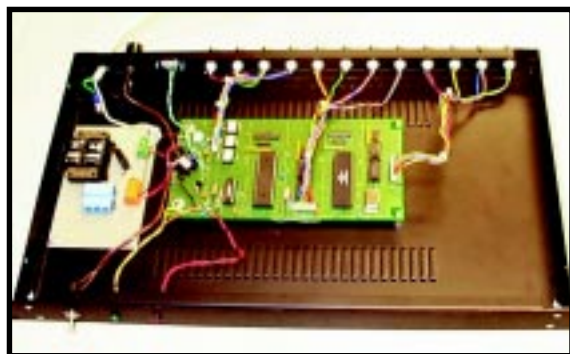


Photo 1—Without all the connectors along the rear panel, the circuit would fit into something smaller than a 19" 1U rack cabinet.

defined pulse trains, but the timer configuration was fixed in firmware.

This time, I used the '16F84 as the controller chip. This \$5 micro has enough I/O lines to control two CTS-9513 timers, a serial link to the host PC, and a status LED.

The PIC's biggest selling point has to be its flash memory, which lets it hold all the configuration and timing parameters for two CTS9513s in nonvolatile memory. So once programmed, the unit works as a freestanding device. Flash memory lets parameter changes be made quickly.

Developing the program code went quicker because I didn't have to wait for a UV erasure of the device with each code modification. Although the '16F84 doesn't contain a UART function, it's fast enough to implement a 9600-bps serial input function in software.

The CTS9513s go for around \$30 and the total chip cost for the circuit is about \$80—quite reasonable considering the unit's capabilities.

HARDWARE DESCRIPTION

I broke up the circuit diagram into two parts. Figure 2 shows the controller, timer ICs, power supply, serial port, and everything else needed. Note that I used one AM9513 and one CTS-9513 to show the different clock circuit needed for each device. Apart from that, programming the devices is the same. Use whatever device you have available.

Figure 3 shows the input clock/gate selection circuitry as well as the output buffering and polarity selection circuits. Depending on your application, the circuitry shown in Figure 3 could be different. For example, the optocoupled inputs and 50-Ω output drivers (necessary in the noisy environment my unit encounters) may not be necessary in your application.

Q1 and the associated parts convert the RS-232 serial data from the host into TTL levels, which is fed into RA0 of the '16F84. Software routines in the micro convert the serial data-stream into parallel form. Notice that



Photo 2—Configuring each timer section is accomplished from the main screen of the Visual Basic host program.

there's no hardware handshaking. The host application paces its transmission of parameters to match the '16F84's response time, which is limited by the flash-memory write cycle time.

I tied the Rx and Tx pins of the DE-9 socket together so all incoming characters are echoed back to the PC. This setup made it easier to troubleshoot early in the design, when I was using a terminal program to send the commands and parameters. My problem was remembering to send the upper-case command mnemonics the PIC firmware expects!

Port pins RA1-RA3 provide the control signals for the '9513 devices. The CTS9513 uses a data-pointer architecture to access its 30+ registers, so only a *WR line is needed for each device (along with a shared command/*data line). This design never reads the '9513's registers, so the *RD line is tied high and the RA4 line drives a status LED, which is described later.

Port B is programmed as an output port to provide the 8-bit data bus for the '9513s. Notice that these devices actually have a 16-bit bus. The upper eight bits must be tied high and a command sent to the device to enable it to operate in the 8-bit mode.

The power supply is conventional except that the bridge rectifier acts as both the full-wave rectifier for the power supply and provides 120-Hz unfiltered DC to optocoupler U3 for the zero-crossing detector circuit. A separate diode, D2, is used if 60-Hz zero-crossing is needed.

The zero-crossing detector is a simple MOC5009 opto device with a Schmitt trigger on the detection side to reject AC line noise. As configured, this circuit provides a trigger whose rising edge leads the AC zero-crossing by about 0.4 ms (slightly dependent on the AC line amplitude). Although a high-gain comparator can be used to sense the true AC line zero-crossing, it would be susceptible to line noise, which this circuit isn't.

By feeding the PIC's *MCLR from the *RESET output of a LM2925T, I delayed the PIC's startup by 0.5 s after powerup. This setup ensures stability when the CTS9513s are loaded, which is important because they are only loaded at powerup. I used the LM2925T, but you can use a conventional '7805 and put a capacitor to ground on the *MCLR line instead.

The gating and output circuitry in Figure 3 is pretty straightforward. I chose '74128 50-Ω driver devices because they provide TTL-level signals with lots of drive for opto-couplers and are more robust when driving long cables that pick up inductive noise. The 74LS86 XOR gate allows for output polarity selection, as well as taking care of the inversion that occurs in the '74128 buffers.

THE PIC FIRMWARE

The PIC16F84's sparse instruction set, although adequate for my project, makes programming in assembly quite tedious. I just can't force myself to use C with small micros containing only a few kilobytes of memory

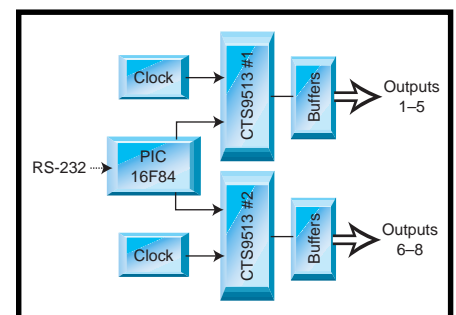


Figure 1—It doesn't seem fair that a modest little PIC chip is telling those two counter/timer chips what to do.

for apps involving time-critical bit manipulation of ports.

The firmware consists of:

- 9600-bps serial data input routine, using a software UART
- simple command mnemonic parsing routine
- data EEPROM writing/reading routines
- '9513 setup and register-loading routine

When writing code for a small device with few debugging facilities, it makes sense to develop general software routines to assist in debugging. So, along with the necessary program code, the firmware also contains

routines to write to any data EEPROM location, read any EEPROM location by sending it out to Port B, and manually load any of the '9513 register(s) via serial port.

Because of limited RAM space in the '16F84, the 58 parameter bytes in the '9513 that are downloaded to the PIC must be stored immediately to the PIC's 64-byte EEPROM data memory. The EEPROM write cycle is around 10 ms, so the host PC application must "pace" the download to allow for this condition. I wanted a 9600-bps routine for other projects, which explains my choice of 9600 bps rather than a much slower rate, which wouldn't have needed any pacing.

The correct operation of the '9513 timer modules depends on their receiving the correct parameters at powerup, so I wanted to incorporate some checks to ensure that the EEPROM data had not been corrupted.

The PIC16F84 has a number of features built in to prevent EEPROM data corruption, but I'm a skeptic. Therefore, the 58 bytes of data that make up the '9513's parameter/configuration data are supplemented by a single checksum byte, which is calcu-

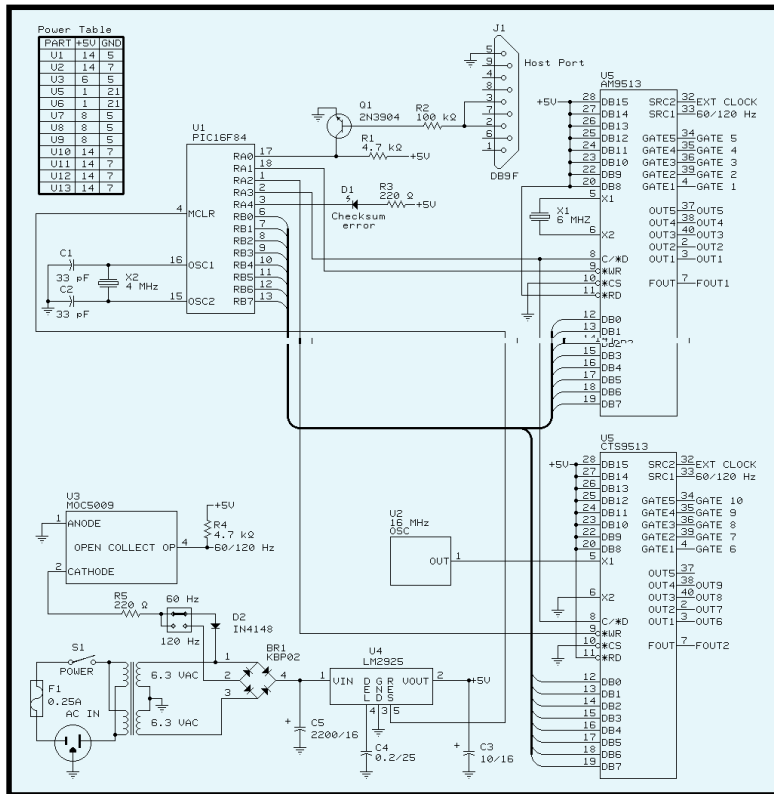


Figure 2—In my prototype, one each of the CTS9513 and AM9513 chips were used to demonstrate their compatibility (translation: I had one of each in my parts cabinet).

lated by the host application program and sent along with the rest of the downloaded data.

Immediately after a download (and at each powerup), a checksum calculation is done on the data EEPROM array. If an error is found, the checksum LED remains lit (it's turned on during a download and flickers during the checksum testing at powerup).

USER INTERFACE

I'm a big fan of Visual Basic and still find that the professional V.3 handles my needs without generating huge program files and loading too many support files into the end user's Windows directory like the new versions do.

Photo 2 shows the program in action. The first time the program runs, a prompt appears to specify which COM port the project is connected to. That's about the extent of setup requirements.

When configuring a '9513, I recommend that you thoroughly study the datasheet (available on the Celeritus web site). Although you can still get the AM9513, it has been discontinued so its databook is no longer available. However, the AM9513 is similar to the CTS part so the CTS datasheet can

be used for configuration purposes.

The first stage of setup is to initialize the device by putting it into 8-bit mode and to prepare the five counter/timers for loading. Because this process is done by the PIC firmware, you needn't be concerned with it. If you're using a '9513 and are unfamiliar with the part, you might want to look over my PIC code listing.

Second, configure the Master Mode register (using the frame at the right in Photo 2) to select binary/BCD division ratios for the scaler, select an Fout source, and so on. Because there are two '9513 devices in this project, there are buttons to

switch between the two Master Mode registers. Press the "save MM configuration" button when you are finished.

Certain bits of the Master Mode register must be set properly for the PIC firmware to work. The host PC program doesn't allow access to these bits.

Last, configure as many of the nine counter/timer modules as necessary. Even though there are 10 modules in the two '9513 devices, there's only enough room in the '16F84's data EEPROM for nine module's worth of parameters.

The output buffers are in a quad configuration, but I only wired eight of the timers to output buffers. Module nine can only be used to feed another timer module.

To complete a counter/timer configuration, select a module from the pull-down menu. Notice that all of the selections are grayed out until the selected module's configuration is saved by pressing the "save current configuration" button.

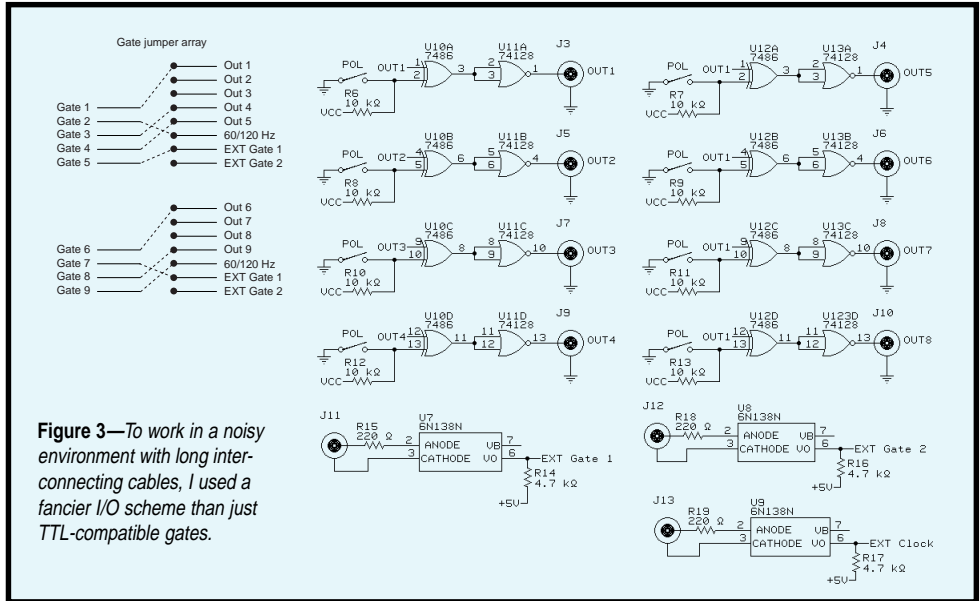
When the proper module is displayed, choose the desired mode. There are 24 modes, but many of them aren't applicable in a circuit such as the one used in this project.

Modes that perform a divide-by-*n*, generate a delayed pulse, or perform PWM functions are applicable. When selected, a message box describing that mode appears. Pick a mode, then choose the clock source and active edge for the clock.

If the selected mode requires gating, a gate frame appears. Not all gating options apply to all modes, so only the applicable ones are enabled.

For all of the modes, a count must be entered into the Load Register window. The range is 0–9999 in BCD and 65536 in Binary mode. Certain modes also require you to place a count in the Hold Register window.

Next, choose the output mode. Only High TC pulse and TC-toggled are useful in this design. High TC produces a pulse (width = clock) at each terminal count. The TC-toggled mode changes the state of the output pin at each terminal count, which produces either a square wave or a



PWM waveform, depending on the mode you have chosen. At powerup, all output flip-flops are reset, so the sense of the PWM waveforms is consistent at each powerup.

The two crystal or oscillator modules you choose for each '9513 device, are entered as the master clock fre-

quency. Setting these is only necessary if you plan on using the figure displayed in the “Time per single count” window for timing calculations.

After the Master Mode registers are set up, and all necessary counter/timer modules configured, the data can be

downloaded to the PIC by clicking on that menu item. This process takes a few seconds during which the checksum LED will light. If this light stays on after the download, then there was a problem with the data transfer.

Test the unit by selecting the "Load 9513 Timers" menu item or powering the unit off and on again. During any powerup, the checksum light flickers briefly if all's well or remains lit if the data EEPROM is corrupted.

Although the configuration and timing parameters were loaded into the PIC's nonvolatile EEPROM, the data displayed by the host PC program is only loaded into RAM and will be lost when the program is exited.

I recommend saving this setup to a disk file via the File Save menu. Different file names would enable you to store various configurations.

WRAP-UP

It occurred to me that other peripheral devices may need to be configured in a particular way at powerup to perform useful functions without

further computer control. The flash-memory-based PIC16F84 would probably serve this function well.

The 8253/8254 triple timer, which is quite inexpensive because it was designed into early IBM PCs and clones, would be another good option. CTS also makes a higher-speed version of this device for about \$7.

Another candidate might be Telco's cross-point switch ICs. For example, the PIC would set up a complex switching arrangement (which doesn't change frequently) PIC at powerup but easily reconfigured using a serial data link.

With the various technologies available today, there's certainly no shortage of choices and options for projects like this one. 📧

Brian Millier has worked as an instrumentation engineer for the last 17 years in the chemistry department of Dalhousie University, Halifax, NS, Canada. He also operates Computer Interface Consultants. You may reach him at brian.millier@dal.ca.

SOFTWARE

The source code for this article is available for download via the Circuit Cellar web site. The code is also available for download at www.bmillier.chem.dal.ca.

SOURCES

CTS9513-2

Celeritous Technical Services Corp.
(800) 687-6510
(806) 783-0904
Fax: (806) 783-0905
www.celeritous.com

PIC16F84

Microchip Technology, Inc.
(800) 437-2767
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

Programmed PIC16F84 \$15
Brian Millier
31 Three Brooks Dr.
Hubley, NS
Canada B3Z 1A3

FEATURE ARTICLE

Arthur J. Carlson

Calling on the Standards

Making Sure Your Modem Can Communicate

Modem testing may be about as exciting as putting up highway mile markers in Nevada, but in the long run it will be worth your time. Get in on this hand as Arthur deals out information on modem testing standards.



Modem testing can be bewildering to OEMs whose primary business doesn't include designing modems. Fortunately, modem test standards TSB 37-A and TSB 38 provide consistent and uniform criteria to compare the performance of modems. Draft digital standards PN 3857 and PN 3856 extend these concepts to PCM modems.

In this article, I describe various facets of performance testing and review the role of testing in the modem design process. I also give information on the statistics of the TSB 37-A/PN 3857 network models and an outline of the role of TSB 38/PN 3856. The article includes examples of typical modem performance curves and describes how they can be interpreted.

Standards tell part of the story, but ad hoc testing and customer feedback are important components of a well-designed modem product. OEMs should expect their modem vendors to follow a comprehensive testing program that includes all three types of testing.

FOR EXAMPLE,

Your latest assignment is to add an embedded voiceband modem to your company's nifty hand-held Personal Peppy Puppy (PPP) product in order to leap ahead of the competition.

You study brochures from every modem vendor in the Thomas Register looking for the right fit for your unique requirements. Each vendor heralds their modem's advanced technology, reliable operation, unsurpassed performance, low cost, instant availability, attractive color scheme, and easy payment plan. They all sound good.

All the demo boards behave about the same. The so-called 56-kbps modems connect at speeds between 38,666 and 46,666 bps at different times of the day, probably depending on unknowable random phenomena in the telephone network and your ISP. Besides, modem speed isn't important because the Internet is pretty slow anyway.

You recommend two vendors, purchasing negotiates a deal, and you start your design. The advertising slogan, "Put more bark in your Peppy Puppy," becomes the company's rallying cry. You enter the beta phase two weeks ahead of schedule. You're a hero.

That's when all hell breaks loose. Every beta site north of the Rio Alpha and west of the Chappanoes reports connection failure rates exceeding 25%. The CEO wants to talk to you. How did you select those modem vendors, anyway?

You wake up in a cold sweat. It's that same old nightmare.

ROLE OF MODEM TESTING

Modem testing is the activity everyone loves to hate. But if you're going to avoid the Peppy Puppy nightmares, you'd better do your homework.

Modems are complicated and the networks they run on have complications of their own. Even Murphy has no idea of how many things can go wrong, but your customers will find out soon enough if you're not diligent.

You have no choice but to design your products as well as you can and then test the heck out of them.

Boring standards such as V.90 and V.34 dictate the basic requirements of a modem design. But conforming to those standards is only the first rung of the product design ladder. Designers must subject new V.90 modem modules to a variety of tests before they can confirm that the design is worthy of production.

Several levels of requirements must be met. Your modem must properly interwork with every modem already on the market. Performance must be as good as the existing modems. And, your modem must be

able to operate in various telephone network environments that can include PBXs, digital links, analog links, satellite hops, undersea cables, and hot coffee spilled on the operator's lap.

Indeed, the task of testing can be quite formidable. Additionally, if Company X tests its modems using test procedure A and if Company Y tests its modems using test procedure B, how will the OEM judge whether modem X is better than modem Y?

TESTING STANDARDS

Enter testing standards. The Telecommunications Industry Association (TIA) has worked for several years to evolve a rational set of modem test procedures. The latest official test documents were published in 1994, and an update is in the works.

These documents introduced the concept of network coverage. When engineers use the models and procedures defined in these documents, the results are straight-up comparisons of a given modem against any others.

After the testing in simulated environments is finished, there are still those pesky situations that come up when a new product ventures into the field. Of course, there are no standards for testing at this level, but an OEM would do well to have assurance from the vendor that the modem has been successful in a wide range of real-world environments.

Once reports of field-related problems begin to come in, the modem vendor must have the know-how to understand them and make the appropriate changes in the modem design. Interesting wrinkles sometimes pop

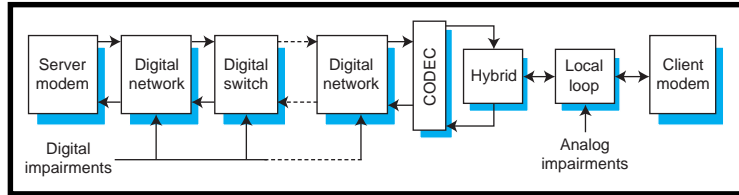


Figure 1—Only one D/A conversion exists between the server and the client. Each digital stage may add digital impairments such as RBS or a digital pad.

up. After all not all design decisions are made for good technical reasons.

Some modems produced by large modem manufacturers don't conform to the mandated standards. When that happens, smaller vendors have to come up with workarounds to successfully interwork with the big-name modems.

I recall the time my company's V.32bis modems experienced problems at a certain ISP. The ISP was using modems supplied by a Well-Known Modem Company (WKMC). Our call-mode modems couldn't connect with the answer-mode WKMC modems and I was assigned to solve the problem.

I discovered that WKMC's modem had a subtle but definite mistake in its training sequence. Usually the error wouldn't cause difficulty, but one of the design choices in our modem stepped right into the little crack exposed by WKMC's error.

I called a senior engineer friend of mine at WKMC who confirmed that their modem had a little mistake in the training sequence. He referred me to a junior engineer who had written that section of code. The junior engineer and I agreed that there was a problem, and it was my understanding that he would work on it.

This was too simple for management to comprehend though, and my VP soon called me in to ask what was going on. He had received a phone call from the VP of engineering at WKMC. To better explain the situation, we scheduled a conference call involving my VP, WKMC's VP, and myself.

To no one's surprise, WKMC's VP was not impressed and he refused to acknowledge that there was anything

wrong with his modems. Certainly he wasn't anxious to upgrade the thousands of modems that WKMC had in the field.

After the call, my VP suggested that I modify our training sequence to work around WKMC's

error. It seemed like a good design decision under the circumstances.

INTEROPERABILITY

A walk down the aisles of your favorite electronics superstore will illustrate that a large number of participants have entered the supply side of the voiceband modem market.

If you design a new modem, one of your steps in qualifying the design is to take your company's credit card, visit said electronics superstore, and buy one of every modem on the shelf. Then, set up interoperability tests of your modem with each of the competing modems.

Don't forget to set up connections in all available modes (V.90, V.34, V.32bis, V.22bis, Bell 212, Bell 103, V.23, V.21). And, oh yes, don't forget the fax modes (V.34, V.17, V.29, V.27ter, V.21).

PERFORMANCE

The definition of performance has evolved along with modem technology. Traditionally, modem performance was expressed as a curve showing error rate as a function of received signal-to-noise ratio.

Modem A was better than modem B if modem A achieved a given error rate with a lower signal-to-noise ratio than that required by Modem B. Because error control has come to be an integral part of their design, virtually all modems now perform error free. So much for error-rate comparison.

Current standards define several measures of performance, one of which is throughput. Throughput is the rate at which user data flows from the interface of the transmitting modem to the interface of the receiving modem.

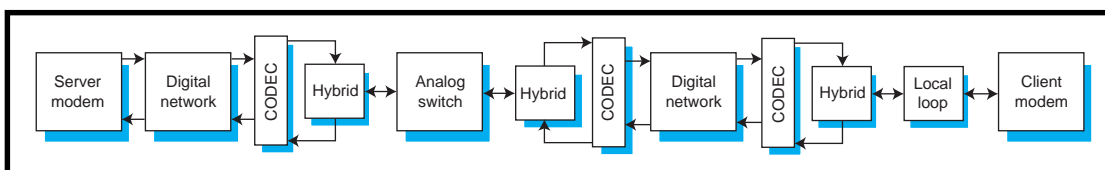


Figure 2—There are two stages of D/A conversion between the server and the client, which means that PCM connection is not possible.

Date	Standard number	Standard title
November 1989	EIA/TIA-496-A	Interface between data circuit-terminating equipment (DCE) and the public switch telephone network (PSTN)
February 1992	EIA/TIA TSB-37	Public switched telephone network transmission simulation for evaluating modem performance
October 1994	TIA/EIA TSB37-A	Telephone network transmission model for evaluating modem performance
December 1994	TIA/EIA TSB38	Testing procedure for evaluation of two-wire 4-kHz voiceband duplex modems
February 1999	TIA PN 3857, Draft 10	North American telephone network transmission model for evaluating analog client to digitally connected server modems
Currently	TIA PN 3856 (Draft)	[Extension to TSB 38 to cover PCM modems]

Table 1—Voice-band modem technology has advanced from 300 bps in the 1960s to modems that approach 56 kbps today (during which time the North American telephone network has evolved from 100% analog to nearly 100% digital). Here you can see how testing standards have kept up with these advances.

Modems achieve error-free data transmission by gathering user data into blocks encoded so an errored block can be detected by the receiving modem. The receiving modem then requests a retransmission.

The detection and retransmission eliminates most errors, but only by trading off throughput. All other things

being equal, modem A is better than modem B if it has higher throughput.

CONNECTIVITY

The meaning of connectivity has also evolved with advances in modem technology. Today's modems nearly always connect, thanks to their multi-mode structure. Typical modems can

switch personalities to become V.90, V.34, or even V.32 or V.22 modems.

Most connections today are over channels capable of supporting PCM operation (e.g., V.90) at speeds beyond 40 kbps. The connection between the server modem at an ISP and a client modem can go through several stages of digital networks and switches.

Sometimes, a DLC even extends the digital network partway from the central office to the subscriber's neighborhood. If the whole conglomeration, starting with the server modem, is digital up to a single CODEC with analog thereafter to the client modem, then a PCM connection is possible. Figure 1 illustrates a PCM-capable connection.

Some network configurations contain more than one D/A–A/D stage, as shown in Figure 2. When this occurs, the conditions that permit PCM operations are lost, and the modems must fall back to V.34 mode.

During their handshake, modems determine whether a PCM connection is possible. If it is possible, the mo-

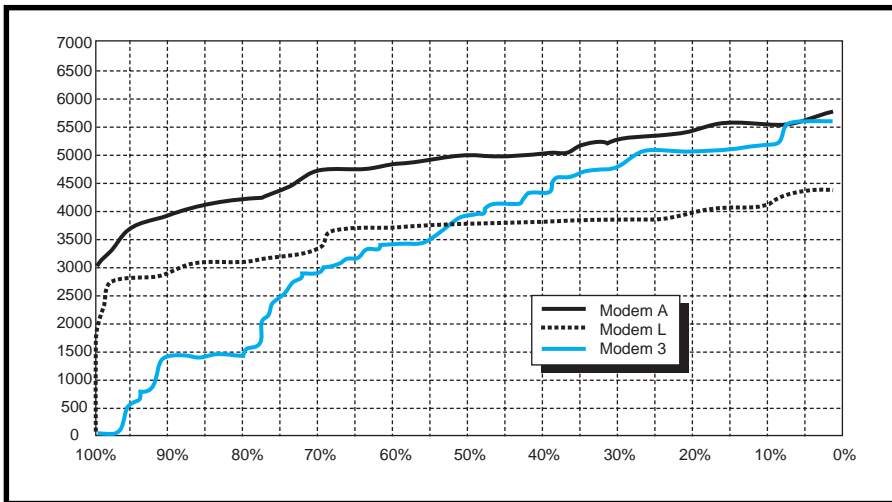


Figure 3—From a network standpoint, Modem A performs at 4700 cps over 70% of the network model whereas Modem L performs at about 3300 cps, and Modem 3 at a throughput of less than 3000 cps. Thus the chart gives the customer an objective means to compare the throughput of these three modems.

dem determines how to cope with any RBS and digital/analog pads that may be present. When in doubt, the modem falls back to V.34.

Modems that take full advantage of a PCM link are clearly superior to those that fall back to V.34 with its attendant reduction in speed.

HISTORY OF STANDARDS

The TIA/EIA developed several North American testing standards to help compare modems. Table 1 presents a chronology of North American modem test standards, and the “Glossary” sidebar contains TIA definitions for some key terms.

Modem testing began to be standardized in 1989 when the EIA/TIA published the 496-A standard. This standard defined a simple telephone network model that included a set of six test channels representative of end office-to-end office telephone connections.

This connection model included the effects of amplitude, delay distortion, as well as white noise, and was adequate for testing modems with speeds up to 2400 bps and modems such as V.22bis. Equipment based on the standard was used for testing V.32 modems, but users were often confronted with apples-to-oranges comparisons when comparing test results because there was no standard for testing echo-canceled modems.

The deficiencies in EIA/TIA-496 were addressed in TSB 37-A, which was published in 1992. TSB 37-A built

on EIA/TIA-496-A but augmented the number of test channels from 6 to 16 and added a laundry list of simulated additional impairments. These impairments included phase jitter, impulse noise, intermodulation distortion, near-end echo, and far-end echo.

The model also included the effects of local loops (i.e., the two-wire paths

that connect the user to the telephone company’s central office) and the effect of the input impedance of local loops. This model was good for evaluating the performance of modems from an error-rate standpoint.

Meanwhile, more and more modems began to include error control, so the TSB 37-A model was only useful when error control was disabled. Disabling error control lightened the load on the processors that implemented the modem, which reduced the utility of the test. It was time for another upgrade in testing standards.

The TIA introduced a new approach to modem testing in 1994 with the TSB 37-A and TSB 38 standards. TSB 37-A defined various telephone-network subsystems that could be combined with LOOs to simulate modem operation over a wide variety of typical connections.

TSB 37-A brought the recipe for the telephone network model to the testing party, but TSB 38 provided the baking instructions. TSB 38, “...provides a consistent set of repeatable

test procedures designed to characterize the performance of modems. This is achieved by stating the precise configuration of all the required test equipment, then giving step-by-step instructions for performing each test. This document also suggests some formats for analyzing, interpreting, and presenting the results" [1].

TSB 37-A and TSB 38 defined network coverage. Test results now indicate the percentage of the telephone network over which a given level of performance can be expected. These standards provide a basis for testing analog modems up to 33.6 kbps (V.34).

The latest (and maybe final) innovation in voiceband modems came when 56-kbps PCM modems were introduced. Besides creating an upheaval in the national and international stan-

dards bodies, these modems created another testing dilemma—TSB 37-A said nothing about digital networks. It was time for yet another upgrade.

DRAFT PN 3857 AND PN 3856

The simulation model defined in TSB 37-A and in its update, PN 3857, employs the concept of network coverage. Defining a model that covers all possible situations is impossible, so the model sets up a large number of possible combinations along with an approximate likelihood of each combination. The details get pretty hairy, but the concept is simple.

PN 5857 is 92 pages long and over 80% of the document contains tables and appendices of details of the model. The PN 5857 model defines suites of combinations of digital/analog network

Glossary

DLC—A digital loop carrier is a system that provides access via a digital carrier link to a central office for a cluster of subscribers. Individual subscribers are serviced off the remote DLC terminal with higher quality individual two-wire analog loops.

Digital pad—attenuation introduced into a PCM link by means of digital code translation

Hybrid—a three-port analog device that connects one duplex port to separate transmit and receive ports

LOO—likelihood of occurrence, a weighting factor applied to components in the PN 5837 network model

PCM—Pulse-coded modulation is a modulation/coding scheme used within networks for digital transmission of voiceband signals. All PCM systems in the U.S. today digitally encode a 4-kHz-wide analog signal into a 64-kbps digital bitstream using a sampling rate of 8 kHz. A/D and D/A converters are commonly implemented in pairs in a CODEC (COder-DECoder). These CODECs use μ -law companding (compression/expansion) in North America as specified in ITU-T Recommendation G.711 to effectively provide a dynamic range equivalent to that of a linear 12-bit coding system, but using a sample word size of only 8 bits, which results in 256 possible signal levels

PCM link—a digital link employing PCM or ADPCM encoding that terminates in a four-wire analog interface

PCM modem—a modem, as specified in the ITU-T V.90 Recommendation, whose line signal at the sample rate is one of the levels generated by a PCM CODEC

RBS—Robbed-bit signaling is a technique that expropriates PCM bits normally used to carry voice-signal information to convey supervisory and call control information. RBS is a form of in-band signaling that uses the least significant bit from every sixth frame of one 64-kbps PCM channel. The end result is that when the least significant bit is used (i.e., robbed), the signal is effectively a seven-bit value for that sample, with the least significant eighth bit effectively becoming a random value. The resulting output appears as the original signal with a low-level impulse noise hit. RBS may be used in DLC systems.

components and the impairments that often accompany each component.

Digital impairments include digital pads and various forms of RBS. Analog impairments include analog pads, the effects of various lengths of 22-, 24-, and 26-AWG twisted wire pairs, bridged taps, loading coils, Gaussian noise, echo, power-line induced noise, and nonlinear distortion. The model assigns an LOO to each component.

PN 3856 defines many tests that apply to the PN 3857 model. A typical test computes the throughput for a given configuration along with the LOO for that test case.

Performing tests in a suite yields a list of throughput values and corresponding LOOs. Displaying throughput as a function of cumulative LOO defines a network coverage curve that characterizes a given modem. The goal is to achieve the largest throughput over the maximum percentage of the network.

REPORTING THE RESULTS

Throughput test results are often presented as a curve displaying the

throughput achieved as a function of the percentage of coverage given by the reference network. Figure 3 shows throughput in characters per second (cps) for three different modems.

For about 5% of the network, modems A and 3 achieve throughput in excess of 5500 cps and modem L achieves less than 4500 cps. Around the 52% coverage point, modems 3 and L both operate at about 3750 cps and modem A operates at about 5400 cps.

At 97%, modem 3 no longer passes data and modems L and A achieve about 2800 cps and 3500 cps, respectively. This curve demonstrates that modem A provides higher throughput than modems L and 3 in this test.

HANGING UP

Designing voiceband modems has become quite a process. The sophisticated DSP algorithms coupled with the power of today's embedded processors can lead to software-based designs. Testing is a key component of the design process although testing itself can never improve a bad design.

Standardized testing provides OEMs with consistent and uniform criteria for comparing modem performance levels and for avoiding those pesky Peppy Puppy problems. 🐶

Arthur J Carlson holds a Ph.D. in Electrical Engineering from the University of Iowa. He spent the first 12 years of his career on the faculty of the University of Missouri, Columbia. He then relocated to Silicon Valley where he has worked in high-speed modem design for 20 years. Art is a senior scientist at AltoCom in Mountain View, CA. You may reach him at ajc@altocom.com.

REFERENCE

- [1] TIA/EIA TSB38, "Testing Procedure for Evaluation of Two-Wire 4-Kilohertz Voiceband Duplex Modems," Introduction.

RESOURCE

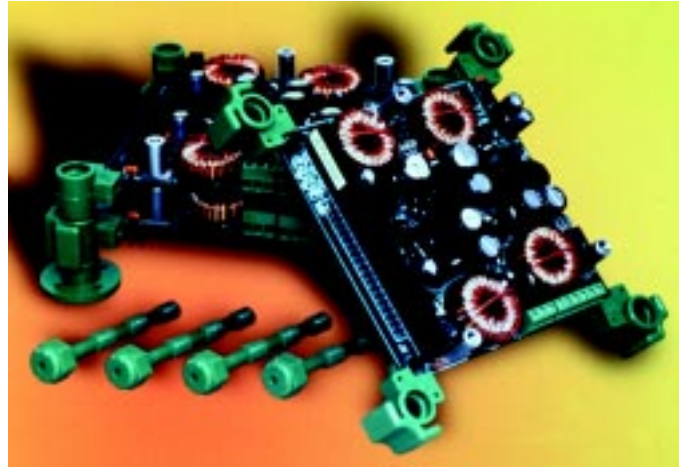
TIA standards info,
www.tiaonline.org

UNIVERSAL PC/104 BOARD MOUNTS

A family of components for mounting PC/104 and PC/104-*plus* boards has been announced by parvus. Collectively called **SnapStik** components, the family consists of a series of snap-together board separators and other components that create a robust, low-cost, incremental card cage for PC/104. SnapStik is designed for embedded applications ranging from industrial equipment panel installation to high-vibration, high-shock environments.

Basic to the SnapStik PC/104 component family is the SnapSlot. SnapSlots are hollow polypropylene card-cage rail segments with length equal to the spacing required for one PC/104 board. SnapSlots attach to the corners of PC/104 boards using a threaded 4-40 insert and machine screw. Multiple PC/104 boards with SnapSlots attached to each corner snap together to form an incremental card cage called a SnapStak.

A SnapGuide is similar to a SnapSlot, except that a SnapGuide has an extension formed to fit the inside of an extruded enclosure. A SnapShok is a SnapSlot fitted with a silicone rubber shock-absorbing tire, designed to fit inside an extruded enclosure. Other SnapStik components include a selection of specialized side and end mounts, bolts, nuts, spacers, and disk drive- and fan-mounting options.



For a limited time, PC/104 developers can order a SnapStik Starter Set, a \$55 value, for an introductory price of **\$24.95** (limit two per customer).

parvus Corp.
(801) 483-1533
Fax: (801) 483-1523
www.parvus.com

CompactPCI CPU BOARD

The **C2P3 CPU** board is targeted at compute-intensive applications such as telecommunications, aerospace, and imaging. It features two Pentium III processors running at 550 MHz, 1 GB of main memory, and 1 MB of L2 cache. It incorporates Intel's 82443BX chip set, the Intel 740 Advanced Graphics Processor (AGP), and is the first CPU board to provide a 100-MHz implementation of Intel's FSB (Front Side Bus). The DEC 21554 Draw Bridge Chip is also included to enable multiprocessing.



Networking and I/O features include dual Ethernet interfaces (twisted pair) operating at either 10 or 100 Mbps, a 40-Mbps ultra-wide SCSI, and a 64-bit AGP graphics engine with 4 MB of video RAM optimized for 3D rendering. Also available are two Ultra-DMA 33 IDE interfaces, a pair of USB ports, dual serial I/O with optional RS-422 drivers, and a parallel port.

The board can support hot swap I/O modules on the CompactPCI bus and can accommodate a 2.5", 9-GB IDE drive. For applications that must be deployed without a rotating hard disk, the board also provides up to 340-MB SanDisk 1.5" flash IDE on the rear panel. The C2P3 runs a variety of popular desktop and real-time operating systems and comes equipped with AMI's BIOS and onboard diagnostics software and status LEDs.

Pricing for the C2P3 starts at **\$1995**, less processor and memory. For high performance at low cost, the C2P3 can accommodate two Celeron PPG370s.

General Micro Systems, Inc.
(909) 980-4863
Fax: (909) 987-4863
www.gms4vme.com

Nouveau PC

edited by Harv Weiner



SINGLE-BOARD COMPUTER

The **CoreModule/P5e** uses the new Intel BGA-packaged Mobile Pentium processor with MMX technology to offer up to 266-MHz performance in a PC/104 form factor.

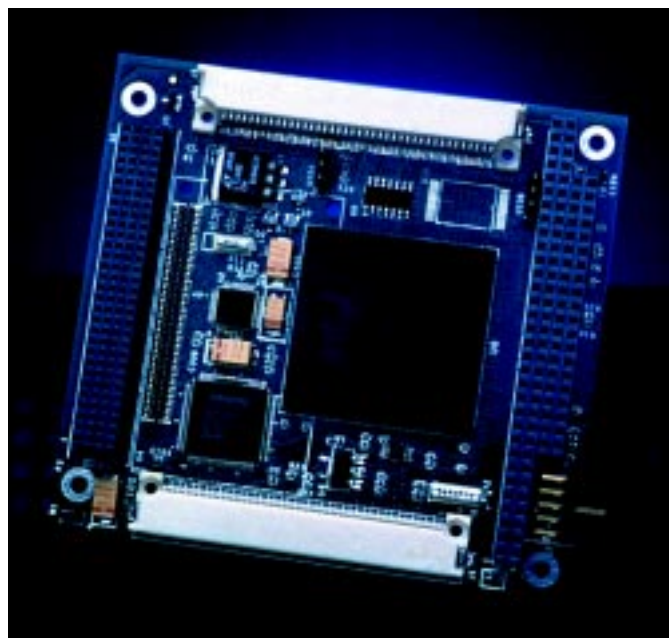
The card contains all the functions of a fully configured PCI-based PC-compatible system, including disk controllers, two FIFO-buffered serial ports, an enhanced-capabilities parallel port, PS/2 keyboard/mouse interfaces, two USB ports, an IrDA port, and a built-in DiskOnChip2000 solid-state disk drive. An efficient DC/DC converter, to supply the special voltage requirements (3.3 and 1.9 VDC) of the processor and core logic, is built directly into the CoreModule/P5e, resulting in single-supply (+5 VDC) system operation and minimal power consumption.

An extended temperature version of the SBC supports -40°C to +85°C and operation with or without a fan is supported. The CoreModule/P5e can withstand 50-G shock and 12-G vibration, under MIL-STD-202F. EMI, EMC, and ESD compliance is in accordance with the European CE mark standards (EN 55022 Class B and IEC 801-2, -3, and -4). Additional reliability enhancements include a watchdog timer and a power-fail NMI generator.

In addition to DOS and Windows 98, NT, and CE, support for most leading RTOSs is also available.

Pricing for the CoreModule/P5e starts at **\$729** in OEM quantities of 100.

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



HIGH-SPEED PARALLEL DSP MODULE

The **CRT-1260** is a high-speed DSP module in a PC/104 form factor. Designed around the NeuroCam NC3001 Parallel DSP chip, the module features 32 fixed-point multiply-and-accumulate processors operating in parallel with a three-stage pipeline. This design enables the module to achieve 1000 MOPS, making it ideal for embedded applications such as real-time vision, optical character recognition, pattern matching, high-speed digital filters, and speech recognition.

The NC3001 pDSP is specifically designed with artificial neural networks for fast learning and recognition. Its architecture is optimized for the implementation of the Reactive Tabu Search learning algorithm, a competitive alternative to back-propagation which leads to a very compact implementation. Internal resources can be assigned either to a single neuron or be partitioned among several neurons to implement multilayer networks.

The CTR-1260 is available in PC/104 form factor with a 16-bit interface. It features a 512-KB frame memory and an auxiliary EPP parallel port for direct connection to digital cameras and other peripherals. It also includes an intuitive Windows-based software interface and is available as a 3U CompactPCI with two pDSP processors.

The CTR-1260 is priced at **\$995**.

EuroTech
+39433-486258
Fax: +39433-486263
www.eurotech.it

Nouveau NPC

What's in a Name?

Windows CE vs. a Hard RTOS

Name recognition is important in today's software market, but it's no substitute for product performance. So, how does Windows CE stack up against some of the established RTOS technologies? Mal checks it out.

With the Windows cachet (in appearance if not reality) and the Microsoft marketing machine, Windows CE has done for the embedded-systems industry in a few short months what established RTOS companies have labored for years to provide—recognition by the general public.

In this sense, it means that much of the publicity generated around Windows CE also benefits the industry through a higher level of awareness of the underlying software platform of new and innovative embedded devices.

But when the hoopla has died down, and embedded designers have to produce a product, they look beyond the hype to the RTOS details to ask if Windows CE meets the technical requirements of the project. No doubt many embedded systems designers are asking that question right now. And Microsoft would certainly have them believe it does.

But to be honest, design engineers don't care whether or not a design uses Windows CE. In all likelihood, they're promot-

ing a network router, a cellular telephone, a set-top box, or a data-acquisition device. They're not out to promote Windows CE, unless they got an unbeatable deal from Microsoft, or unless their product needs all the help it can get.

Microsoft presents Windows CE as though it were breaking new ground in embedded systems. But RTOSs and the devices they control have existed for years.

When it comes down to choosing a software platform, there are several alternatives that were available many years before Microsoft ever conceived of Windows CE. Comparing Windows CE technologies with the established RTOSs shows just how far Microsoft has gone to provide a serious product for embedded markets.

THE BUSINESS MODEL

For designers looking for a platform capable of supporting a unique design, the ability to work with the platform vendor to devise the optimum RTOS configuration, performance, and footprint is one of the

most important selection criteria (see Figure 1). Out of the box, no OS meets most embedded developers' needs, so the ability to develop a close relationship between platform vendor and customer is critical.

This is a fundamentally different model than the one Microsoft used to build its PC operating-system business. On the desktop, the platform is a mass-market product that's distributed in an identical version to hundreds of PC vendors, who then build the hardware around it. Is it any wonder that all PCs are practically the same?

Microsoft is trying to use a similar business model for distributing Windows CE. Rather than selling it directly to design engineers and working with them to incorporate it into a product, Microsoft licenses a single version to a handful of OEM distributors who are responsible for implementing board-support packages and providing software utilities and consulting services to embedded designers.

Industry analysts have lauded Microsoft for bringing in experienced embedded-

system vendors as distribution partners rather than trying to go at an unfamiliar market alone. Their plan works well, up to a point.

In general, these OEM distributors do a credible job of providing tools, sample code, and consulting services to help get a design working with CE. Designers may even consider the distributor a business and technology partner in their efforts.

But Microsoft is not in this loop. They do provide some telephone technical support for Windows CE, although this same support covers all manner of embedded systems, handheld PCs, Windows terminals, and any other market that CE penetrates. The support covers OS issues like building a customized platform, scheduling processes, and setting up networking utilities, but it doesn't go far in providing detailed technical assistance to a specific design.

In contrast, almost all traditional RTOS vendors sell directly to vendors or engineers engaged in development projects and provide direct system software and support services for their customers. When engineers call for technical support, to report a problem, or to discuss what features they'd like to see in future releases, they're talking directly to the company (and often the individuals) that designed and implemented the OS. There's no better way to get feedback directly into the hands of those who can use it.

ARCHITECTURE AND PROCESSOR SUPPORT

The architecture of the OS by itself doesn't explain any advantages of the software, but it helps define the ability to implement characteristics like scalability, performance, and reliability (see Figure 2). And, it often says something more accurate about the intended market of the OS than any position paper.

Windows CE was designed by Windows system architects and it shows. It's designed as a large group of OS files, most running in kernel space (which improves

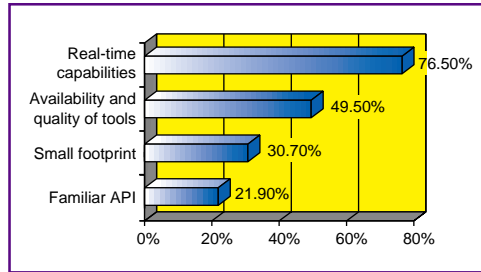


Figure 1—What are the key criteria for selecting an RTOS? Developers are looking at real-time capabilities, development-tool quality, and footprint size. A familiar API was considerably less important (source: Venture Development Corp.).

performance but results in a relatively large memory footprint).

Rather than using a messaging system to communicate between user and kernel objects and between kernel objects, CE uses an event model that converts interrupt events to messages, which are then queued in priority order. The message queue is polled, so it's possible for a message to be sitting in the queue for a relatively long time before it's acknowledged and processed. Although this is an acceptable design for a desktop windowing system, it's not a real-time activity.

Much of the software available with Windows CE (e.g., TCP/IP and remote access, web browser, e-mail) are versions of the same software found in desktop Windows. Although this can provide familiarity to end users, it also means that the software wasn't designed for embedded use. The result is software that is larger and slower than it should be for a typical embedded system.

On the other hand, traditional RTOSs were designed for systems where memory

and processing power were limited. Take the QNX RTOS, for example. The original QNX microkernel was about 12 KB, whereas the QNX/Neutrino microkernel is 34 KB. Within this 34 KB is enough memory for interprocess communications, interrupt handling, and thread scheduling—all the fundamental facilities needed for a basic OS in a limited-memory environment.

Since its inception, Windows CE has had broad processor support that includes MIPS, Hitachi, ARM, Intel, and others. It's important to realize, however, that it's not Microsoft that provides this support, at least not directly. Microsoft provides a single code base to distributors and processor vendors, who work together to produce the port.

The process was designed to put CE on several different processors in a short period of time, and it succeeded. But embedded-systems designers had no input into the process, so the processors that are supported may not be the most useful ones.

Other RTOS vendors have long recognized the value of supporting multiple

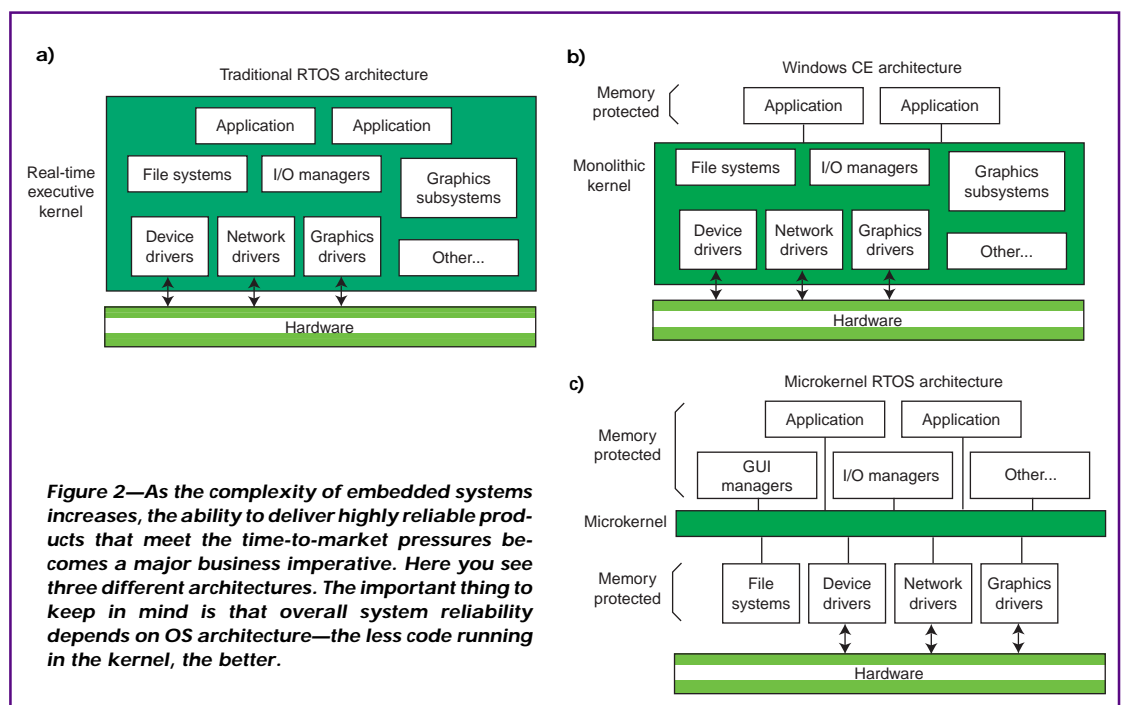


Figure 2—As the complexity of embedded systems increases, the ability to deliver highly reliable products that meet the time-to-market pressures becomes a major business imperative. Here you see three different architectures. The important thing to keep in mind is that overall system reliability depends on OS architecture—the less code running in the kernel, the better.

processor families. By providing a choice of processors, RTOSs such as QNX, Microware's OS-9, and ISI's pSOS enable designers to choose processors by cost, performance, or other technical consideration, while keeping the same system software characteristics and programming tools.

GRAPHICS AND WINDOWING

One of the major misconceptions of Windows CE is that it is related to the desktop Windows. Microsoft gives a perception of a strong relationship, both with the product name and through its offering of limited versions of traditional desktop applications (e.g., Excel, Internet Explorer).

In reality, the only feature shared between Windows CE and desktop Windows is the Windows API—the programming interface for Windows platforms. The underlying OS is almost entirely new, but it borrows technologies from other Windows OSs.

A separate but related issue is that (at least for embedded systems) most people believe there is a Windows-like graphical user shell, like the ones found on CE-based handheld PC units. But, Microsoft doesn't provide a user shell for the embedded version of the OS.

Embedded developers have to build their own shell or buy a commercially available one. The supposed advantage of the Windows-like graphical desktop simply doesn't exist.

Embedded-system developers have the same requirements for a GUI as they do for the RTOS itself. They don't care if the program can be ported from the desktop because applications on handheld and embedded devices have different features and uses. What is important is size and performance, both characteristics that are far better represented by the likes of QNX's Photon and Microware's MAUI.

For example, Photon is small (under 200 KB for a complete implementation), fast, and easy to use. Also, its memory footprint is small, so engineers can include more user features into the same amount of space required by Windows CE.

DEVELOPMENT TOOLS

The availability of rich and comprehensive application development environments is supposedly the true strength of Windows CE. This includes both the Win32 API (the de facto standard on the desktop)

and the popular Visual Developer Studio IDE. Microsoft would have you believe that you can apply the same APIs and development techniques to write CE applications that you would use on the desktop.

The reality is more complicated. The full Windows CE supports only about a third of the APIs found in Windows 95 and NT. And they aren't exactly the same calls. Microsoft refers to them as being "in compliance" with Win32. In fact, it's different enough that Microsoft refers to it as Win32 for Windows CE API and gave it its own programming references.

For a programming interface, many established RTOSs use the only platform-neutral programming standard available today—POSIX (or some variation thereof). For the most part, programs are developed specifically for embedded systems rather than ported to an embedded device from a desktop application.

Microsoft's use of the Win32 API, a proprietary programming interface specifically for Windows systems, encourages programmers to use desktop code and programming practices for embedded development, perpetuating slow and bulky programs that are inappropriate for embedded systems.

POSIX is a true standard with international recognition and steering committees representing broad industry segments. This arrangement ensures that embedded code won't be broken by arbitrary or unannounced changes made to benefit one vendor.

Microsoft notes that you can use Visual Basic or Visual J++ (its own brand of Java), in addition to Visual C++, to write Windows CE applications. This sounds like an impressive array of languages, but few, if any designers are going to use Visual Basic for an embedded application because its run-time support alone requires well over 1 MB. Not to mention that pure Java development tools for producing truly portable code are available from dozens of independent software vendors, without the confusion generated by Microsoft's own extensions to the language.

But thanks to the extensible nature of Visual Developer Studio and Visual C++, it's possible to use this fine environment with virtually any other RTOS. In fact, several embedded vendors use the Visual Studio with plug-in modules for software development with their own RTOSs.

At least one other programming product is a strong competitor to the Microsoft development tools hegemony. MetroWerks' Codewarrior is the same compiler and IDE that dominated the Macintosh software industry and is now making its mark in embedded-systems programming.

Codewarrior is available on many processors and RTOSs, and it offers a wide range of programming languages. In other words, it's possible for an embedded system to have a first-class set of tools without depending on Microsoft.

FUTURE SUPPORT, OR LEFT HANGING?

Technical comparisons are all well and good because they compare the characteristics of RTOSs at a particular point in time, but you also have to consider how these characteristics change over time. Windows CE is a new RTOS that has little history in supporting products over a period of time. If history is any guide, Microsoft has little compunction in changing software architectures and interfaces every few years and requiring computer designers, programmers, and users to change everything to keep up.

Traditional RTOS suppliers are much more cognizant of how dramatically even minor changes affect system designers. Changes that affect how the RTOS is used in an embedded design are embarked on quite infrequently and in consultation with customers on implications to their products.

For example, QNX changed program architecture only when it became a 32-bit OS, over ten years ago. Even QNX/Neutrino, a kernel introduced in 1996, uses the same basic microkernel structure as the older implementation (with support for symmetric multiprocessing and different processor families). Its strict adherence to the POSIX interface means that programs developed years ago still run today.

Microsoft has a canned answer to all of the technical deficiencies of Windows CE—wait until the next version. And they promise that the next version will be a complete rewrite that improves real-time characteristics, but at the expense of backward compatibility.

To get real-time response for applications, Microsoft is supplementing the Win32 API with a new real-time API, negating one of its biggest selling points—the familiar programming interface.

Such practices may work on the desktop, where Microsoft's dominant market share and application base mean that users have little alternative but to wait. But, time to market has a different meaning in embedded systems. Embedded designers need specific technical characteristics to implement their products' features. They can't wait until Microsoft decides that what they need is worth including in a future version.

Granted, Windows CE has advantages that traditional RTOSs don't: automatic recognition, of course, and Microsoft's ability to improve CE over time by allocating immense resources to the effort.

The question is not whether the Microsoft developers can improve CE, but whether they have an architecture that can scale down from the handheld PC over to more traditional embedded applications. Or, will engineers suffer through countless versions of OS incompatibilities, forced to constantly change their code because Microsoft couldn't get it right the first time? [EPC](#)

Mal Raddalgoda is the senior technology analyst for QNX Software Systems Ltd., concentrating on the North American market. Mal has over 10 years of experience

in the computer and telecommunications industry and has held positions in product development, product management, and marketing. He is also a member of the Embedded Software Association (ESOFTA). You may reach him at mal@qnx.com.

SOURCES

Windows CE, Visual Basic, Visual J++, Visual C++
Microsoft Corp.
(206) 882-8080
Fax: (206) 936-7329
www.microsoft.com

QNX/Neutrino, Photon
QNX Software Systems, Ltd.
(613) 591-0931
Fax: (613) 591-3579
www.qnx.com

OS-9, MAUI
Microware Systems Corp.
(515) 223-8000
Fax: (515) 224-1352
www.microware.com

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

Codewarrior
Metrowerks, Inc.
(800) 377-5416
(512) 873-4700
Fax: (512) 873-4901
www.metrowerks.com

Real-Time PC

Ingo Cyliax

Where in the World...

Part 3: Fighting the Wind with GPS

Even the best GPS technology can't control the elements of nature, so Ingo had to compensate his data-acquisition system for the effects of crosswinds to make sure the ground-mapping camera shoots straight.

For the last two months I've written about GPS and some of its applications. Last time, I wrote about a prototype system that I developed for Near Earth Observation Systems (NEOS), a company specializing in airborne remote sensing.

This system is a GPS-aided data-acquisition device that also uses GPS to aid in navigation. The annotated data is integrated in geophysical information systems.

One application for this type of system is to take infrared imagery and use spectral analysis to determine the quantity and type of ground cover that exists. Gathering this data is important for land management such as agriculture and forestry.

The prototype was successful, but I needed to add more features. In this final article of this series, I'll describe one of the features that we're adding to the system.

As I described in Part 2, a small aircraft flies a preplanned grid and takes pictures of the ground using a downward-facing camera. Following the grid, the plane

flies north and south tracks covering the ground in swaths. The width of the swath depends on the field of view of the camera and the altitude above ground level.

The north-south tracks make it easy to integrate the data with existing maps and data. NEOS primarily uses ultralight aircraft to fly these missions. If there's any significant crosswind, the pilot has to turn the aircraft into the wind to keep on course (see Figure 1).

Here's the clincher. As the aircraft turns into the wind, the photos it takes will

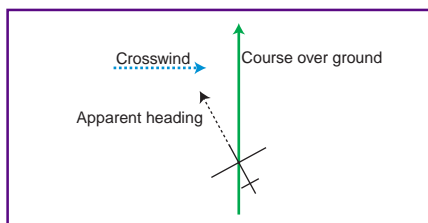


Figure 1—Here's what an airplane that is fighting crosswind looks like. The official term for this is "crabbing" because, in its extreme form, it looks like a crab walking sideways.

be angled with respect to the ground track the flight was intended to cover. With severe crosswinds, the angle of adjustment can be quite significant.

Figure 2 shows what a track of the crosswind pictures looks like. As you can see, the pictures are not squared up with the north-south meridian and you have to rotate them to use the data. Also, because the edge of the swath is ragged, we have to overlap the swaths to make sure we cover all the little nooks. This adjustment causes us to burn more fuel because the effective swath width is reduced and we have to fly more swaths for the same area.

The solution is to mount the camera on a camera pod that can be turned relative to the aircraft. This way, the camera can be pointed to compensate for the crosswind. However, it's yet another thing for the pilot to manage while flying.

One of the objectives for this system is to reduce the pilot workload. Less work makes for less mistakes. A mistake usually

means reflying the swath or even the entire mission. Also, less workload lets the pilot focus on flying.

Camera-pod rotation is an obvious thing to automate. We have the heading the plane is flying over the ground (north-south track) and also the direction we want the camera aligned with. If we can figure out how far off the plane's apparent heading is, we can drive a small actuator to turn the camera.

Of course, there's already a GPS receiver in the system as a pilot-navigation

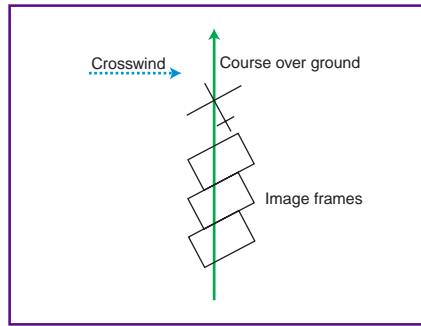


Figure 2—When we try to take pictures as we are turning into the wind, the pictures are no longer aligned to the north-south meridian, which is what we would like to see.

aid and to annotate the data, but it only gives us the heading over the ground (i.e., the heading of the course we are tracking over the ground). It obtains this information by taking position samples and computing the direction between the position fixes it has made. It doesn't know which way the aircraft is turned. Incidentally, the GPS receiver can't compute an accurate heading when it's not moving.

But we need to figure out what direction the aircraft is pointed because the camera is attached to the aircraft's frame. Once we know this heading, it's easy to compute the difference and adjust the angle.

Getting the direction of the aircraft is actually pretty easy. Often we get caught up in new technologies and miss obvious solutions—like using a magnetic compass.

The magnetic compass always gives the direction of the airframe relative to the earth's magnetic flux lines. Before GPS, this behavior was a nuisance because you had to know the crosswind error to calculate the true course heading. For our purposes, the preadjusted heading is just what we want. There's one small gotcha.

Compass headings are magnetic. The magnetic headings are based on the magnetic north pole, which is somewhere in North Canada and nowhere near the true North Pole used for map datums, which means there's a magnetic declination to consider.

The declination value is usually indicated on maps, to show how much the magnetic heading differs for the true heading. The GPS receiver finds this information by computing the heading to the magnetic north pole from our current position.

The angle we have to adjust the camera compared to the airframe is shown by:

$$\text{ang_camera} = (0 - (\text{ang_maghead} + \text{ang_magdecl}) \bmod 180)$$

The `ang_maghead` and `ang_magdecl` variables are provided by the compass and GPS receiver, respectively.

PROJECT

This project entails building a heading compensator that rotates the camper platform to compensate for the crosswind heading error. For this, we need a compass, a GPS receiver, and an actuator.

We have a GPS receiver in our system and we've looked at the NMEA message

formats in previous articles. If you remember, the GPRMC sentence most GPS receivers generate contains a field that describes the magnetic declination.

The declination is usually a small angle that describes the difference between the magnetic and true heading. Where I live, this difference is about 2.9° west. That is, the magnetic north pole is 2.9° west of true north. Or, a 0° magnetic heading would be equal to a true heading of 2.9° east.

There are various compass schemes. Mechanical compasses use a magnetic needle. To measure the position of the needle, you need a rotational encoder. These compasses are easy to build, but they contain movements that are subject to vibration damage and oil that can leak.

Electronic compasses use a magnetic flux sensor to measure the magnetic field directly. This setup is preferred because no moving parts means the compasses are robust. Flux sensors are either solid-state HAL-effect sensors or coil-based.

There are several techniques used with coil-based sensors. The trick is to make them sensitive and small, yet immune from various noise sources. Electronic compasses used to be rather exotic and expensive, but with advances in microprocessors, it's possible to implement signal processing in software, which makes these compasses less expensive now.

I wrote about Precision Navigation's Vector 2D module in "Robot Navigation Schemes" (*Circuit Cellar* 81). This module is inexpensive and available from mail-order sources such as Jameco.

One of the drawbacks of this module is that it only uses two coils and can thus only be used when it is level. A gimballed version allows it to be used in nonlevel applications. However, this module uses a serial bus interface, which would make it hard to interface with our system.

Precision Navigation also makes a NMEA-based compass module—the TCM2. This module is flexible, has a temperate sensor, and in addition to the heading, can measure roll and pitch.

The TCM2 comes in several different versions, mostly based on how much inclination they can handle. The low-end version can handle ±20° and the top-end version handles ±80°. The TCM2 has low-power modes, is robust, and was designed to be used in mobile applications like ours (see Photo 1).

The most important feature for us is that the modules can speak NMEA protocol via an RS-232 link. A three-wire interface (Tx/Rx/ground) is all that's needed to interface it to a PC-based system like ours.

Like all NMEA devices, this device uses a two-letter prefix to indicate what kind of device it is. In our case, HC is the prefix for a magnetic compass.

Magnetic compasses only have one sentence type, the HDM (heading magnetic) sentence. HDM has two fields—HD is the magnetic heading value and M indicates that the heading is magnetic.

An example of an NMEA message from this module is:

```
$HCHDM,182.3,M*21<cr><lf>
```

This message indicates a heading of 182.3° west. The module can also be programmed to send nonstandard NMEA messages with more than just the heading. Also, by making our application use standard messages, it's possible to simply drop in another compass module, provided it speaks NMEA.

Placement of the compass module is an exercise in compromises.

We can place the compass module on the camera pod itself or on the aircraft. If we put the module on the camera pod, we can use a module with lower inclination tolerance (cheaper) because the camera pod is gimballed and always level. Also, calibration is simpler because the compass module is coupled to the camera platform.

Mounting the compass on the airframe has the advantage that we can mount it away from most of the metal and electrical wiring by putting it at the tip of the wing. It's easy to change in the software, so we can leave this issue open and experiment.

For the actuator, I'll use radio control (RC) servos. These servos are simple to control and come in various sizes with different torque performances. High-end servos have metal gears and ball bearings for long life and can withstand quite a lot of abuse. Also, because they are mass-manufactured for the RC model industry, you can't beat the cost.

The control signal to an RC servo consists of a single signal that carries a pulse-width signal. The pulse width varies from 1 to 2 ms, with 1.5 ms being a center position. The pulse is repeated between 10 and 20 ms. Most servos have about $\pm 45^\circ$ of travel, which lets us compensate to crosswinds of up to:

$$\text{cross_wind} = 80 \text{ mph} \cos(45)$$

or 56 mph (far more than we'll be concerned about).

Interfacing a servo to a PC-based system could be tricky because we would need a pulse-width generator that can generate 1- to 2-ms pulses with a step size of about $(1 \text{ ms}/256) = 4 \mu\text{s}$. But luckily, this is a common enough problem that several servo adapters are available.

One such adapter is the serial servo controller (SSC) module made by Scott Edwards Electronics (see Photo 2). This PIC-based controller receives commands from an RS-232-based serial interface and can control up to eight servos per module.

By adding an address jumper on one of the modules, two modules can be bused together to expand it up to 16 servos. We only need one servo for now.

The transfer rate is adjusted on the module with another jumper and can be set for 2400 or 9600 bps. The word format is eight bits with no parity and two stop bits. The protocol is simple and consists of three bytes—sync, servo number, and position.

The sync byte is an all ones byte (0xff). So, to set servo number five to the midpoint, you would send 0xff 0x05 0x80. That's all there is to it.

Take a look at Listing 1 and you'll see that the code (Tcl) to make this subsystem work is pretty simple. The loop in Listing 1 waits for messages on a file description \$magchan. This is a serial port, on which the magnetic compass sits.

The compass outputs NMEA messages. NMEA messages are line oriented so we can use the Tcl gets function. In its default behavior, this function reads from the input channel until it encounters an end of line, in our case `<cr><lf>`. The line is returned in the variable line.

The next function is nmea_valid. This function checks to make sure that each sentence has a start \$ and an end * delimiter and it also

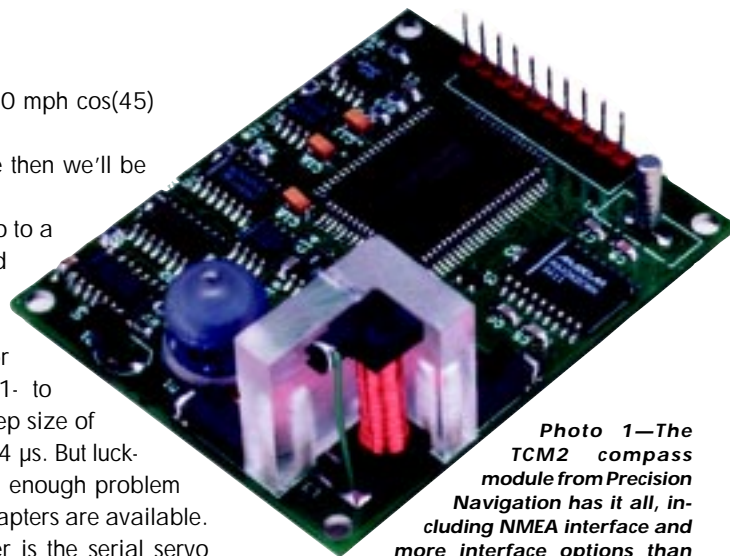


Photo 1—The TCM2 compass module from Precision Navigation has it all, including NMEA interface and more interface options than you'll ever use.

checks the checksum of the message. Remember that an NMEA checksum is simply the XOR of all the characters, excluding the delimiters.

If the NMEA message is valid, we parse the message into its fields. Tcl has a function called split for this parsing process. You specify the separation character (,) and it returns a list of all the fields.

Now we check the first field to make sure it's the message we're interested in, \$HCHDM. Because a \$ is special to Tcl, it tells it that this is a variable expansion, so we have to escape it with a backslash (\) character. The heading is the first field after the sentence-identifier field.

Next, we can compute the true heading by subtracting the magnetic declination and converting it to the servo code. The servo will go from 0 to 90° in 256 steps, so we scale the result.

By taking the modulus, we make sure the servo will stay within 45° of the north/south meridian. If we mount the compass module on the camera pod, we need to reverse the direction. In this mode, this loop behaves like a closed-loop system.

After we compute the servo position, we send it the servo controller. Tcl normally deals with strings and characters, so we have to use the binary conversion routine to convert integer values to a representation suitable for sending over the output routing (puts). Here again, the file descriptor \$srvchan is used for the serial port that has the servo controller attached to it. Once this is done, we do the whole thing again.

Integrating this code into the final system is straightforward. Get rid of the loop,

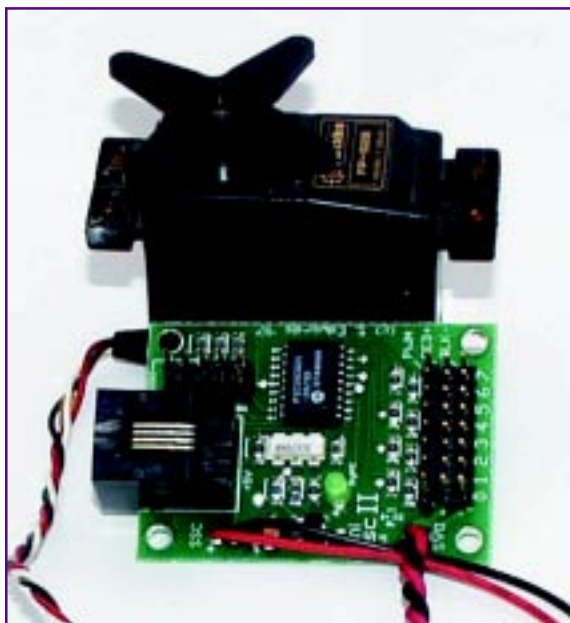


Photo 2—The serial servo controller is used to interface radio control (RC) servos to any computer with a serial port.

Listing 1—This simple code to implement the heading compensator is written in Tcl, which most of the project is based on.

```

set mag_dec1 2.9
#
# main loop, wait for new heading and update
#
while {[gets $magchan line] != -1} {
  #
  # check NMEA message format and checksum
  #
  if {[nmea_valid $line] != 0} {
    continue
  }
  #
  # split sentence into fields
  #
  set fields [split $line ',']
  if {[lindex $fields 0] != "$HCHDM"} {
    continue
  }
  set hdg [lindex $fields 1]
  set off [expr ($hdg - $mag_dec1) + 90]
  set pos [expr int (($off * 256) / 90) % 256]
  #
  # output the servo message
  #
  puts $srvchan [binary format "ccc" 255 0 $pos]
}

```

and make the code segment part of an event handler that gets called whenever there is activity on the serial port for the magnetic compass. Except for simple programs, using event-driven programming is the best programming style to use in Tcl.

WHERE I'M HEADED

Well, this article concludes my series on GPS. But, because I'm knee deep in this project, GPS is bound to pop up again. Next month, however, in response to several readers' requests, I want to cover serial

port interfacing and programming. Almost every computing device has a serial port, making it one of the most commonly used interfaces. [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.

SOURCES

Compass modules

Precision Navigation
(707) 566-2260
Fax: (707) 566-2261
www.precisionnav.com

Servo controller

Parallax, Inc.
(916) 624-8333
Fax: (916) 624-8003
www.parallaxinc.com

Remote sensors

NEOS, Ltd.
(909) 694-4096
Fax: (909) 677-7081
www.neosltd.com

GoAhead for Nothing

Getting the Server Started

If the words "Free Internet download" make you somewhat skeptical, you're not alone. Fred took a magnifying glass to GoAhead's web server software and found enough evidence to satisfy even his Internet-freebie skepticism.

Nothing, absolutely nothing, is free. No free lunch. No free pass. And especially, no free software.

So, why is this GoAhead WebServer stuff I found on the Internet given away for free? There's tons of "free" stuff on the Internet and frankly, I'm scared of a lot of it. What's the catch? Is this another scam?

As it turns out, there is a "catch." You can download the latest version of GoAhead WebServer V.2.0 for absolutely nothing. But in return, you must grace it with your application code and share your new-found knowledge (and modified web server) with other GoAhead users. Too good to be true? Let's go to the phones.

I spoke with the top folks at GoAhead and, after a nice discussion that covered automating recreational beverage delivery from the kitchen fridge as well as some really deep web-server queuing theory, I was convinced that the idea behind giving away the web-server code was sound.

Seems that GoAhead wants to establish a presence in the embedded control-my-

widget-over-the-web market. They figure the best way to do that is to provide an operational embedded web-server platform that's easy to port and modify.

Once the "free" web-server code is ported to an embedded platform, the next logical step is to incorporate it into an embedded web product. The more the merrier. These guys and gals may be giving away the store, but so far, so good.

The GoAhead WebServer runs out of the box with various flavors of Unix, QNX, VxWorks 5.3.1, Windows NT, Windows 95 with Service Pack 2, Windows 98, Windows CE, LynxOS, and the up and coming under-penguin, Linux.

GOAHEAD, FRED, TELL ME MORE

GoAhead WebServer is a standards-based full-featured C-based web-server application. You can download the application from www.goahead.com/web-server/wsregister.htm. GoAhead requires that you fill in some personal data first. I figure they want to be the first to say they

knew you before you got famous by using their web-server code in your device.

Once the code is "yours," the downloaded file should be unzipped into a directory called `webserver`. There are numerous directories under `webserver`, including a subdirectory for each of the publicly ported GoAhead WebServer operating environments.

A `Makefile` is included in the individual OS directories for each OS compiler. In addition to the `Makefile`, there's also a main program that invokes and initializes the server.

I pick on Bill, but I also use Bill's stuff a bunch, too. This time around I'll use the well-known-but-not-so-embedded-oriented Windows NT. One reason for using NT is that it's already running on the *Circuit Cellar* Florida Room network.

Windows NT isn't usually thought of as embeddable, but it's on the GoAhead already-ported list. Thus, Windows NT workstation 4.0 is a perfect springboard for embedding the server on another platform.



Photo 1—This whole GoAhead web server thing is a sleeper until you get to this point. You can dig through the HTML files for answers until you're blue. The answer is to just install it!

I'm using Windows NT here to explore, not deploy, an embedded version of the server. In reality, NT and GoAhead WebServer would work well together in an Intranet- or Internet-application environment.

This server incorporates several features, including Active Server Pages (ASP). ASP is another one of Bill's concoctions developed to serve web pages with dynamic content. ASP documents are delimited by an .asp extension and use embedded scripting to insert dynamic data before a page is sent to the user's browser.

The GoAhead WebServer supports an open-scripting architecture. This enables scripting engines to be selected at run-time with the possibility of individual pages using multiple scripting engines at will.

JavaScript is the only scripting engine you'll hear about in this segment because the GoAhead WebServer V.2.0 supports Embedded JavaScript natively. The package even includes an embedded-oriented JavaScript API set.

The current scripting engine is specified by the `language=` specifier at the beginning of the ASP script. The language last specified remains the default until a new `language=` keyword is encountered at the start of another ASP script. If no language is specified, the default language is JavaScript.

It's recommended that the language be specified in the `<HEAD></HEAD>` section of your page, as in:

```
<% language=javascript %>
```

JavaScript is easy to learn, and its syntax resembles models you are already used to. To create an ASP script field in a GoAhead WebServer ASP document, use the `<%` and `%>` ASP delimiters. For example:

```
<h1>You are reading <% write  
("Circuit Cellar APC"); %></  
h1>
```

will output "Circuit Cellar APC" in place of the ASP-delimited field.

I won't write a dissertation on ASP in this column, but here's a quickie on just how the ASP process executes. When a user's browser requests an ASP document,

the default URL handler determines if the page is an ASP document by examining the file extension. If .asp is the extension, then ASP processing is invoked.

The document is read from the file system or ROM store in a one-pass operation. ROM store? Yep. Along with the new Embedded JavaScript parser, V.2.0 supports another new feature called webcomp. Webcomp is a web-page compiler that generates ROMable web pages and source code for systems that don't have a file system.

Getting back to the process following the one-pass read, text before the ASP delimiters is copied directly to the requesting browser. Any text found between ASP delimiters is passed to the relevant scripting engine (JavaScript here) for execution. The postscripted text is immediately passed back to the browser and the line-by-line process continues until the end of the document.

If you're familiar with ASP, you've probably noticed that the process I described is a bit different from other ASP implementations you may have encountered. GoAhead WebServer doesn't buffer the entire ASP output and thus doesn't permit scripted iteration over HTML tags.

It doesn't support these methods because they require the entire ASP document to be sucked into memory before being processed and returned to the browser. The documentation suggests that *.htm files be used for large amounts of output data.

In addition to those features, this server includes support for in-memory CGI processing, security, and URL handlers. All the details on those goodies can be found in the API documentation available from the demo server setup web pages.

SPINNING UP A WEB SERVER

To be honest, before I downloaded the GoAhead WebServer V.2.0 code, I really didn't think I would get this to work. In my experience, most of the stuff you get for "free" is worth exactly what you pay for it. So, I prepared myself for bunches of bit twiddling and customization.

I was also concerned with the lack of online documentation the web site kept alluding to. When I couldn't find the how-to stuff I thought I'd need, I checked out the newsgroup. I found some interesting

Listing 1—An oasis of words. Guess which set of instructions I executed?

To build and run the GoAhead WebServer, change to the relevant operating system directory and use make to initiate the build. Some of the make or batch files may need to be modified for the configuration of your system or the target system. See the "Configuring the GoAhead WebServer" section.

```
For VxWorks:
  cd VXW486
  make
Load webs.o onto the target system and use standard VxWorks
procedures to load the program into memory and execute it.
```

```
For Windows NT:
  cd WIN
  nmake /f webs.mak
  webs
```

To stop the web server, right click on the taskbar icon for the GoAhead WebServer and select "Close".

entries, but nothing I could use at the time.

One entry even asked, "Where or how do I start?"

With that, I was determined to answer that question for myself.

Not knowing what I was getting into, I prepared a Windows NT 4.0 workstation with lots of disk and memory space and a hefty processor. I loaded up every service I thought I would need.

Just in case I needed access to external utilities from other machines in the lab, I added the web-server machine to the lab domain. This arrangement would also make it easy to test and troubleshoot my newfound web server. I assigned the IP address of 10.10.0.1 to the initial test web-server mothership. Finally, I laid in Bill's C++ V.5, Netscape Communicator 4.5, and downloaded GoAhead Web-Server V.2.0.

Before running the unzip process, I opened the zip file with WinZip and looked over the package. I was overwhelmed with the amount of .htm files with names that looked like they might point me in the "get started" direction.

I started looking through them and got the scent, but I still couldn't tree anything. (For those of you consulting your tech manuals for the "tree" process, that's a southern hunting term for the process of cornering and thus "treeing" your game. As this term is usually invoked when hunting squirrels and opossums, "tree" is a very appropriate term here.) Eventually, I found a couple of README files and the

Listing 2—This snippet uses the JavaScript API. All JavaScript API members begin with ej. The definition for aspTest can also be found in the main.c code example.

```
// Test Javascript binding for ASP. This will be invoked when
// "aspTest" is embedded in an ASP page. See web/asp.asp for
// usage. Set browser to "localhost/asp.asp" to test.

static int aspTest(int eid, webs_t wp, int argc, char_t **argv)
{
    char_t *name, *address;
    if (ejArgs(argc, argv, T("%s %s"), &name, &address) < 2)
    {
        websError(wp, 400, T("Insufficient args\n"));
        return -1;
    }
    return websWrite(wp, T("Name: %s, Address %s"), name, address);
}
```

chase began. (I've never done the fox and horses thing, so let's move on.)

The README.TXT file was most helpful. The first paragraph apologized for the seeming lack of documentation and reminded me that this was a beta version. I was assured that full documentation would be provided with the full release of V.2.0. I was then instructed to consult the home page (//localhost/home.asp) for access to the beta documentation.

As I read further into the README text, I was also assured that I had everything I needed to compile and run a GoAhead WebServer. I was beginning to feel better, but I still had no clue until I read the blurb you see in Listing 1. With this discovery, I decided to GoAhead and unzip the web server package and attempt the compile.

Everything was unzipped and ended up in the webserver directory. Just like the README file said, there was a subdirectory for every supported OS and a gaggle of C files and C header files spattered about the webserver directory.

It wasn't difficult to determine that WIN was the directory I needed. The README text in Listing 1 pointed to it, and WIN stood out among other subdirectories such as Linux and Qnx4. I was told that I had everything, so I ran the NT compile commands in Listing 1.

The linker gave me the hi sign. The compile and link were successful. To my surprise, after executing webs, I had a working GoAhead Web-

Server! Well, at least according to the System Tray GoAhead WebServer entry. The only way to know for real was to serve up a page from 10.10.0.1.

I added an entry to the Host file of another workstation to map GoAhead to 10.10.0.1. On the same workstation I started Netscape and entered http://goahead and the home page shown in Photo 1 appeared in the browser window.

At this point, everything I was looking for was now in one place. All of the documentation could be accessed from the browser. By following the included source code, I was able to easily follow the flow of how the server and its Embedded JavaScript and API sets worked together.

As it turns out, I got the server to operate without twiddling bits or special customization. This isn't to say that it cannot or should not be customized. The idea behind GoAhead WebServer is to welcome any type of customization.

There's a section that describes some basic GoAhead WebServer configuration switches, but the server is intended to provide a basic platform for the web developer to build on. I must agree that everything needed to assemble a fully functional web server is included for free. For that person in the newsgroup who was pleading for help, just go after it like I did. You'll be pleasantly surprised.

main.c and socket.c in the WIN directory describe how the server is initialized. These files also show how to map web-server functions to C code and the API sets. Listing 2 is a code snippet from main.c that describes how a test JavaScript applet is coded behind the scenes.

The test applets are included with the server code and can be accessed from the



Photo 2—This embedded PC is a desktop in embedded clothing. You can get this baby with Pentium power, too.

home.asp page. The whole GoAhead WebServer package is wide open like this. Proprietary is a bad word here. Within this server, there's enough API functionality to fill three or four columns.

As I've shown, it's a snap to put this server up. The key is to install the WebServer prototype, which unlocks the soul of the product. So, download a copy and install it, and I'll leave the reading to you.

THE HARD(WARE) PART

Their home page states that GoAhead WebServer was designed expressly for the embedded web-server developer. This means that the footprint must be small and the functionality must be immense.

Now that I'm over my skepticism of the software, how does it take to various types of hardware? My first thought was to find every piece of embedded hardware in the Florida Room and attempt to install the server and serve some pages.

RIDING A SNAKE

The first piece of hardware I came across was a Teknor VIPer806. This board is quite broad as far as peripherals are concerned. My VIPer has a 100-MHz '486DX2 processor, 16 MB of RAM, an SMC Ethernet adapter, the usual complement of parallel and serial ports, IDE and floppy interfaces, keyboard and mouse ports, an SVGA adapter, and bunches of other stuff that you would normally find on desktop machines.

I figured this little monster would easily handle a load of NT and the server software. I see the VIPer as an embedded desktop.

To make things easier, I dug out an ISA-based Tempustech backplane. Using a standard passive backplane makes changing embedded platforms easy. The power plane is an integral part of the backplane, and the embedded engines I will use (or attempt to use) plug right into the ISA socket strips.

Because NT is being used as the test platform, a hard disk and a CD-ROM drive were used here too. Of course in an actual embedded implementation, I would employ the flash memory on the VIPer806 and use a smaller or embeddable OS. I'm sure I could fit NT into an embedded platform by tossing out some of the fat, but that's not why we're here today.

If you're familiar with Windows NT, you know it's fat and robust, and that there are



Photo 3—It may be cheap and it may be ugly, but it runs!

many ways to load NT. I configured the VIPer to use the hard disk, floppy, and CD-ROM just as a desktop would.

I attempted to load NT Workstation using the three install diskettes. Everything went fine until the reboot after the text install phase. Seems the VIPer couldn't find the boot files on the hard disk and just hung up.

After a few hours, I gave up and decided that I had somehow damaged the VIPer or the hard disk was loopy. So, I loaded Bill's DOS 6.22. I knew I couldn't boot from CD-ROM and load NT because the BIOS in the VIPer wasn't at the latest level and didn't mention CD-ROM support.

I knew I would eventually have to use the CD-ROM to complete the load of NT. DOS was loaded and operational, and the VIPer seemed to be OK. So, why not load the CD-ROM driver under DOS and perform the NT install from the CD-ROM?

It took some time, but I finally completed the install and hooked the VIPer into the lab domain. Loading software isn't always a piece of cake, and I had some trouble getting the VIPer806 Ethernet drivers to take, thanks to a signature on the original VIPer driver diskette that NT didn't like.

I ended up imaging the diskette to a directory on the VIPer hard disk, and I got rid of the offending file by process of elimination. After getting the SMC 8416 Ethernet card to load, the server code went on as easily as it did on the mother-ship, and I was serving pages in no time.

If you're going to run NT on the VIPer806, be sure to use plenty of RAM. It's a wet dog in the cold with 16 MB of RAM. And for you NT gurus, no, I didn't check the NT hardware compatibility list to see if the VIPer or the CD-ROM drive was tested or accepted.

The way I see it, if I can make it work with troublesome and problematic hardware, just think of how big a hero you'll be by making it work with approved equipment. The VIPer setup is shown in Photo 2.

GOAHEAD ON THE CHEAP

I've gotten GoAhead WebServer to load and run on two pretty rich machines with not-so-uncommon backgrounds. Let's step it down a notch. Photo 3 is the reduced-instruction setup. The "downgraded" web server is based on an inexpensive Tempustech VMAX SBC301.

The VMAX board is a '486SXL running at 66 MHz with only 8 MB of RAM. That means Windows NT won't fly. Just for grins I tried loading Windows 98.

Did you know Win 98 requires a math coprocessor? Did you know the program tells you if you don't have one? OK, I get it. Windows 95 it is.

In addition to the VMAX CPU board, the "downgraded server" hardware includes a \$14 SVGA board, an SMC8416 Ethernet adapter, the VIPer spinning disk set, and the Tempustech ISA backplane.

During the Win 95 WebServer install I encountered an interesting anomaly. I used a 2-GB IDE drive for the VIPer configuration and attempted to do the same with the VMAX board set. The VMAX wouldn't recognize or configure the 2-GB drive. I couldn't even manually enter the cylinder/sector count data in the setup fields. The VMAX BIOS maxed out at 2048 cylinders and the 4092-cylinder 2-GB drive hung the system at startup.

The smallest drive I had on the bench weighed in at 1.7 GB. I figured the VMAX would choke on that one, too. So, I resorted to a 250-MB Western Digital I found in a box in the corner of the lab.

After I installed the little WD drive, the VMAX went into autoconfiguration mode on powerup and found the smaller drive, just like the manual said it would. Sometimes bigger isn't better.

The good news is that I have some clear spinning media looking for some new code. The bad news is that I'm sure

I'll have to shoehorn every bit of that new code including the OS, Bill's C++ compiler, and the server onto it.

Only 225 MB later, I was serving web pages to the domain members from the VMAX platform. But, I wasn't able to successfully compile the server code in the native VMAX environment (probably because of the lack of real memory and environment space).

I copied over the server executable from the mothership via the LAN. VMAX didn't know the difference. The VMAX "stack" is shown in living color in Photo 3.

GOING AHEAD

I'm a firm believer in the future of anything that's embedded and uses the Internet or Intranet to control and monitor remote devices. GoAhead WebServer is a step toward bringing this technology to all of us faster. With Windows NT, this server is a good way to start or extend your understanding of embedded web-server technology.

As you've seen, Windows NT isn't necessarily the best (or the only) solution for projects such as this one. The real beauty of this freeware concept is that you can use this software with any embedded platform you can port it to. Not to mention that having access to the free code gives you a head start on making a difference in the embedded control world. [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

Windows NT

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

GoAhead WebServer

GoAhead Software, Inc.
(425) 453-1900
Fax: (425) 637-1117
www.goahead.com

VIPer806

Teknor Microsystems, Inc.
(514) 437-5682
Fax: (514) 437-8053
www.teknor.com

VMAX

Tempustech
(800) 634-0701
(941) 634-2424
Fax: (941) 643-4981
www.tempustech.com

FEATURE ARTICLE

Jon Varteresian

Hands-On PIC Trainer

Programming in Assembly

Sometimes it takes a simple project to learn the practical (and fun) applications of a basic technology. That's the idea behind Jon's assembly-coded device that provides immediate visual feedback without a simulator.



If I can borrow a line from Fred Eady, I'd have to agree, "It doesn't have to be complicated to be embedded." Sometimes it can be simple. Sometimes it can just be fun. And I'm sure you already know that doing something fun is a great way to learn the basics.

The assembly-language firmware described in this article demonstrates basic PWM techniques, timer operation, and general I/O manipulation. The information presented here will enable you to tackle a wide range of embedded-control problems.

I designed a simple electronic safety device for nighttime joggers—or even, given that it's October, trick-or-treaters. Its high-brightness LEDs alert oncoming traffic to your presence.

This device is built around Microchip's PIC16LC54A. This microcontroller includes an 8-bit data path, 12-bit instructions, an 8-bit timer, 512 bytes of ROM, and 25 bytes of RAM.

As for power, typically, coin cell battery-operated devices and LEDs aren't a good match. A current-hungry LED can drain a battery in mere hours. Having more than one LED on at any one time drains the battery even faster. Unless you use

some kind of intelligent power-conservation technique, your battery bill is going to be quite high.

Several power-conservation techniques can be used in this design, including:

- lowering the LEDs' forward current—this extends battery life but at the cost of the wearer's safety
- enabling one LED at a time, cycling through all eight—this helps preserve the battery, but the visual pattern of the LEDs is limited, if not boring

It's more visually appealing if there were two (or more) LEDs on at any given time. This pattern can be accomplished by turning on LED A for a short time while B is off, and then turning A off and B on.

If the cycle repeats fast enough, the human eye can't detect the flashing. So, it appears that both LEDs are on, granted at a slightly reduced light-level output.

That's the technique employed in this firmware. It produces an ever-changing, multi-LED pattern even though only one LED is conducting at any given time. This technique extends the device's 3-V, 150-mAh battery life to well over 10 h.

First I'm going to describe the assembly-language programming details, and then I discuss how to build the project. Building the device is a piece of cake—it uses surface-mount and through-hole components on a single-sided PCB. All you need is a fine-tipped soldering iron and, depending on your eyesight, maybe a magnifying glass.

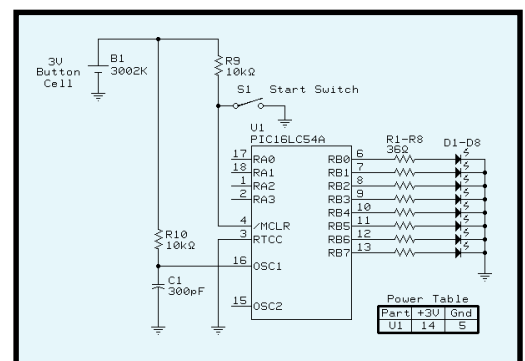


Figure 1—The start switch (S1) that activates the LEDs is connected to the microcontroller's master clear input pin (*MCLR).

CIRCUIT OVERVIEW

The circuit shown in Figure 1 comprises a microcontroller (U1, PIC16LC54A), eight LEDs, one switch, one battery, and a sprinkling of resistors and capacitors.

When you press and release the switch, the PIC's program counter is reset to the beginning of code space and the lighting sequence begins. To turn the LEDs off, press and release the button again.

Power is supplied to the microcontroller and LEDs through the battery (B1), which is a standard 20-mm 3-V coin cell. Resistor R10 and capacitor C1 generate an RC clock input for the PIC. The operating frequency is a nominal 256 kHz, which means that the instruction rate inside the PIC is:

$$\frac{1}{256 \text{ kHz}} \times 4 = 15.625 \mu\text{s}$$

The RC operating mode of the PIC is inherently imprecise and can vary as much as 20%. However, this application is relatively timing-insensitive. Resistors R1–R8 (36 W) limit the current through the LEDs to 25 mA.

Note that, besides the LEDs, all of the components are of the surface-mount variety. If you decide to get creative and put artwork on the opposite side of the PCB, through-hole components would get in the way.

FIRMWARE

This device is a useful educational tool because when you experiment with the firmware's assembly code, you get immediate feedback. Did you program it right? You'll know right away.

Everything begins with a low-to-high pulse on *MCLR. Once this pulse is generated, it resets the pro-

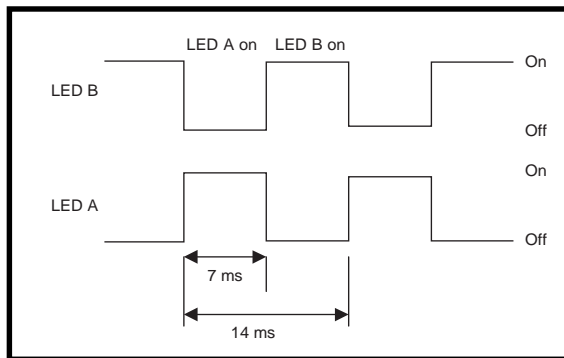


Figure 2—As this LED enable waveform shows, LED A is enabled while LED B is disabled, and then the roles are reversed.

gram counter of the PIC to the beginning of code space.

The PIC initializes all software variables, configures all I/O, and sets up timer0 for internal clocking and a prescaler value of 4, so every time the internal instruction clock increments four times, the timer increments by 1. Remember that the PIC is executing instructions every 15.625 μs .

Next, the PIC inverts `Power_cycle`. This variable decides whether to begin flashing the LEDs or put the PIC into Sleep mode. If `Power_cycle` is a 1, the PIC begins flashing the LEDs. If it's a 0, the PIC goes to sleep.

Remember, software variables don't lose their state in Sleep mode. The only time they lose their state is when power is removed. By inverting `Power_cycle` every time someone presses the momentary button, the device effectively turns on and off with every push.

The PIC loads timer0 with a value equal to 7 ms. Note that the timer counts up and is considered to be at its terminal count when it rolls from 0xFF to 0x00.

Normally when a timer rolls, an interrupt is generated, causing an interrupt handler to handle the event.

Because the '16LC54A doesn't have interrupt capability, the counter must be manually polled. When it finally rolls, the PIC reloads the timer to count another 7 ms, executes the next section of the code, and then returns to the tight loop before the 7 ms are up.

Before we can discuss what happens when the 7-ms counter rolls, we need to take a look at how the LEDs are enabled. First, let's define a rule that restricts how the firmware operates.

Because the battery is small and can't provide a lot of instantaneous current, only one LED can be enabled at any given time. So, if you want two LEDs on at once, you'll have to find another way to do it.

Let's assume that you want two LEDs (A and B) to appear to be on at the same time. The device does this by turning LED A on and LED B off for a short period of time. Then LED A is turned off and LED B is turned on for the same short period of time. The process is then repeated continuously.

If the LEDs are turned on and off fast enough (faster than 60 times per second), human eyes can't perceive that the diodes are actually turning on and off. Although their brightness is reduced, both LEDs will appear to be on at the same time.

The LEDs are turned on and off every 14 ms (71.5-Hz rate). That explains the reason for the 7-ms counter. It equals half of the on/off period.

If you only want one of the LEDs to be lit, you would still need to use the above steps except that LED B would be absent. LED A is lit for a short amount of time and then it is shut off for the same amount of time.

Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5	Pattern 6	Pattern 7	Pattern 8
1	2	3	4	5	6	7	8
8 and 2	1 and 3	2 and 4	3 and 5	4 and 6	5 and 7	6 and 8	7 and 1
7 and 3	8 and 4	1 and 5	2 and 6	3 and 7	4 and 8	5 and 1	6 and 2
6 and 4	7 and 5	8 and 6	1 and 7	2 and 8	3 and 1	4 and 2	5 and 3
5	6	7	8	1	2	3	4
6 and 4	7 and 5	8 and 6	1 and 7	2 and 8	3 and 1	4 and 2	5 and 3
7 and 3	8 and 4	1 and 5	2 and 6	3 and 7	4 and 8	5 and 1	6 and 2
8 and 2	1 and 3	2 and 4	3 and 5	4 and 6	5 and 7	6 and 8	7 and 1
1	2	3	4	5	6	7	8

Table 1—Here's the order in which the LEDs are enabled for each of the eight patterns. When Pattern 8 is completed, the cycle restarts at Pattern 1.

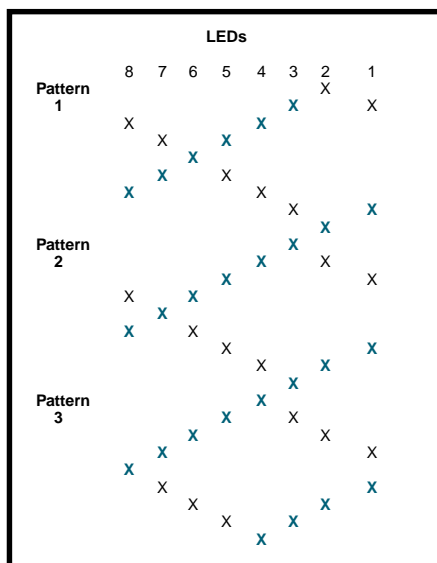


Figure 3—Seeing the order in which the LEDs are lit for the first three patterns makes it easier to extract the formulas to describe each LED pattern.

Figure 2 shows the LED-enable waveforms and the on-off periods.

Now, let's define a pattern. Each pattern starts as a single lighted LED and then travels in pairs around the circuit board until reaching the opposite side where it ends in a single LED and then travels back. At that point,

the single LED rotates one LED clockwise and the pattern repeats.

The eight possible LED patterns are shown in Table 1. Note that each subpattern is enabled for $14 \text{ ms} \times 8$, which equals 112 ms.

Now, we need to find a way to code that pattern without explicitly coding each step. Although there's no reason why a brute-force method of coding the pattern wouldn't work, it's just not very challenging. There has to be an overlying pattern that we can use to generate each subpattern.

To find this overlying pattern, take an 8-bit register and put an X in the bit location when you want an LED to turn on. If you start with Pattern 1, you'll end up with Figure 3. The LEDs are numbered from 1 to 8, and the registers are numbered 0 to 7.

Figure 3 should help you figure out the rules for determining which diodes should light. Note the cells with the bold blue X in them. This pattern is nothing more than a circular left shift.

Describing the cells with the non-bold blue Xs is only a little more complicated. It's a circular right shift with

a twist. The twist is that every ninth shift is a left instead of right. Now, by inclusive-ORing these two shift patterns together, you can generate the patterns you want.

This circular right shift technique is much easier than trying to code all the individual subpattern states. The two registers that contain the shift patterns are called `Reg_chain1` and `Reg_chain2`.

Once the 14-ms cycle time is defined and the patterns described, we need to determine which LEDs will be on and off within the subpatterns. This dilemma is resolved by taking the first subpattern in each pattern and defining the A LEDs to be the bold-X LEDs plus $X + 1$, $X + 2$, and $X + 3$. Each of the remaining four LEDs plays the part of the B LED.

If you try to assign a fixed pattern to the LEDs (e.g., LEDs 1–4 are A, and 5–8 are B), it will result in some subpatterns having both desired LEDs in either the A or the B group. The variable `LED_pattern` contains the A or B designation information and is used by logical ANDing `LED_pattern` with

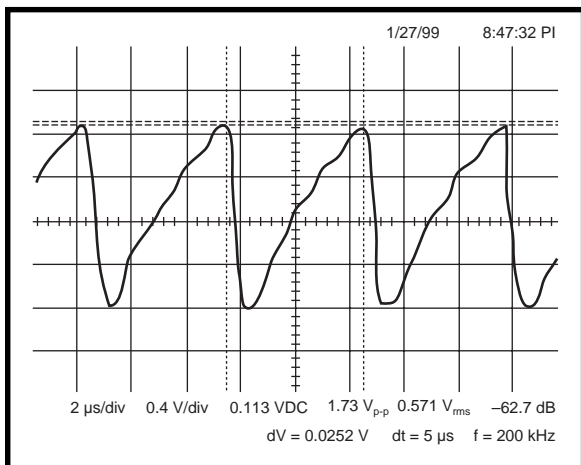


Figure 4—If everything has been done right up to this point, probing the PIC's clock signal (pin 16 of U1) with an oscilloscope should result in a waveform like the one shown here.

the inclusive OR of the `Reg_chain1` and `Reg_chain2` variables.

By coding the LED pattern in this manner, substantial code compression is achieved, which leaves valuable code space to handle other tasks.

By the way, if you wish to modify this firmware, you need a suite of tools available from Microchip. Most of the PIC tools are free, including the assembler and simulator, and can be downloaded the Microchip web site.

ASSEMBLY

Begin by programming the PIC with the source and programming files, which are available via the *Circuit Cellar* web site. Next, solder the PIC to the circuit board and note the location of pin 1.

Attach the switch (S1) and solder the 10-k Ω resistors (R9 and R10) and the 300-pF capacitor (C1). Now solder the eight 36- Ω resistors and the coin cell battery holder (B1). And that completes the main circuit assembly except for the LEDs.

Of course, you can put a label on the front of the device. Then, using a common straight pin, poke 16 holes for the LEDs and push the leads through the circuit board. Bend the leads slightly to hold each LED in place while you insert the others.

Before you insert each LED, note the location of pin 1 (or anode). Pin 1, (identified by the longer of the two leads) must go to the square pad on the circuit board. When you're done, solder all 16 leads and trim the excess.

DID YOU DO IT RIGHT?

The last step is to insert the coin cell battery in the battery holder with the positive terminal positioned up. Now press and release the reset button (S1).

If you've assembled everything correctly, the LEDs will start to flash. If they don't, check your work for solder bridges (shorts) or faulty solder joints. If that doesn't help, use an oscilloscope to probe the PIC's clock signal (see Figure 4).

And there you have it—a simple way to learn programming in assembly on a PIC. 📺

Jon Varteresian owns and operates JV Enterprises, which offers educational electronic kits and provides consulting services. Jon specializes in deep embedded control of wireless transceivers and user interface management. You may reach him at jventerprises@worldnet.att.net.

SOFTWARE

Along with a complete parts list, the source and programming files for this project can be downloaded from the *Circuit Cellar* web site.

SOURCES

Project components

JV Enterprises
(978) 928-5655
jventerprises.home.att.net

Digi-Key Corp.
(800) 344-4539
(218) 681-6674
Fax: (218) 681-3380
www.digi-key.com

All Electronics
(800) 826-5432
(818) 904-0524
Fax: (818) 781-2653
www.allelectronics.com

Jameco
(800) 536-4316
(415) 592-8097
Fax: (415) 592-2503
www.jameco.com

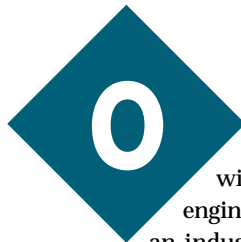
FEATURE ARTICLE

Hari Ramachandran

IrDA Technology

Part 1: An Overview

Cameras talking to PDAs, talking to PCs, talking over the Internet, talking to each other...? Once you get the latest word on IR communication standards, your embedded designs will want to join the conversation, too.



Our industry is rife with acronyms. Most engineers shudder when an industry association is formed and yet another set of acronyms is spawned.

In 1994, the Infrared Data Association (IrDA) was formed by a group of companies, including HP, IBM, and Sharp. The goal of the association was to promote the ubiquitous deployment of infrared appliances that interoperate with each other.

For this to happen, both the physical layer and protocols for communication needed to be established. Much to the credit of the companies in the IrDA, these speci-

fications were adopted and ratified in less than a year.

By 1995, laptop PCs and IrDA PC adapters that adhered to the standard made their appearance and Microsoft released IrDA support for Windows 95. Since then, in fits and starts, IrDA has been adopted in almost all laptops and is being used in consumer devices (e.g., digital cameras, cell phones, and pagers). However, the breadth of the technology and its applications has left even seasoned industry watchers and engineers a bit dazed and confused.

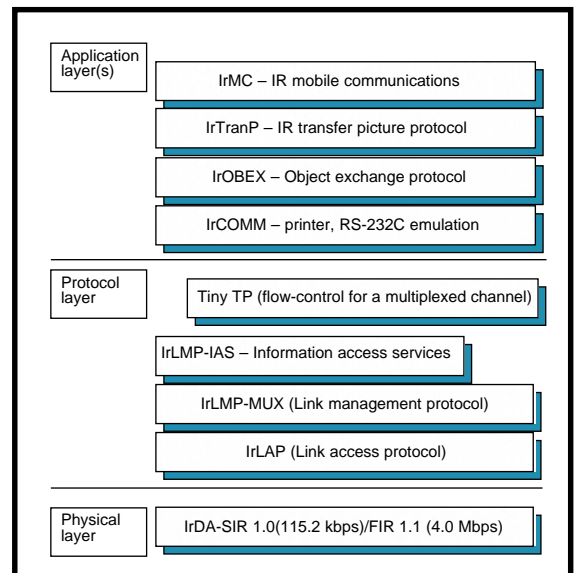
My goal here is to dispel some of the confusion and provide an overview of IrDA technology. I want to focus specifically on physical-layer implementations, which will be the focus of Part 2 next month.

THE VISION

Before delving into the details, it's a good idea to discuss the purpose of the IrDA. The key concept is ubiquitous interoperability—the ability to interconnect to a variety of devices, exchange files, electronic business cards, and images, print documents, and synchronize data between PDAs and desktop PCs, cell phones, and IR watches.

For the IrDA's goals to be reached, a number of key technical hurdles needed to be overcome. First of all, the physical-layer specification had to be ratified. These specifications included wavelength, transmission power, communication speed, and data-modulation schemes.

Figure 1—There are three major layers needed in a basic IrDA application—physical, protocol, and application. A basic IrDA link requires a reliable physical layer and IrLAP/IrLMP layers. What's loaded on top of those layers is up to you.



Signaling rate	Modulation	Rate tolerance % of rate	Pulse duration minimum	Pulse duration nominal	Pulse duration maximum
2.4 kbps	RZI	±0.87	1.41 μs	78.13 μs	88.55 μs
9.6 kbps	RZI	±0.87	1.41 μs	19.53 μs	22.13 μs
19.2 kbps	RZI	±0.87	1.41 μs	9.77 μs	11.07 μs
38.4 kbps	RZI	±0.87	1.41 μs	4.88 μs	5.96 μs
57.6 kbps	RZI	±0.87	1.41 μs	3.26 μs	4.34 μs
115.2 kbps	RZI	±0.87	1.41 μs	1.63 μs	2.23 μs
0.576 Mbps	RZI	±0.1	295.2 ns	434.0 ns	520.8 ns
1.152 Mbps	RZI	±0.1	147.6 ns	217.0 ns	260.4 ns
4.0 Mbps					
(single pulse)	4PPM	±0.01	115.0 ns	125.0 ns	135.0 ns
(double pulse)	4PPM	±0.01	240.0 ns	250.0 ns	260.0 ns

Table 1—The pulse durations for each signaling rate can be critical in building a reliable physical-layer implementation. Issues can arise if the pulse duration generated by the IrDA transceiver doesn't match that supported by the encoder/decoder.

Next, a robust reliable data communication protocol had to be established. The protocol needed to take into account the ad hoc nature of a wireless connection as well as support point-to-point connectivity and the ability for devices to negotiate capabilities like data frame size and maximum communication transfer rate.

And last, in order for devices to communicate, application layers for communication between cell phones, digital cameras, and PDAs needed to be established.

IrDA LAYERS

The end result of almost three years of discussions and various levels of execution is the IrDA architecture. Each layer is described in the sections shown in Figure 1.

At the lowest level, the physical layer specifies the physical characteristics of the infrared medium, the data-modulation scheme used, and the structure of an IrDA frame. Generally, the physical layer comprises a UART, a modulator/demodulator ASIC, and an IrDA-compatible transceiver. Tables 1–2 describe key characteristics of the IrDA physical layer.

For data rates up to and including 1.152 Mbps, the return to zero inverted (RZI) modulation scheme is used and a light pulse represents a 0. For rates up to and including 115.2 kbps, the optical pulse duration is $\frac{3}{16}$ of a bit duration (or $\frac{3}{16}$ of a 115.2 kbps-bit duration).

For 0.576 and 1.152 Mbps, the optical pulse duration is $\frac{1}{4}$ of a bit duration. Informally, communication rates up to 115.2 kbps are termed serial infrared

(SIR). A signaling rate of 1.152 Mbps is termed MIR, and a 4-Mbps rate, FIR. Figure 2 illustrates the basic building blocks for an SIR implementation.

The RZI (SIR) encoding scheme shown in Figure 3 relies on a clock to drive the modulation engine. This clock (16XCLK) is set to 16× the communication transfer rate.

For example, if communication at 115.2 kbps is required, 16XCLK is set to:

$$16 \times 115200 = 1.8432 \text{ MHz}$$

A space or 0 TXD value is encoded as a single pulse (IRTX) of duration ($\frac{3}{16}$) of the bit time or:

$$\frac{1}{115200} \times \frac{3}{16} = 1.63 \mu\text{s}$$

The demodulation scheme stretches the received pulses (IRRX) to recreate the original signal (RXD). A practical implementation of an SIR physical layer solution is described in Figure 4, which outlines a practical microcomputer-based SIR physical layer solution.

The key building blocks are the HP HSDL 1001 (an IrDA transceiver designed to accommodate signaling rates up to 115.2 kbps), an HP HSDL 7001 or Parallax's PLX 7001–IrDA Endec (applies the RZI [SIR] encoding scheme as described in Figure 3), and Dallas' 80C320 (8051 CPU derivate).

If run at 25 MHz, the 80C320 can easily accommodate communication up to 115.2 kbps. The UART signals (TXD and RXD) are tied to the HSDL 7001. Three control lines from the Dallas chip are used to set the signaling rate of the HSDL 7001.

Specification	Data rates	Minimum	Maximum
a)			
Peak wavelength, up, μm	All	0.85	0.90
Max. intensity in angular range (mW/Sr)	All	—	500
Min. intensity in angular range (mW/Sr)	≤ 115.2 kbps	40	—
Min. intensity in angular range (mW/Sr)	115.2 kbps +	100	—
Half-angle, degrees	All	±15	±30
Signaling rate (i.e., clock accuracy)	All	See Table 1	See Table 1
Rise time (Tr) 10–90%, fall time (Tf) 90–10% (ns)	≤ 115.2 kbps	—	600
Rise time (Tr) 10–90%, fall time (Tf) 90–10% (ns)	115.2 kbps +	—	40
Pulse duration	All	See Table 1	See Table 1
Optical overshoot, %	All	—	25
Edge jitter, % of nominal bit duration	≤ 115.2 kbps	—	±6.5
Edge jitter relative to reference clock, % of nominal bit duration	0.576 & 1.152 Mbps	—	±2.9
Edge jitter, % of nominal chip duration	4.0 Mbps	—	±4.0
b)			
Maximum irradiance in angular range (mW/cm ²)	All	—	500
Minimum irradiance in angular range (μW/cm ²)	≤ 115.2 kbps	4.0	—
Minimum irradiance in angular range (μW/cm ²)	115.2 kbps +	10.0	—
Half-angle, degrees	All	±15	—
Receiver latency allowance (ms)	All	—	10

Table 2a—To be compliant, IrDA optoelectronics must comply to these physical specifications as well as the irradiance specifications outlined in (b). **b**—Matching the transmission/receiver power of an IrDA optoelectronic pairing is critical. The IrDA protocol allows multiple devices to be “discovered,” so a device that transmits too much energy can saturate other listeners and drown out other devices.

Listing 1, written in 8051 assembly language, demonstrates how to set up both the Dallas 80C320 as well as the HSDL 7001 to send out the character "h" repeatedly.

Because the HP HSDL 1001 doesn't optically isolate the send and receive channels, you'll see data coupling back when you transmit data, causing an apparently spurious reception of data. When implementing a half-duplex link, be sure to accommodate this situation by clearing the receive buffer after sending out a frame or byte of data.

One other critical issue to understand when implementing an IrDA link is latency, which ties into the fact that there's no optoisolation between the send and receive channels. During transmission, the sender sends out a byte to the responder device, which causes the receiver on the sender to saturate, thereby causing a loss in its receiver sensitivity.

For the HSDL 1001, the latency is 10 ms. In that 10-ms period, any data received will most likely be corrupted. The trick is for the responder to hold on for 10 ms after receiving before attempting to send any data to the sender. In IrDA terminology, this time lag is known as the minimum turnaround time and is ordinarily handled by the infrared link access protocol (IrLAP) layer.

4PPM MODULATION

For 4.0-Mbps communication, the modulation scheme is 4PPM. 4PPM modulation is achieved by defining a data symbol duration (Dt), ordinarily set to 500 ns, and dividing this into four time slices, called chips. The duration of each chip is 125 ns.

Because the signaling rate is 4 Mbps, each Dt period works out to two data bits ($Dt = 500 \text{ ns} = 2 \times \frac{1}{4} \text{ Mbps}$). During a specific Dt , only one chip can be a logical one.

Despite the relative simplicity of the 4PPM modulation scheme, building a practical 4-Mbps hardware scheme is non-trivial. Simply put, the 4-Mbps datastream is fast! In addition to a 4PPM

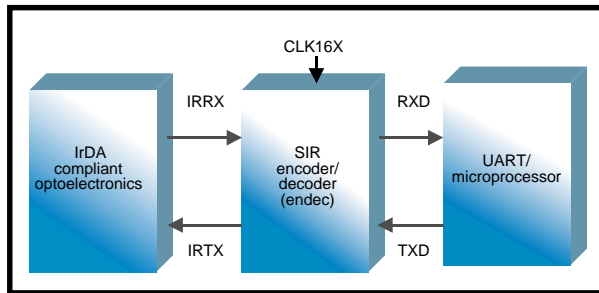


Figure 2—These building blocks are what's required for a basic SIR physical-layer implementation. You'll need an IrDA-compliant optoelectronic device, an SIR encoder/decoder chip, and a microprocessor with a built-in UART.

modulation engine, you need a serial communication controller that handles DMA if you're going to build a practical 4-Mbps hardware scheme.

Figure 5 outlines a relatively low-cost 4-Mbps hardware scheme. Let's look at the key modules for this solution. The HP HSDL 1100 FIR transceiver supports communication all the way from 9600 bps to 4 Mbps.

The NS 87109 UIR controller encompasses the 4PPM and RZI modulation schemes (supports 9600 bps to 4 Mbps). It also functions as a serial communication controller, and handles data integrity, bit stuffing, and CRC calculations (which, at 4 Mbps, are too complex to do in firmware).

The '87109 also supports DMA, which is another critical performance factor. Though a polled I/O approach can be used to receive 4-Mbps data, it requires too much CPU overhead.

Because effective communication with the '87109 using regular polled I/O is usually not workable, I chose the NEC V850E/MS1 single-chip processor. It supports up to 128-KB internal flash memory and up to 4 DMA channels, so it's ideal for I/O-intensive applications.

PROTOCOL LAYERS

Once a stable physical-layer implementation is put in place, the next step is to build the protocol layers.

First-pass processing of received IR frames is handled by the IrLAP layer. The IrLAP layer is a variant of the IBM HDLC protocol, modified to take into account the ad hoc nature of wireless connectivity.

The IrLAP layer controls discovery of devices within range, uniquely identifies those devices, and establishes a reliable, error-free communication channel. You can think of it as a glorified wire that's implemented using infrared.

IrLAP commands are passed up to the infrared link management protocol (IrLMP) layer, which allows the single IrLAP channel to be shared across multiple logical channels. This way, a single IrDA device can support multiple functions (e.g., faxing, printing, and LAN access) through different logical channels while using the same physical layer (a single IrLAP connection). IrLMP is like a switchbox that routes data to and from the IrLAP layer.

INFORMATION ACCESS SERVICE

Because the IrDA protocol is intended to be used with a variety of appliances, we need a way to identify the features supported by the device. The information access service (IAS) accomplishes just that. Think of the IAS as a Yellow Pages of services and features supported by the device.

For example, if the device is a camera, it advertises itself by an entry in the IAS database that indicates that it is a camera and includes information about an external device that can connect to it. The first thing a connecting device does is send a "query IAS" command to determine what the device is and what it can do.

Finally, on the protocol layer, is the tiny transporta-

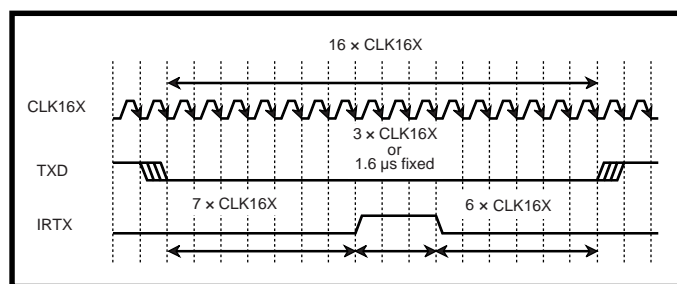


Figure 3—The SIR encoding scheme relies on a clock ($CLK16X$) set to $16\times$ the transmission bit time. The signal to be encoded is on the falling edge of TXD , an internal timer "times" seven of the $CLK16X$ pulses before pulling $IRTX$ high for three clock cycles and low for the remaining six cycles. SIR decoding is the inverse of this process.

Listing 1—The code I used to implement a simple 115.2-kbps communication link uses the hardware configuration from Figure 4 and programs the UART of the 80C320, sets up the 7001 for 115.2 kbps, and sends out a character "h." Don't confuse this code with a complete IrDA protocol stack implementation.

```

start:
; Do some initialization before proceeding.
  lcall Setup_HSDL7001      ; setup HSDL 7001
  lcall SetBaud115200      ; set data rate to 115.2 kbps
; Keep looping and sending out h's @ 115.2 kbps

doit: mov a, #68h ; send out an 'h'
      lcall PutByte      ; call PutByte to send out data
      mov a,#ffh
waitsend: djnz a,waitsend ; delay a bit before continuing
          ljmp doit      ; continue looping

PutByte:                ; sends byte out through UART
  mov r1,a              ; data is stored in the accumulator
  mov SBUF,r1          ; send the byte out
waitTI:  jnb  TI,waitTI ; wait till it's sent out
  clr  TI
ret

SetBaud115200:
;Program serial control registers & relevant timer registers
; to enable required data rate.
  mov  SCON, #50h
  mov  RCAP2H,#0FFh
  mov  A,#RCAP2L_115200
  mov  RCAP2L,A        ;0DCH -9600 bps- #0FDh 115200 bps
  mov  T2CON, #34H
  clr  RI
  clr  TI
  mov  IE,#0
  setb REN

; Select HSDL-7001 signaling rate - for 115.2 kbps, select:
; A0 = 0 -> tied to P1.0
; A1 = 0 -> tied to P1.1
; A2 = 0 -> tied to P1.2
;
  clr P1.0
  clr P1.1
  clr P1.2
ret

;Setup_HSDL7001 sets HSDL-7001 into the following operational mode:
; CLKSEL -> Internal oscillator
; PULSEMODE -> Set to 1.6-µs mode
; POWERDN -> Set Low (powerdown not active)

Setup_HSDL7001: ;Test hardware has been configured as follows:
; CLKSEL -> P1.3 LOW=> Internal Clock HI=> External 16XCLK
; PULSEMOD -> P1.4 LOW=> 3/16 modulation HI=> 1.6-µs pulse mode
; POWERDN -> P1.5 LOW=> Normal Operation HI=> Powered down
;
; POWERDN mode only deactivates internal analog oscillator block within HSDL-
; 7001. If CLKSEL is set to high and external 16X clk is provided,
; the chip should continue to work.
  clr P1.3 ; CLKSEL = Internal oscillator
  setb P1.4 ; PULSEMODE = 16 us
  clr P1.5 ; POWERDN = normal mode
ret

```

tion protocol (Tiny TP). Because the IrLMP layer supports multiple logical channels, Tiny TP is needed to prevent a deadlock situation in which one channel hogs the IrLAP/IrLMP channel. It uses a credit transaction scheme to control (or buy) access to the channel.

APPLICATION LAYERS

After the physical and protocol layers are built up, the next step is to build application support for the device(s) you want to communicate with. I'll just give a preliminary summary of the application layers here.

The main methods of exchanging data through an IR connection are IrCOMM, IrOBEX, and IrTRANP.

IrCOMM is intended for legacy-wired serial or parallel port applications that are to be converted to support IrDA. For instance, if you want to replace the RS-232 wired connection between a modem and a PC, you need to connect an IrDA adapter to your PC and build an IrDA modem adapter.

If the IrDA stack on your PC and modem adapter contain the IrCOMM extensions, both your PC and modem continue to operate as if the RS-232 cable was connected. You can continue to use the PC modem application as if you were using a virtual COM port.

If you install the Windows 95 IrDA drivers, the IrDA protocol is exposed as virtual COM and LPT ports on your system. The IrCOMM layer essentially opens up the IrDA protocol through a series of communication API look-alike commands, so changes to the base application aren't required.

The infrared object exchange (IrOBEX) standard is intended as an OS-independent method of transferring objects. If you want to send a file from one device to another, IrOBEX establishes the means of identifying and converting the file into a universal object that both devices can understand and interpret. It also specifies a simplified protocol of putting and getting objects from devices.

Sony, Sharp, Casio, and a few other digital camera vendors established the infrared picture transfer protocol (IrTRANP). IrTRANP enables users to beam images between cameras, PDAs, PCs, and even directly onto the Internet.

An application developed by NTT for the Nagano Olympics enabled users to beam their images onto a personal web page via an IrDA kiosk. IrTRANP

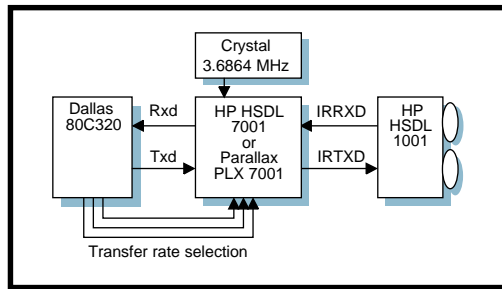


Figure 4—This design uses a Dallas 80C320, the HP HSDL 7001 (encoder/decoder), and HP HSDL 1001 (IrDA optoelectronics) and is all that's necessary to build a working IrDA (SIR) hardware implementation. To complete the full IrDA implementation, you'll need to code the IrDA protocol stack, which can be the most daunting aspect of implementing an IrDA solution.

used the IrCOMM layer to open a reliable communication medium between devices, a simple command execution protocol (SCEP) to manage the communication session, and binary ftp commands to exchange images.

SAMPLE COMMUNICATION SESSION

Now you've been introduced to a set of incredibly poetic-sounding acronyms and nifty concepts. But how does the IrLAP layer talk to the IrLMP layer, and how does IrTRANP come into the picture? Let's look at a session between a PDA and an IrTRANP camera.

The user selects an image from a IrTRANP camera, points it at the PDA, and presses Send. All communication is at the default IrLAP 9600-bps signaling rate. The camera sends an IrLAP discovery command, attempting to find IrDA-compliant devices.

After the PDA responds and identifies itself, the camera sends a negotiation request (to determine the PDA's maximum communication speed) and connects at that speed. Once an IrLAP connection is established, the camera initiates an IrLMP connection to channel 0 (IAS channel) and queries whether the PDA supports IrCOMM services.

The PDA responds, "Yes. I support IrCOMM, go ahead and connect to channel x, which is my IrCOMM channel." The camera disconnects from the IAS channel (channel 0), connects to the IrCOMM channel (channel x), and sends an IrTRANP connect command through the IrCOMM channel.

After the PDA acknowledges the IrTRANP command, the camera executes an IrTRANP put command and sends the image over. Once the image is transferred, the camera shuts down the IrTRANP, IrCOMM, IrLMP, and finally, the IrLAP session.

NOW YOU KNOW

Although the vision espoused by the IrDA is simple, understanding the technical aspects can be somewhat daunting. Now that you've been introduced to the concepts of IrDA, I hope that you'll find it easier to implement your own IrDA solutions. ☐

Hari Ramachandran is the managing director and founder of Parallax Research, a design company focused on providing IrDA solutions. He designed the HP HSDL 7001 and HP HSDL 7000 IrDA chips, and developed IrDA protocol stack software for numerous companies and specific applications. You may reach him at hari@parallax.com.sg.

SOURCES

80C320

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

HSDL7001, '1001, '1100

Hewlett-Packard
(408) 435-4303
Fax: (408) 435-4303
www.hp.com

PLX7001

Parallax Research Pte Ltd.
+65 791 7388
Fax: +65 793 0086
www.parallax-research.com

NS87109 UIR

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

V850/MS1

NEC Electronics
(408) 588-5008
Fax: (408) 588-5017
www.necel.com

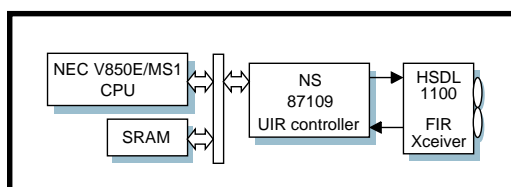


Figure 5—Building a 4-Mbps IrDA solution is a more complex (and more expensive) undertaking than building a basic SIR solution. This practical building block uses an NEC V850E/MS1 CPU and NS 87109 universal IR controller chip and the HSDL 1100 IrDA optoelectric assembly.

MICRO SERIES

Monte Dalrymple

Rolling Your Own Microprocessor

Design Application with the Rabbit-80

Part
2
of
2

Prick up your ears as Monte

takes a detailed look at the ports, timers, interrupts, and other features that make this new microprocessor a hair (er, hare?) better than the Z80.



Last month, I covered the design and debug process for the Rabbit-80 micro.

Now, I'd like to introduce you to the chip's features. The Rabbit-80 is built around an enhanced Z80-compatible CPU and contains a set of on-chip peripherals designed to address typical embedded-control applications.

Note that no on-chip memory is included. The majority of applications served by Z-World controllers require more flash memory and RAM than we could include on-chip at a reasonable price, especially given the low prices of individual 256K × 8 flash memory and 128K × 8 RAM devices.

Let's look at each of the blocks of Figure 1.

THE CPU

For the most part, the CPU in the Rabbit is Z80-compatible. We made changes to the instruction set to provide a greater

degree of support for the C language. Several seldom-used instructions (see Table 1a) were removed in favor of the new instructions listed in Table 1b.

Many of the new instructions provide 16-bit operations that weren't readily available in the Z80. These additions provide a big performance boost in the math libraries, especially with the new 16 × 16 signed multiply. The MUL instruction requires just 12 clocks to provide a full 32-bit result.

Operations such as the stack-relative load and store, and the ability to directly allocate or deallocate stack space simplify parameter passing to the stack. This feature is a must for C-language support.

The original Z80 architecture had dedicated I/O instructions that required specific registers and addressing modes. Instead, the Rabbit uses two prefix bytes, which precede an instruction intended to access I/O.

The prefix bytes identify whether the desired I/O location is internal or external, and change the timing and control signals appropriately to provide access to the peripheral. This setup gives greater flexibility when accessing peripherals because the prefixes can be used with any instruction that accesses memory.

A similar approach provides access to the Z80 alternate register set. The ALTD prefix switches the destination register to the corresponding register in the alternate register set, which cuts down on the bank switching necessary when using the alternate registers.

The final additions support the large physical address space. Loads

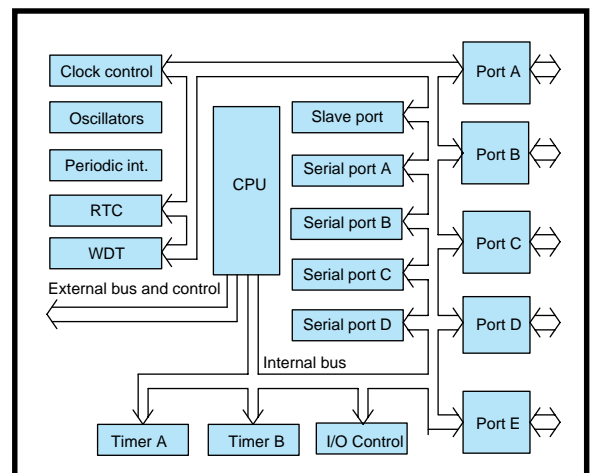


Figure 1—This is what the Rabbit-80 looks like on the inside. Only external memories are required.

and stores use a direct 20-bit physical address instead of the usual 16-bit logical address. The program flow-control (jump, call, and return) instructions load one of the MMU bank registers in addition to the PC to provide program access to the full physical address space.

MEMORY MANAGEMENT

The Z180 provides an MMU with two bank registers to create the larger (1 MB) physical address space. The Rabbit adds another bank register and the ability to bank-switch the memory chips for even more program and data space. The MMU address translation algorithm is shown in Figure 2.

The MMU uses the upper four bits of the logical address to select one of the bank registers to use for address translation. The exact boundary where one bank ends and another begins is controlled by another register.

The exception is a newly added bank register, which is always used for translation when the logical address is between E000h and FFFFh. This new bank register is accessed by the new program control instructions.

Translated addresses may be modified to separate instruction and data areas in the memory chips. This arrangement only works for accesses using one of the bank registers and allows execution out of RAM while accessing data in flash memory. Address bits 16 or 19 (or both) can be inverted for data access, so data can be accessed at either 64 or 512 KB above or below the code.

At this point, the address goes to the chip-select decoder, which uses the upper two bits of the 20-bit address to generate three chip-select signals, two write-enable signals, and two output-enable signals. Each memory quadrant can use any combination of a single chip select, an output enable, and a write enable. Figure 3 shows a typical system configuration.

The chip-select signal typically used for RAM can be forced continually active to circumvent the delay inherent in routing the signal through an external power controller for battery-backed applications. Each memory quadrant can be individually write-protected.

Instruction type	Mnemonics
a)	
Conditional call	CALL cc
BCD operations	DAA, RLD, RRD
Block I/O	INI, OUTI, etc.
Byte multiply	MLT
Interrupt control	DI, EI, IM n, RETN
I/O	IN, OUT, TSTIO, etc.
Test	TST
b)	
Alternate register bank	ALTD
instruction prefix	
I/O instruction prefix	IOIP, IOEP
16-bit math	AND, OR, RL, RR, BOOL, MUL
Stack-relative addressing	LD (SP + n)
Stack-space allocation	ADD SP,d
Long address program flow	LCALL, LJP, LRET
Interrupt priority stack	IP n, IPRES, PUSH IP, POP IP
16-bit load and store	LD
Load/store (physical address)	LDP

The final address modification allows the two most significant bits of the address to be flipped (individually for each memory quadrant) after the chip-select decision is made. This arrangement allows a 1M × 8 memory chip to be bank-switched within a 256-KB quadrant of the physical memory space.

BUS TIMING

The basic instruction time and bus-cycle time are both two clock cycles. Memory-write cycles have one wait state automatically added. The basic read cycle is shown in Figure 4a, and the basic write cycle, in Figure 4b. For each of the four memory quadrants, there is a choice of zero, one, two, or four automatic wait states.

Separate strobes are provided for I/O transactions and up to eight I/O chip-select signals can be output on Port E. The 64-KB external I/O address space is evenly divided for the eight I/O chip selects. Each I/O region can be programmed for 1, 3, 7, or 15 automatic wait states and can be write-protected.

Internal I/O addresses are separate from external I/O addresses. Of course, most of the time a Rabbit system won't need external peripherals because most of the digital peripherals an embedded system might need are on-chip.

Table 1a—These Z80 and Z180 instructions were removed to make room for new instructions. They can all still be emulated in software. **b**—These new Rabbit-80 instructions provide higher performance. The main additions are the 16-bit math operations and long address operations.

SLAVE PORT

One unique feature of the Rabbit is the slave port. This byte-wide port is a bidirectional communications channel between the code on the Rabbit and a master system. To the master, the Rabbit looks like a combination of

three write registers, three read registers, and a status register. The Rabbit CPU has the same view of the slave port.

Figure 5 shows the slave-port hardware interface. Since the chip select is designed into the slave-port interface, a number of Rabbit slaves can be simultaneously used in a system.

The master writing to the slave port can generate a Rabbit interrupt, and a Rabbit writing to the slave port can activate the interrupt signal to the master. This setup permits high-speed interrupt-driven transfers.

A primary use of the slave port is to offload the intelligence required to drive peripheral devices onto the Rabbit system. Thus a small Rabbit system would look like a group of "smart" peripheral devices to a master system.

SERIAL PORTS

The Rabbit has four independent serial ports: two are async-only, and two are combination async/clocked serial.

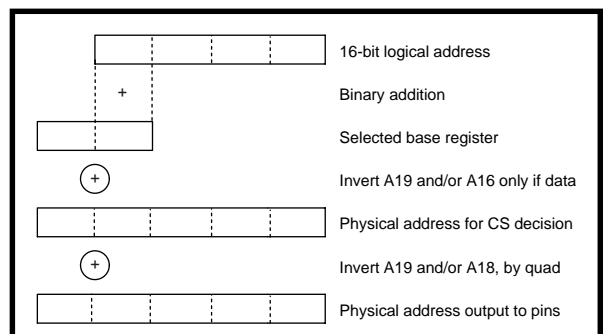


Figure 2—The MMU translates logical addresses to physical addresses. It also provides for instruction/data separation and bank switching.

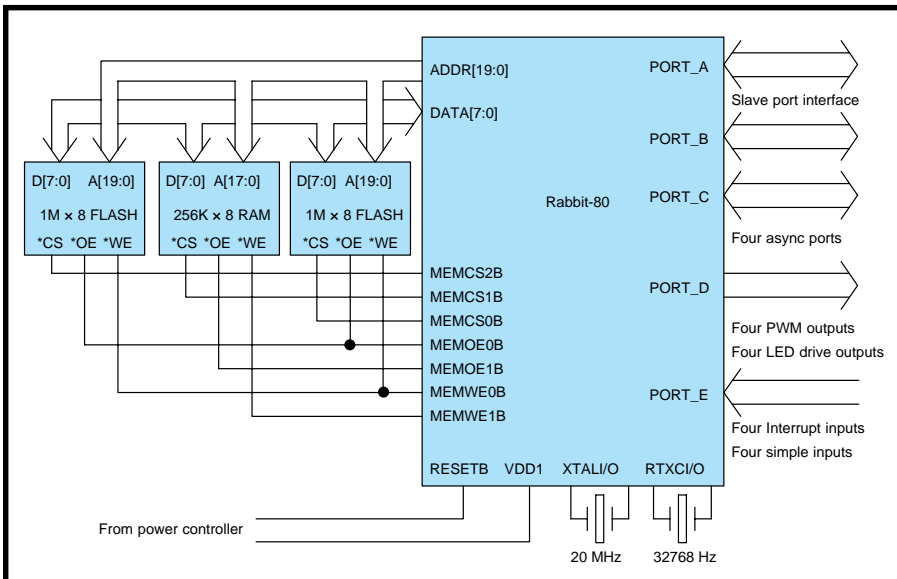


Figure 3—In a typical Rabbit system, only external memory chips are required and they connect without any glue logic.

Parallel Port C is the primary serial-port interface, but the combination serial ports can also be connected through parallel Port D. This multiplexing allows the combination ports to be used in both modes, with separate line drivers, although not both at the same time.

The async capabilities don't include all of the seldom-used options often found in serial ports, but everything you need for 99% of the possible applications is there. The ports require a 16x clock and automatically include start and stop bits.

They can also be programmed to send and receive an extra address bit for multidrop applications. There's a single byte of buffering for both the receiver and the transmitter.

The clocked-serial capabilities allow communication with SPI and I²C devices using an internal or external clock. External A/D and D/A devices often communicate using these types

of serial transfers. Although it would require software to do the frame formatting, these ports could be used to send and receive HDLC messages.

TIMERS

The first set of timing resources on the Rabbit, Timer A, contains five separate 8-bit counters. Four of these are used to supply the clocks for the serial ports. The fifth can be used to time outputs on the parallel ports.

The counters can be cascaded if slow transfer rates are necessary. Of course, a programmable interrupt is available for the terminal-count condition on all five counters.

The second set of timing resources, called Timer B, is significantly more powerful than Timer A because it was designed to facilitate PWM outputs on the parallel ports. Timer B consists of a 10-bit counter, a pair of 10-bit match registers, and a pair of 10-bit buffer registers. Timer B is shown in Figure 6.

The counter is free-running and is clocked by the system clock divided by two or eight, or by the output of one of the counters in Timer A. Each match register generates an output pulse when the counter matches its contents. This pulse can be used by two parallel ports to precisely time output transitions.

Whenever a match occurs, the corresponding match register is loaded from the buffer register and an interrupt is generated so the buffer can be reloaded, which allows PWM to be accomplished with little CPU overhead. Using the two match registers in tandem makes it easy to create the quadrature signals necessary to drive a servo.

PARALLEL PORTS

The five parallel ports on the Rabbit can be used individually or programmed to provide inputs and outputs for the other on-chip peripheral devices.

Port A is a byte-wide port that can be used for simple byte-wide I/O or as the data bus of the slave port. Port B consists of two outputs, four inputs, and two I/O. Port B is mostly used for the control signals for the slave port, in addition to the two clocks for the clocked serial ports. No special timing options are available in output mode for these two ports.

Port C consists of four inputs and four outputs, and serves as the primary I/O for the four serial ports. Port C inputs are always reported in the data register (even when being used as serial port inputs) to allow the timing of a received break signal. Port C outputs can be individually selected as simple outputs or serial port outputs.

Four bits of Port D provide alternate I/O for the two dual-function serial ports, but all eight bits of Port D are

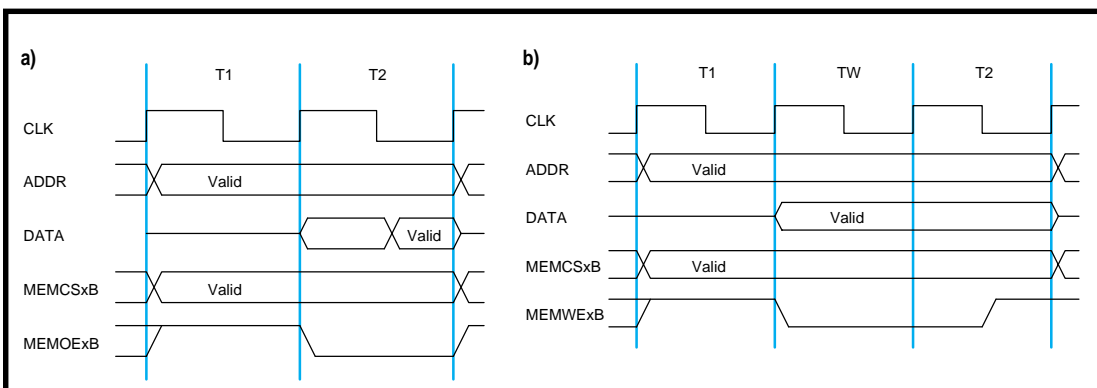


Figure 4a—Memory read timing fits perfectly with SRAMs or flash memory. Access time is two full clock cycles. b—Memory write timing looks a little different. The extra clock cycle allows for a wider write pulse and some data hold time.

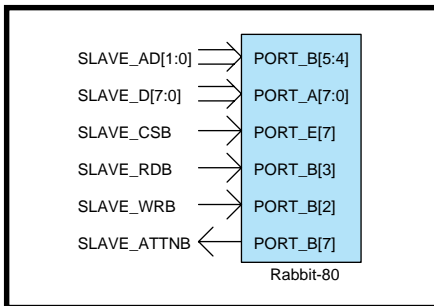


Figure 5—The Rabbit looks like an I/O device when using the slave port. This feature allows the Rabbit to be used as a smart peripheral device.

individually programmable for data direction. Each bit can be programmed to be open-drain as an output, and Port D has higher drive capability than the other ports for driving LEDs, optoisolators, solid-state relays, and more.

As outputs, the two nibbles of Port D can be clocked independently by Timer A's output or by either Timer B output. Any data written to the port's data register isn't clocked to the pin until the selected signal is active, allowing precise timing of the Port D outputs.

Port E has capabilities similar to Port D's but lacks the high drive out-

puts or open-drain capability. Port E can be programmed to provide the I/O chip selects and the chip-select input for the slave port. Four Port E pins can also be used as CPU interrupt inputs.

INTERRUPTS

The Z80 interrupt mechanism has been completely revised in the Rabbit. In the Rabbit interrupt scheme, there are four interrupt priority levels.

The CPU's current priority level determines whether or not an interrupt is accepted, thereby allowing only interrupt requests with higher priority to be accepted. As well, the CPU keeps a stack of the four most recent priority levels, which allows preemption.

Every interrupt source can be programmed to request an interrupt at one of the four levels. Of course, an interrupt requested at level 0 would never be accepted because it isn't higher than anything. This is how individual interrupts are disabled. Similarly, when the CPU is at level 3, no interrupts will be accepted, because there isn't a higher priority.

A typical priority assignment might be to have the serial ports and timers request at level 1, external interrupts at level 2, and one external interrupt at level 3, simulating a nonmaskable interrupt. Then the normal CPU priority would be level 0, enabling all interrupts. Of course, if there are critical sections of code that can't be interrupted, then they should be run while the priority is temporarily set to level 3.

All accepted interrupts cause an automatic branch to a fixed location in memory, as shown in Table 2. Inter-

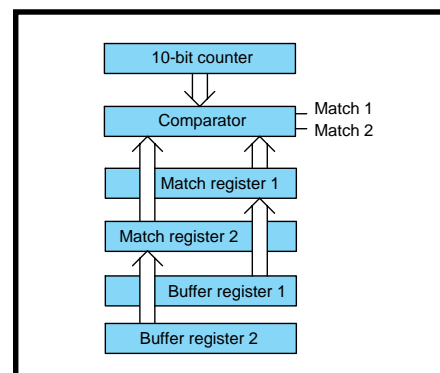


Figure 6—Timer B takes care of PWM tasks easily. The double buffering reduces time-critical overhead.

rupts from internal peripherals use the R register to set the upper eight bits of the address of the service routine and the external interrupts use the I register to set the upper eight bits of the address, which is similar to how these registers are used in the Z80.

The entry points are separated by 16 bytes in memory. Some interrupts can be serviced in this amount of code or enough housekeeping can be done to share the bulk of a service routine. For example, the four serial ports can use identical code, as long as the port addresses and data-table pointers are set up in these 16 bytes before branching to the common routine.

The Rabbit has one other interrupt—the periodic interrupt, which occurs at a rate of 2.048 kHz. This interrupt is useful for time-slicing or low-speed status monitoring and is generated from the 32-kHz oscillator, which also clocks the watchdog timer and real-time clock.

WDT AND RTC

The watchdog timer (WDT) can be programmed to time out after 250 ms, 500 ms, 1 s, or 2 s. Its control register recognizes four specific bytes to set the next timeout period. Any other bytes written to this register are ignored, and if one of these four bytes isn't written in time, the Rabbit is reset.

The RTC doesn't keep hours/minutes/seconds time; it's a 48-bit counter clocked at 32 kHz. It can be set to a specific count by software before it starts counting up. If the software starts counting from zero, that's 272 years of timekeeping (yes, it's Y2K compliant).

The Rabbit contains a completely separate power plane for the RTC, so

Interrupt source	ISR starting address
Periodic interrupt	{R, 00000000}
Slave port	{R, 10000000}
Timer A	{R, 10100000}
Timer B	{R, 10110000}
Serial port A	{R, 11000000}
Serial port B	{R, 11010000}
Serial port C	{R, 11100000}
Serial port D	{R, 11110000}
External interrupt 0	{I, 00000000}
External interrupt 1	{I, 00010000}

Table 2—Interrupt routines start on 16-byte boundaries, which provides space for some housekeeping before jumping into the main service routines.

Listing 1—Just 12 bytes are needed for the bootstrap loader. The hardware automatically modifies the bytes to select the bootstrap source and to select an internal I/O destination.

```

ORG 0h
LD L, n      ;n = 0C0h for serial, n = 020h for parallel
IOIP
LD D, (HL)   ;fetch the address msb
IOIP
LD E, (HL)   ;fetch the address lsb
IOIP
LD A, (HL)   ;fetch the data
IOIP or NOP  ;IOIP if D(7) = 1, NOP if D(7) = 0
LD (DE), A   ;store the data
JR 0h        ;loop back

```

a battery-backed system can keep time when the main power fails. Both the 32-kHz oscillator and the RTC are optimized for power consumption. If the main power is absent, a 150-mAh battery powers the RTC, so the Rabbit can keep time for about a year.

CLOCKING AND POWER OPTIONS

The 32-kHz oscillator is accompanied by a separate high-speed oscillator for the remainder of the chip, but a number of clocking options are available. After reset, everything starts out running in divide-by-8 mode from the high-speed oscillator, but you can also select full speed or 2× full speed.

The 2× option saves some power in the high-speed oscillator. The Rabbit can be programmed to use the 32-kHz clock and the high-speed oscillator can be turned off during inactive periods. Or, you can power down everything but the RTC for the ultimate power savings.

BOOTSTRAP

A primary design goal was to allow for remote bootstrap (i.e., to make it possible to have RAM-only systems using the Rabbit). In this mode, the CPU fetches instructions from a small (12 byte) internal ROM containing the code in Listing 1. At each internal I/O read instruction, the internal logic holds the CPU in a wait state until a byte is available from the selected peripheral.

Two pins control bootstrapping by selecting normal operation, boot from the slave port, or boot from serial Port A in async or clocked serial mode. The Rabbit samples the pins after reset (when the PC address is 0000 in the four least significant bits) to enable the boot ROM.

The bootstrap mode assumes data transfers in groups of three bytes. The first two bytes are the address for the data, followed by the data itself. This arrangement allows data to be loaded anywhere in memory.

To select between external memory and internal peripheral control registers, the Rabbit uses the most significant bit of the downloaded address. Once the peripherals are accessible, the MMU can be programmed, providing full memory access.

To exit bootstrap mode, write to a bit in an internal control register or change the pins back to normal operation. It's easy to exit this mode, because the I/O location that disables it is accessible during bootstrap.

ROLLING ALONG

Now that you know how the Rabbit was designed and what it contains, what can you do with it? Anything a Z80 or Z180 can do, and more. Plus all the development software is available. ☐

Thanks to everyone at Z-World, especially Norm Rogers for conceiving the Rabbit-80, and Pedram Abolghasem for running the project (and doing much of the work) there.

Monte J. Dalrymple has been designing ICs for more than 20 years. He currently has his own company, which develops intellectual property. You may reach him at monted@systemyde.com.

SOURCE

Rabbit-80
Rabbit Semiconductors
(530) 757-8400
www.rabbitmicro.com

Get SmartMedia

Part 2: Hands On

FROM THE BENCH

Jeff Bachiochi

INSERT HERE

Many SmartMedia sockets have an internal microswitch which enables you to determine if a device has been inserted into the socket by the physical operation of the switch's contacts. This mechanical insertion verification can also be achieved through the SmartMedia contacts.

The interface has a pull-up resistor tied to pin 11 of the SmartMedia socket. When the device is not fully inserted into the socket, the pullup registers a logic high with the interface. When fully inserted, pin 11, which is physically connected to the V_{SS} (pins 1 and 10) contacts of the SmartMedia device, grounds the pullup and registers as a logic low to the interface.

PARAMETER DETERMINATION

Once insertion has been resolved, the proper operation voltage must be determined because, certainly, applying 5 V to a 3.3-V device is just asking for trouble. Again, mechanical microswitches can be used to determine operating voltage (at the same time as insertion).

Besides the fact that the operating voltage is written on the SmartMedia, the packaging has either a left-clipped corner, indicating 5-V operation, or a right-clipped corner, indicating 3.3-V operation. So, two switches strategically placed at either corner of the socket could be used to mechanically identify the proper operating voltage. The SmartMedia's clipped corner would not engage one of the two switches, thus indicating which media was inserted.

Determining the proper voltage can be done electrically as well, via pin 18. This time a pulldown is used. On 5-V devices, pin 18 floats (internally unconnected), whereas on a 3.3-V device, pin 18 is connected to V_{CC} .

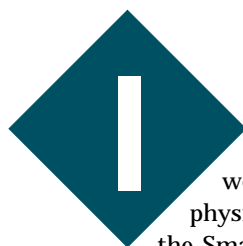
So, once inserted, a 5-V SmartMedia device will not affect the logic low read by the interface, indicating the need for a 5-V supply. On the other hand, a 3.3-V device with pin 18 connected to V_{CC} raises the interface to a logic high, which indicates the need for a 3.3-V supply.

Personally, I think a pullup makes more sense here. This way, the interface would see the need for a 3.3-V

Your latest assignment: Follow



Jeff as he goes undercover (under the embedded-device cover, that is) to investigate how to access the SmartMedia nonvolatile flash memory.



Last month, I went over the physical makeup of the SmartMedia flash-memory device. I'd like to start out this article by introducing some circuitry I designed that enables you to access the SmartMedia.

As you can see from Figure 1, there's not much to the interface. A microcontroller handles both the SmartMedia interfacing as well as a serial communications channel.

One of the important points of this circuit is that it can run at either 3.3 or 5 V. The SmartMedia device inserted into the SmartMedia socket indicates the necessary voltage through the output pin 17. (If you missed last month's column, now might be a good time to go back and read it because I'm going to assume that you're familiar with the signals of the SmartMedia.)

SmartMedia products cover such a large range, you must be able to sense the type and size of the device inserted into your equipment. If you don't recognize the SmartMedia, you should make accommodations to reject or at least not corrupt the data held in the device.

Therefore, it's recommended that you follow certain steps to determine whether the device is compatible with your equipment: the detection of insertion, operating voltage, type and size, physical format, and logical format.

supply when no device is inserted and it would prevent the interface from thinking it should supply 5 V, which would be a bad thing during a 3.3-V device insertion. Anyway, using pin 18 enables the device to be probed at the lower voltage and raised only if necessary (and safe).

SHOW SOME ID

At this point, the proper operational-voltage parameter can be set, the media type and size identified, and we can begin to communicate with the SmartMedia device. A ReadID command regurgitates both a maker ID and device code. The manufacturer's ID may be 98H (Toshiba), ECH (Samsung), or another code registered with the SSFDC.

You'd think that with so few original players creating the standard, device type and size codes would be standardized throughout manufacturers. Although this is true for many codes, some device types and sizes have different codes from each manufacturer for the same part description.

The device code indicates parameters like device size, operating voltage, and memory type. If you find ID and device codes that you don't recognize, it only makes sense to reject the device. A further operation on an unknown device runs the risk of damaging the data held within the device.

PHYSICALLY FIT

SmartMedia manufacturers pretest every page of flash memory for stuck (bad) bits. If a bad bit is detected in any page within the block, every page in that block is marked as bad.

Remember that this indicator byte is flash memory and when the block is erased, so is the bad block byte. So, when erasing a block, the user must remark it as bad either from a user-compiled list of bad blocks or by re-testing the block for stuck bits.

Size	1 MB	2 MB	4 MB	8 MB	16 MB	32 MB	64 MB	128 MB
Cylinders	125	125	250	250	500	500	500	500
Heads/Track	4	4	4	4	4	8	8	16
Sectors/Head	4	8	8	16	16	16	32	32
Total sectors	2000	4000	8000	16,000	32,000	64,000	128,000	256,000
Sector size	512	512	512	512	512	512	512	512

Table 1—CHS parameters are predefined for present and future SmartMedia devices.

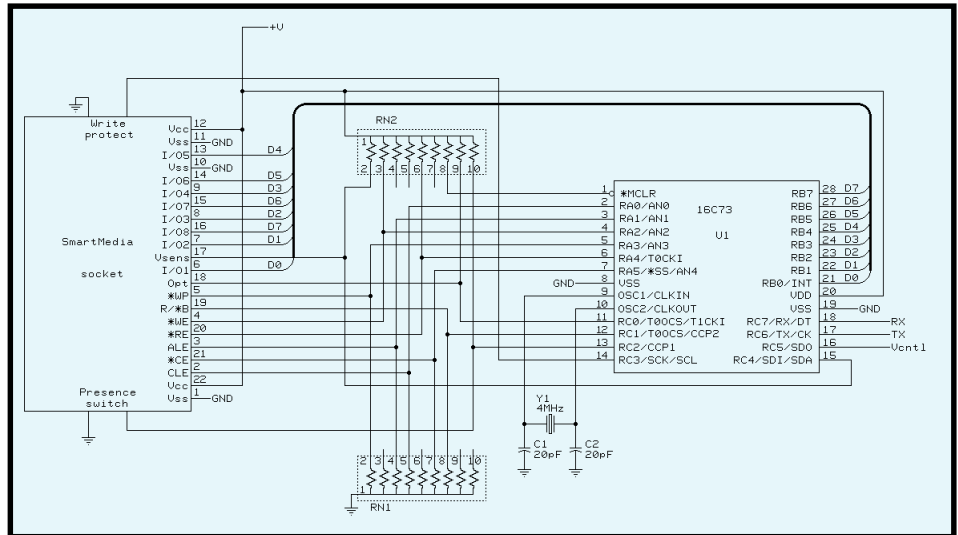


Figure 1—A PIC processor can be used to create a simple interface to a SmartMedia device socket. A 5-/3.3-V supply is not shown. However, the PIC can operate at either voltage. The VCNT1 connection is used to control the appropriate operating voltage, depending on the device inserted into the SmartMedia socket.

In addition to testing and marking bad blocks, the manufacturer does some additional housekeeping to comply with the physical format specifications. The first physical page (or first good physical page) must have card information structure and/or identify drive information (CIS/IDI) data written to it.

You should note that, although early SmartMedia devices used a 256-byte page (with an 8-byte redundant area), the larger devices made today use a 512-byte page (with a 16-byte redundant area). For the remainder of this discussion, however, I'll only talk about the 528-byte (512 + 16) page.

The CIS field is the first 128 bytes. It identifies the device and determines if the device has been physically formatted. The IDI field is the second 128-byte data area, which can be used by the ATA-interface-equipped host system.

As a precautionary measure, the CIS/IDI data is repeated in the second half of the page (see Figure 2). The CIS/IDI format is defined by PCMCIA.

The PCMCIA slots on most laptops interface to many peripherals, includ-

ing PCMCIA memory cards. When a SmartMedia card is inserted into a PCMCIA ATA adapter card, the PCMCIA slot looks for this information as if it were a PCMCIA memory card. Voilà! Instant device recognition based on a previously designed standard.

The 16-byte redundant area holds information pertinent to the validity of the previous 512 bytes of the page (see Figure 2). This area is broken down into seven pieces of information. The first 4 bytes are reserved for extension information.

The next byte is the page data validity marker. If the data written is not correct, a 0 indicates that the user should not consider this page to be good and should proceed to the next page for CIS/IDI information.

The next byte is fixed at FF, which is normally the bad block marker. The next two bytes are fixed at 00 and 00 (the physical block number). Following those two bytes are three bytes of error correction code (ECC) data.

The ECC's 22 bits are calculated from the first 256 bytes of the page. Finally, the physical block number, in this case a fixed 0000, is repeated followed by an ECC on the second 256 bytes of the page.

So, not only is the CIS/IDI data repeated twice within this page, but separate ECCs are calculated and saved in the page's redundant area. The ECC values enable a

```

1 - Sequential Data In
2 - Read 1
3 - Read 2
4 - Read ID
5 - Reset
6 - Page Program
7 - Block Erase
8 - Read Status
0 - Quit

3
Enter the page # to read
A page is 512 bytes of data + 16 spare byte

32
Reading page #32
000 01 03 D9 01 FF 18 02 DF 01 20 04 00 00 00 00 21
010 02 04 01 22 02 01 01 22 03 02 04 07 1A 05 01 03
020 00 02 0F 18 08 C0 C0 A1 01 55 08 00 20 18 0A C1
030 41 99 01 55 64 F0 FF FF 20 1B 0C 82 41 18 EA 61
040 F0 01 07 F6 03 01 EE 1B 0C 83 41 18 EA 61 70 01
050 07 76 03 01 EE 15 14 05 00 20 20 20 20 20 20 20
060 00 20 20 20 20 00 30 2E 30 00 FF 14 00 FF 00 00
070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

100 01 03 D9 01 FF 18 02 DF 01 20 04 00 00 00 00 21
110 02 04 01 22 02 01 01 22 03 02 04 07 1A 05 01 03
120 00 02 0F 18 08 C0 C0 A1 01 55 08 00 20 18 0A C1
130 41 99 01 55 64 F0 FF FF 20 1B 0C 82 41 18 EA 61
140 F0 01 07 F6 03 01 EE 1B 0C 83 41 18 EA 61 70 01
150 07 76 03 01 EE 15 14 05 00 20 20 20 20 20 20
160 00 20 20 20 20 00 30 2E 30 00 FF 14 00 FF 00 00
170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

200 FF FF FF FF FF FF 00 00 0C CC C3 00 00 0C CC C3

```

Figure 2—With a serial terminal, the SmartMedia interface can be used to investigate SmartMedia devices. Here's the boot record of a device.

single bit error in the 256-byte page area to be repaired. Multiple bit errors can be identified but not repaired.

Note here that, except for the CIS/IDI block, the physical block number stored in the redundant area of each page is in the range 0–999 (actually $2 \times (0-999) + 1000h + \text{even parity}$).

The blocks are kept in groups (or zones) of 1024. Although larger devices may require the use of several zones, smaller devices may call for a partial zone.

Within a zone, not all available blocks are designated for use. On a device with 1024 physical blocks, only 1000 logical blocks are used. The remaining blocks are spares (or in the case of blocks with bad bits, unusable). But, as you will see, the good spare blocks do get rotated into use within the zone.

HOW ECC WORKS

The ECC is a string of odd parity bits representing the rows and columns of the 256-byte data field. For the 8-bit (2^3) row parity, six bits (2×3) of ECC data are required. For the 256-byte (2^8) column parity, 16 bits (2×8) of ECC data are required.

Each ECC row bit holds odd parity data for four different column bits of all 256 bytes. Each ECC column bit holds odd parity data for eight different row bytes of all eight bits.

The ECC data is written into the page's redundant area along with the page data. When the page is read from the device, the ECC data can be compared to newly calculated ECC data. A good compare will ensure good data integrity.

What happens when a bad bit occurs somewhere? If the bad bit is in the data area, every ECC data bit that uses the bad bit in its calculation will be affected and will show up in one of each pair of ECC bits. From this pattern you can determine which data bit is at fault and correct it.

If two data bits are bad, the ECC data will reflect an error in both bits of an ECC pair. This process identifies an error, but the bits involved cannot be pinpointed and so the errors can't be fixed.

If a bit in the ECC data is bad, it shows up as a single ECC bit error. This kind of error cannot be caused by any combination of bad data-area bits. So, it can be ignored and filed in the proverbial bit bucket.

The 22 bits of ECC data is in the following format: most significant byte of the 16 line-parity bits, followed by the least significant

byte of the line parity, and finally the six column-parity bits shifted left twice with the least significant bits filled with 1s.

Calculating the ECC data can be done ahead of time where the parity equivalent of each data value (0–255) fills up a look-up table, or it can be calculated on a byte-by-byte basis. Figure 3 shows which bits of each byte value are XOR'd to create the line parity (LP) and column parity (CP) data.

ECC calculations can be handled by hardware or software. The program listed in the Software section is a BASIC program that calculates the 22-bit ECC for a half page of data (256 bytes).

This program runs on a PC under QBASIC and will respond with the LPHigh, LPLow, and CP bytes for the array D(1)–D(256). You can plug values into the array to see how changes in the array affect the parity generated.

THAT'S LOGICAL

To make using SmartMedia simpler, there must be even more complexity. Huh? Yeah, that's right—just like most "user friendly" devices, it often takes many layers of complexity to simplify its use.

Although we could stop here and use the SmartMedia in an application, the data in the flash-memory device would be unreadable elsewhere (unless you wrote drivers for every piece of equipment that needed to access the

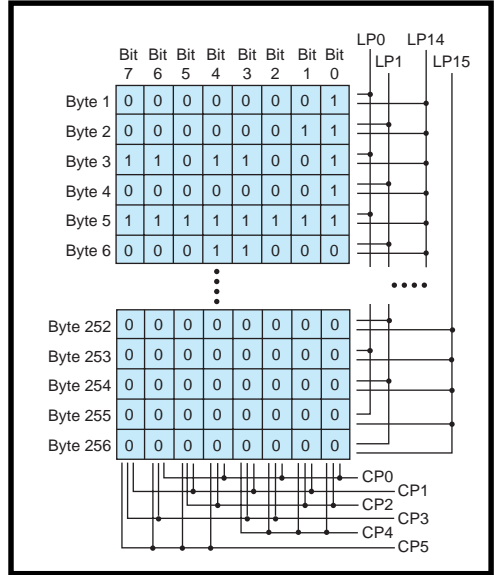


Figure 3—Line parity and column parity data bits can indicate and correct for single bit errors. This figure shows how each half page of data is used to create the 22 bits of ECC (error correction code) data.

data). Formatting the physical format with a DOS file logical format makes SmartMedia compatible with other equipment that's set up to process the DOS file format. We use floppies, hard drives, and CDs everyday—all of which use the DOS FAT file structure.

Three items are necessary for the proper format. First, there must be a definition of CHS parameters. Take a look at Table 1 to see how the cylinders, heads, and sectors are defined based on the media's size.

The CHS parameters are based on the logical blocks set up in the physical formatting. With a sector size of 512 bytes (a page), and 32 sectors per head (a block), calculations are easy.

The first logical block consists of sectors 0-1Fh of cylinder 0, head 0. The second logical block consists of sectors 20h-3Fh of cylinder 0 head 1, and so on.

Second, the media must have a master boot sector located on the first good sector of the device. If you remember back to the physical format, the CIS/IDI also resides there. There is no conflict because the first partition info starts at offset 1BEh with the boot ID 80h. Table 2 lists the rest of the partition parameters.

Offset 1C6h-1C9h indicates where the partition boot sector is located (in this case, logical sector 37h). Given one page per sector, the PBS is on cylinder 0, head 1, and sector 17h.

The final piece is the partition boot sector. Again, there are a number of parameters here, as well as the FAT and the directory. Table 3 gives you all the pertinent information.

The FAT follows the partition boot sector, which indicates how many sectors are required for the FAT and that there are two copies (redundancy = security). So, the offset to the second FAT and to the first directory entry can be calculated.

A directory entry is 32 bytes long and contains the file name, extension,

Table 2—Once the device is formatted, the boot sector provides information about the device, including where to find the first partition.

Offset	Parameter	Data (64-MB device)
000h-1BDh	Boot command (not used)	00h-00h
Partition 1		
1BEh	Boot ID	80h
1BFh	Start head	01h
1C0h	Start sector	18h
1C1h	Start cylinder	00h
1C2h	System ID	01h
1C3h	End head	07h
1C4h	End sector	60h
1C5h	End cylinder	F3h
1C6h-1C9h	Start logical sector	00 00 00 37h
1CAh-1CDh	Partition size	00 01 F3 C9h
1CEh-1DDh	Partition 2 (not used)	00h-00h
1DEh-1Edh	Partition 3 (not used)	00h-00h
1EEh-1FDh	Partition 4 (not used)	00h-00h
1FEh-1FFh	Signature	AA 55h

and attribute, along with the last edit time/date, the starting cluster of the file, and its length. Instead of speaking in terms of sectors, the term "cluster" is used. A cluster is defined in the partition boot sector and is the minimum number of sectors that can be allocated to a file.

From the starting cluster number, the starting logical sector of the file's data can be determined. If the file's length exceeds the cluster size, additional clusters can be allocated through the FAT.

SECTOR MANIPULATION

Remember when I mentioned how not all of the physical blocks were given logical addresses, and that some extra blocks were left unallocated? Well, the first time the block is written with

data, the data merely gets programmed into the erased logical blocks.

When a block needs to be changed, the block must be read from the device (including the redundant area), the pertinent data changed, and the block written back to any unused block in the same zone. The redundant area is also written, identifying it as the same logical block that was just read. Now, the original block is erased, removing it from service. If you are keeping a table of which physical block is which logical block, you must correct the table.

I've purposely not mentioned this

logical-to-physical table until now because it requires 1000 entries and not all equipment has enough RAM to keep a table of this size. It speeds things up, though, since you don't have to search the whole device, physical sector by physical sector, looking for the logical sector you require. As long as you build the table and keep it current, you won't have to continually search the device.

PROBING SMARTMEDIA

The PIC I used in this project ('16C63) has less than 256 bytes of RAM, so there's no way I can attempt to keep track of physical and logical mapping. Nor can I buffer 528 bytes of

Offset	Parameter	Data (64-MB device)
000h-002h	Jump command	E9 00 00h
000h-00Ah	Manufacturer's name and version (ASCII, 8 bytes)	
00Bh-00Ch	Bytes/sector	02 00h
00Dh	Sectors/cluster	20h
00Eh-00Fh	Reserved sectors	00 01h
010h	Duplicate FATs	02h
011h-012h	Root directories	01 00h
013h-014h	Total partition sectors	00 00h
015h	ID byte	F8h
016h-017h	FAT sectors	00 0Ch
018h-019h	Sectors/track	00 20h
01Ah-01Bh	Heads	00 08h
01Ch-01Fh	Hidden sectors	00 00 00 37h
020h-023h	Total partition sectors (32 bit)	00 01 F3 C9h
024h	Physical drive	00h
025h	Reserved	00h
026h	Extended boot record signatures	00h
027h-02Ah	Volume ID (4 bytes)	00 00 00 00h
02Bh-035h	Volume label (ASCII, 11 bytes)	00h-00h
036h-03Dh	File system type (ASCII, 8 bytes)	FAT12
03Eh-1FDh	Reserved (IPL code area)	00h-00h
1FEh-1FFh	Signature	AA 55h

Table 3—The partition boot sector has more information, including the location of the directory and FAT.

```

000 49 4D 30 31 54 4F 53 48 20 20 20 10 00 00 00 00
010 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00
020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

Figure 3—After I took a single picture with my digital camera, the directory holds a single entry. The file name is IM01TOSH.

data for a single page and its redundant area.

What I can do is enable the user to use the basic operations available with SmartMedia by dumping data read from the device to a serial port. By using a COMM program like PROCOMM (or even HyperTerminal) the user can fully investigate SmartMedia.

But, you'll be required to prepare page data manually, including the redundant area (ECC calculation, block address, etc.). Figure 3 illustrates a dump of the directory area of

one of my digital camera's SmartMedia cards.

Using a SmartMedia card takes some work to be able to fully handle all the necessary formatting. But, only a few routines are necessary if all you wish to do is save some data. You can get away with a minimum amount of RAM, but there is a tradeoff of speed if you need to do any searching.

The only other problem I see is if you are collecting data and a page (sector) you have just written comes up with an error. Correcting the data (reading, correcting, and rewriting) is impossible without enough RAM to hold the page. In this case, you might just keep going. If it's only a one-bit error, it can be corrected using the ECC when the data is read back out of the device.

If you're designing equipment with a removable storage media requirement, consider using SmartMedia. Its physical size versus storage capacity is quite remarkable. ☐

Thanks to Doug Wong, Toshiba, for his SmartMedia expertise and documentation, and to Jeff Schmoyer, microEngineering Labs, for the PIC BASIC Pro.

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.

SOFTWARE

The BASIC program used to calculate the 22-bit ECC is available via the *Circuit Cellar* web site.

SOURCES

- SmartMedia**
Toshiba
(408) 737-9844
Fax: (408) 737-9905
www.toshiba.com
- Samsung Semiconductor, Inc.**
(408) 544-4000
Fax: (408) 544-4907
www.usa.samsungsemi.com

SILICON UPDATE

Tom Cantrell

'Net-in-a-Chip



There are plenty of options when it comes to

the hardware and software for getting an embedded gadget on the Internet. This month, Tom considers using the S-7600A as an entrance ramp to the I-way.



When it comes to getting embedded gadgets onto the I-way, designers have quite a few options to choose from, with more emerging all the time. The challenge is picking one that best matches your apps requirements.

If circumstances allow, why not rely on a standard off-the-shelf PC as your 'Net gateway? Thanks to the march of silicon (and some might say, the death-wish pricing tendencies of PC suppliers), you can get a perfectly adequate PC for well under \$1000.

And besides, even the cheapest PC has plenty of horsepower, comes with

the software to accomplish 'Net transactions, and has an unparalleled quiver of development tools.

More power to you if the PC approach works for you, but desktop PCs carry a lot of baggage. They're big, consume a bunch of power (consider going with a laptop), take a long time to boot, and neither the hardware nor software is especially robust.

Embedded PCs, whether Compact-PCI, PC/104, or a stand-alone SBC, can easily address the packaging and power concerns raised by their desktop brethren. Another option is to use Linux instead of Windows, exactly the tack taken by Stanford University's Wearable Computing Lab.

They built a tiny, Linux-based web server using a miniscule EPC [1]. Refer to the lab's web page or back to Ingo's articles on embedding Linux (*Circuit Cellar* 100-104) if you're interested in this approach.

Note that I didn't include price in the list of concerns. The fact is, EPCs cost hundreds of dollars (the EMJ board used at Stanford lists for over \$400) and at best are just competitive with desktop pricing. It would be easy to spend more putting together an EPC than buying a clone off the shelf.

A more streamlined solution that offers PC-class performance without extra baggage is a 32-bit CPU ('x86 or RISC) running an RTOS or Windows CE, fully loaded with protocol stacks, Java, and all the rest. You still need a hefty amount of silicon, and cobbling

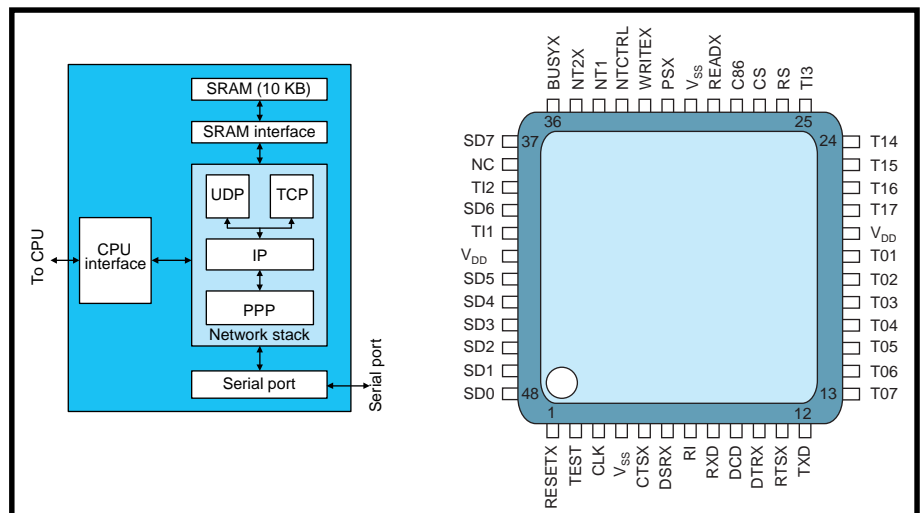


Figure 1—The Seiko '7600A is a standard chip based on iReady i1000 protocol processing hardware that handles all the details of TCP, UDP, IP, and PPP.

together all the pieces isn't trivial, but packaging, power consumption, and price challenges are easier to overcome without the Wintel baggage.

One of the best of this breed is the NET+ARM from NETsilicon. It combines a high-integration ARM CPU with networking hardware (Ethernet, UART, etc.) under the supervision of an RTOS that includes all the requisite networking software. NET+ARM pricing ranges from \$10 to \$40 depending on version and volume, and includes a glueless interface to (EP)ROM, SRAM, DRAM, SDRAM, and so on, making for a quick and easy minimal chip count design.

As an interesting aside, NETsilicon tipped me off to the formation of the Industrial Automation Open Networking Alliance by several embedded Internet players, including big guns like GE Fanuc, Siemens, Cutler-Hammer, Andover Controls, and Sun [2].

It'll be interesting to see how the IEEE 1451 standard (*Circuit Cellar* 103) and its prime proponent HP fit in. There's always been an interest in bridging the gap between office and factory automation. Now, the 'Net provides a way.

'NET UART

Whichever way you cut it, embedded 'Net solutions call for a 32-bit CPU and a fair amount of memory. Cutting more fat can get interesting.

Consider Myron Lowen's approach in "Internet Appliance Interface" (*Circuit Cellar* 108). He got on the

'Net with little more than a PIC and 2400-bps modem by short-circuiting a bunch of protocol bloatware. Risky? Indeed, in terms of ISP and tool compatibility. Intriguing? Very!

Back in May ("Betting On Webware," *Circuit Cellar* 106), I described a nontraditional solution offered by iReady. The i1000 protocol engine offloads a host CPU by handling all the details of TCP/IP, HTTP, SMTP, POP, and so on. Because it works with any CPU, the i1000 is a potentially easy and inexpensive way to add network capability to existing designs.

I say "potentially" because, as I lamented then, the i1000 isn't actually offered as a chip but as a bunch of IP (i.e., Verilog) from which you are supposed to build an ASIC—a fire drill that's neither cheap nor easy. What's needed is an off-the-shelf IC to support innovative and specialized apps.

Now, iReady has made a deal with Seiko to create just such a device, the S-7600A, also known as the iChip (see Figure 1). Packed in a tiny 48-pin QFP and under \$10 in volume, the '7600A is a networking solution that won't bog down efficient embedded designs.

On one side is a generic processor interface. On the other is a conventional RS-232 port. In between is the networking know-how and silicon (protocol engine and SRAM) to handle two sockets worth of IP, TCP, UDP, and PPP.

BOUNTFUL BUS

Connecting with your favorite controller is easy, thanks to a versa-



Photo 1—Exploiting the '7600A, Mike Johnson fields a worthy contender for "world's smallest web server" bragging rights.

tile interface. At RESETX, the '7600A checks the state of PSX and C86 to determine whether to use a serial or parallel host interface and, if the latter, whether Intel or Motorola flavor. The Intel convention features separate read and write strobes, but Motorola uses a read/write status line and a data strobe. The appropriate functions are mapped to READX and WRITEX.

SD7-SD0 make up the parallel data bus. In serial mode, SD7-SD5 become the serial data in, clock, and out, respectively. Chip select (CS) enables access when asserted and tristating outputs when deasserted so the '7600A can share the bus with other peripheral ICs.

The '7600A uses the BUSYX output to handshake with the host. The active-high and active-low interrupt request outputs (INT1 and INT2X) are configured as totem-pole or open-drain by setting INTCTRL high or low.

Inside the '7600A, you'll find dozens of registers that control various aspects of the chip's operation, as you see in Table 1. There aren't enough address lines for direct access, so getting at a specific register is a multistep process (see Figure 2).

First, the host writes the desired address with the register select (RS) pin low. Then, it issues a read or write command with RS high to initiate the register access. The

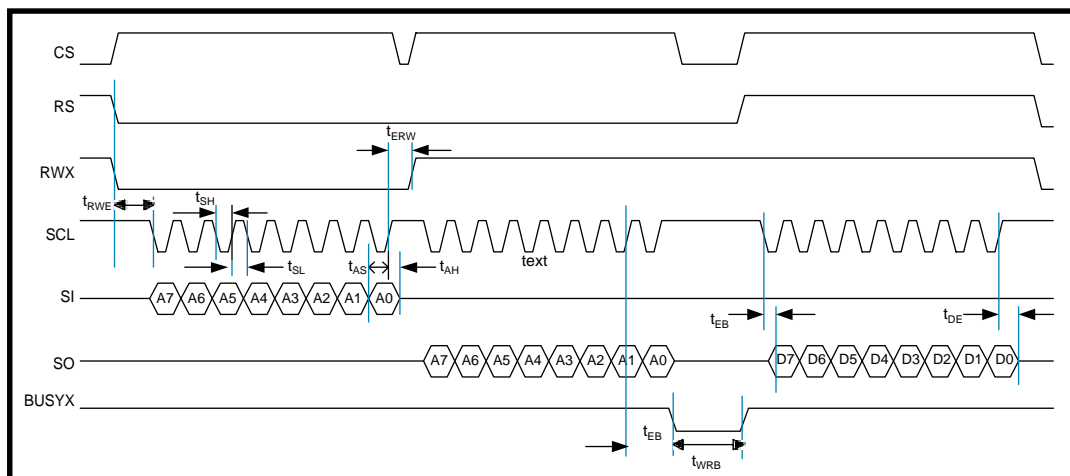


Figure 2—Whether you're using the parallel or serial (shown here) interface mode, reading data from the '7600A is a three-step process—write the desired register address, start the read operation and, after BUSYX returns high, read the data.

'7600A asserts BUSYX while it performs the data transfer. For a read, the host retrieves the data with a third access after BUSYX is deasserted.

The network connection is a simple UART. There's a 16-byte receive FIFO, optional RTS/CTS handshaking, and individual interrupt-enable bits for receive and transmit ready. Format is fixed at 8N1, and the data rate is derived as a programmable divide ratio of the chip clock.

Speaking of the clock (CLK), it can be quite leisurely because the '7600A implements protocol processing with hardware state machines. The minimum clock rate depends on the data rate demands of the physical channel, but it's only on the order 1 Hz per bps of physical channel bandwidth, so a mere 100 kHz could handle any modem. The low-speed clock and optimized hardware also make for low power consumption (i.e., milliamps in operation and microamps in standby).

PROTOCOL POWER

In Figure 2, after RESET, the host initializes the '7600A and does house-keeping via registers 0-1D. For example, it sets the serial port data rate divider (and a similar divider responsible for generating a 1-kHz timebase), enables interrupts, and so on.

Here, your software controls the serial port and is responsible for establishing communication. The likely scenario is to issue the ATD command to a connected modem and dial an ISP. Here's where it gets interesting.

After the modems establish communication, your software flips the serial control (SCTL) bit in the configuration register (0x08) and the onchip PPP logic (registers 0x60-0x6F) takes control of the serial port and starts yacking with the ISP's software. The '7600A handles the back-and-forth of options, passwords, and such that otherwise call for a lot of code.

Once PPP does its thing, including assigning Our_IP_Address (i.e., negotiated or floating assignment) if you didn't set it explicitly, you're on the 'Net and action shifts to registers 0x20-0x3F, which front the TCP/IP service. Note that this set of registers is duplicated for each of the two sock-

ets the '7600A supports. The value (0 or 1) set in the Index register (0x20) defines which is accessible.

In Internet-speak, a socket is a full-duplex port uniquely identified by the concatenation of IP address and a socket number. A pair of sockets define a connection (also unique), though a single socket can simultaneously participate in multiple connections.

Per TCP, at this point you'd need a bunch of software to open a socket and establish a connection as shown in Figure 3. Instead, the '7600A handles all the grunt work. All you have to do is set an activate socket bit and wait for a connection established bit.

Once connected (i.e., two sockets talking via TCP/IP), the '7600A handles all the details (e.g., flow control, and error recovery) and it's easy to transfer data, either polling or under interrupt control at your discretion.

All buffering is handled on-chip with 10 KB of SRAM. Each socket has a 1-KB send buffer and 2-KB receive buffer. The remaining 4 KB is allocated to protocol-specific (e.g., TCP, IP, PPP) buffers and data structures.

LAYER IT ON ME

As I explained in May, a fully loaded i1000 could incorporate hardware support for higher layer protocols such as

Add	Register	Bit definitions							
		Major revision number				Minor revision number			
0x00	Revision								
0x01	General_Control	-	-	-	-	-	-	-	SW_RST
0x02	General_Socket_Location	0	0	0	0	0	0	1	1
0x04	Master_Interrupt	-	-	-	-	-	PT_INT	LINK_INT	SOCK_INT
0x08	Serial_Port_Config	S_DAV / Loop back mode	DCD / Parallel mode	DSR / HWFC	CTS	RI	DTR	RTS	SCTL
0x09	Serial_Port_Int	PINT	DSINT	-	-	-	-	-	-
0x0A	Serial_Port_Int_Mask	PINT_EN	DSINT_EN	-	-	-	-	-	-
0x0B	Serial_Port_Data	Serial port data register							
0x0C-0x0D	BAUD_Rate_Div	Baud rate divider registers							
0x10-0x13	Our_IP_Address	Our IP address							
0x1C	Clock_Div_Low	Low byte for 1-kHz clock divider							
0x1D	Clock_Div_High	High byte for 1-kHz clock divider							
0x20	Index	Socket index							
0x21	TOS*	Type of service field							
0x22	Socket_Config_Status_Low*	TO	Buff_Empty	Buff_Full	Data_Avail/RST	-	Protocol_Type		
0x23	Socket_Status_Mid*	URG	RST	Term	ConU	TCP State			
0x24	Socket_Activate	-	-	-	-	-	-	S1	S0
0x26	Socket_Interrupt	-	-	-	-	-	-	I1	I0
0x28	Socket_Data_Available	-	-	-	-	-	-	DAV1	DAV0
0x2A	Socket_Interrupt_Mask_Low*	TO_EN	Buff_Emp_En	Buff_Full	Data_Avail_En	-	-	-	-
0x2B	Socket_Interrupt_Mask_High*	URG_En	RST_En	Term_En	ConU_En	-	-	-	-
0x2C	Socket_Interrupt_Low*	TO	Buff_Empty	Buff_Full	Data_Avail	-	-	-	-
0x2D	Socket_Interrupt_High*	URG	RST	Term	ConU	-	-	-	-
0x2E	Socket_Data*	Socket 8-bit data							
0x30	TCP_Data_Send (WO)*	Any write causes data to be sent							
0x30-0x31	Buffer_Out (RO)*	Buffer out length							
0x32-0x33	Buffer_In (RO)*	Buffer in length							
0x34-0x35	Urgent_Data_Pointer*	Urgent data offset pointer, UDP datagram size							
0x36-0x37	Their_Port*	Target port address							
0x38-0x39	Our_Port*	Our port address							
0x3A	Socket_Status_High*	-	-	-	-	-	-	-	Snd_Bsy
0x3C-0x3F	Their_IP_Address*	Target IP address							
0x60	PPP_Control_Status	PPP_Int	Con_Val	Use_PAP	TO_Dis	PPP_Int_En	Kick	PPP_En	PPP_Up / SR
0x61	PPP_Interrupt_Code	Interrupt code							
0x62	PPP_Max_Retry	-				PPP maximum retry			
0x64	PAP_String	Pap user name and password							
0x6F	PPP Test Control	-	-	-	-	Test	Bypass	-	Loop Back

Table 1—The host controls the '7600A operation and transfers data through various registers. A number of the registers in the range 0x21-0x3F (marked with an asterisk) are duplicated for each socket and selected with the Index register (0x20).

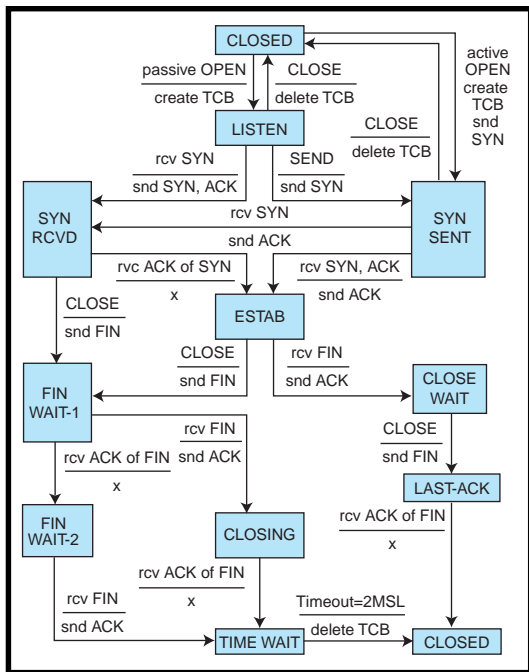


Figure 3—As this diagram, duplicated from the original 1981 Request For Comment (RFC793) shows, babysitting TCP isn't easy. Don't worry, the '7600A handles all the details. If you're curious, find out what state you're currently in by checking the TCP State field in register 0x23.

software to handle simple web, e-mail, and file duties isn't that hard at all.

However, if the '7600A put these higher layer protocols in silicon, there'd be little choice but to offer a rather complete implementation—probably overkill (and overpriced) for high-volume, cost-sensitive apps.

SILICON SERVER

Mike Johnson at iReady combined the '7600A with a PIC to create a minimalist web server (see Photo 1). Add a modem and you're on the air.

Don't believe that such a measly amount of silicon can push your big bad browser around? See for yourself at <http://xsmall.mycal.net>.

Mike was kind enough to send me a copy of his code, which demonstrates the '7600A's power to turn a lowly micro into a web warrior. Remember, the PIC ('16F84) has a 1024-instruction (14 bits each) code memory that leaves little room for bloat.

Even the limited 1-KB instruction capacity overstates the software's

e-mail (i.e., SMTP and POP3), web (i.e., HTTP and HTML), and file (FTP). But, the '7600A incorporates none of these protocols. Is this a problem?

Not really. Remember, the i1000 isn't an actual chip but rather a collection of know-how to enable you to design your own ASIC. Thus, each i1000-based chip should only include the functions required by an application (e.g., a chip designed to handle e-mail doesn't need web support).

By contrast, the '7600A (like all standard chips) has to strike a compromise. It needs enough features to meet the requirements of a broad range of applications; add too many extras, and customers will object to paying for features they don't need. The '7600A designers say TCP/IP and PPP are the sweet spot.

First, of the alphabet soup of protocols that abound, TCP/IP and PPP are among the most complex. A quick way to illustrate the situation is to check the file sizes of the various RFCs (Request for Comments; i.e., the seminal I-way specs from the DARPA days). TCP/IP and PPP comprise about 300 KB (that's just the main RFCs, not all the embellishments and add-ons) compared to 150 KB for FTP, 120 KB for SMTP, and a mere 35 KB for POP3.

Furthermore, TCP/IP and PPP are the common denominator for all the other protocols. By contrast, higher

level functions such as e-mail, web, and file are designed to function independently. If you just need FTP, there's no need to bother with HTTP, SMTP, POP3, and so on. Whichever protocol(s) you choose, you'll need TCP/IP and PPP under the hood.

Finally, many embedded Internet apps, such as security, data logging, maintenance, and inventory, require minimal ability to communicate. So, it's likely that a particular app won't require all the frills of a full-blown server. In fact, if you strip away a bunch of the fat you'll find that the

Listing 1—When it comes to web service, it's much easier to give than receive. To create the page, the PIC just spits out hardwired HTML (a). Otherwise, the software mainly just talks to the '7600A with register access macros (b), leaving the chip to handle all the details.

```

a)
00133 look_table
00134 web_page1
343C 3468 3474 00135 DT "<html>"
346D 346C 343E
343C 3474 3469 00136 DT "<title>Worlds Smallest Webserver</title>"
3474 346C 3465
343E 3457 346F
...

b)
; Open a socket to listen on port 80
writex SOCKET_OUR_PORT,.80
0297 3038 M movlw 0x38
0298 008D M movwf address
0299 3050 M movlw .80
029A 008E M movwf dbyte
029B 219F M call writwb
writex SOCKET_OUR_PORT+1,0x00
029C 3039 M movlw 0x38+1
029D 008D M movwf address
029E 3000 M movlw 0x00
029F 008E M movwf dbyte
02A0 219F M call writwb
writex SOCKET_CONFIGURATION,SOCKET_TCP_SERVER_MODE
02A1 3022 M movlw 0x22
02A2 008D M movwf address
02A3 3006 M movlw 0x06
02A4 008E M movwf dbyte
02A5 219F M call writwb
writex SOCKET_ACTIVATE_REG,1
02A6 3024 M movlw 0x24
02A7 008D M movwf address
02A8 3001 M movlw 1
02A9 008E M movwf dbyte
02AA 219F M call writwb

```

complexity. The HTML for the home page (stored in the PIC's code memory) chews up nearly half the space as shown in Listing 1a, and a significant portion of the code handles routine tasks such as macros to bit-bang read and write '7600A registers.

The guts of the server is less than 100 lines of real code, not counting macro expansions (see Listing 1b) that take five PIC instructions for each '7600A register access. The process boils down to: open a socket, listen for a request, increment the hit counter, and then dump the page.

RAY OF HOPE

The '7600A is an exciting development on the embedded Internet front—partly because it's available off the shelf, which opens the door for specialized applications and garage shops that would never be able to justify an ASIC.

Also, I like the idea of sticking with PPP and TCP/IP and leaving the decisions about higher level services to be determined by the needs of spe-

cific applications. I suspect time will show this to be the right decision. Few customers need all the trimmings, and most will be glad they don't have to pay for features they don't need.

I expect the '7600A to spawn an explosion of embedded 'Net gadgets because it makes networking simple and inexpensive, which it really wasn't before.

So, if you've been contemplating putting your embedded app on the Internet but couldn't justify the cost or complexity, you'd better think again. And if you weren't contemplating putting it on the 'Net, better think again anyway because the '7600A makes it real easy for you, and for your competitors. ☒

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.

REFERENCES

- [1] Stanford University Wearable Computing Lab, Information on embedding Linux, www.wearables.stanford.edu
- [2] Information about IA Open Networking Alliance, www.iaopennetworking.com

SOURCES

iChip

iReady Corp.
(408) 330-9450
Fax: (408) 330-9451
www.ireadycorp.com

Seiko Instruments, Inc.
Semiconductor Products Group
(408) 433-3208
Fax: (408) 433-3214
www.seiko-usa-eed.com/intcir/html/whatsnew

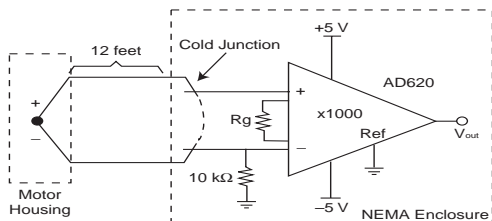
NET+ARM

NETsilicon, Inc.
(781) 647-1234
Fax: (781) 893-1338
www.netsilicon.com

CIRCUIT CELLAR Test Your EQ

Problem 1—The circuit shown in the figure is built on a standard FR-4 PCB. It is mounted in a NEMA enclosure on a factory floor. The temperature in the NEMA enclosure is 30°C. The thermocouple, an E-type, is mounted on the housing of a three-phase synchronous motor. The motor housing is 75°C.

The IA's offset voltage is a +125 μV. The IA's intended gain is 1000 V/V, but there is a 2% gain error that produces an actual gain of 1020. Assume the E-type thermocouple is linear and has temperature coefficient of 60 μV/°C. What is the output voltage of the IA?

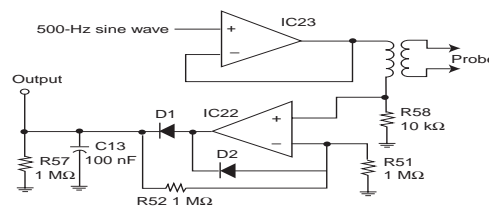


Problem 2—Given the same pressure and temperature conditions, is a cubic meter of humid air more or less dense than a cubic meter of dry air?

Problem 3—Can you rewrite this function to significantly increase its speed, without changing the basic algorithm?

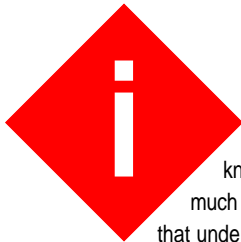
```
void bbs (int *numbers, int count)
{
    int i, j, temp;
    for (/9i=0; i<count; i++)
    {
        for (j=count-1; j>0; j--)
        {
            if (numbers[j-1] > numbers[j])
            {
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
}
```

Problem 4—What does this circuit do?



PRIORITY INTERRUPT

Spreading the Wealth of Knowledge



I'd like to say *Circuit Cellar* exists because I've always had a grand scheme for the perpetuation of technical knowledge, but I'm afraid it's a little simpler than that. I've just been doing things that seem worthwhile (and pretty much being on target). At times I've made silly gadgets and told funny stories, but I don't think anyone misunderstood that underneath all the amusing rhetoric and ridiculous contraptions, there was always a real engineering message.

Credibility is something we've always maintained even though our methods of presentation have frequently bordered on the extreme. Our engineering message is clear. The important thing around *Circuit Cellar* is application, application, application.

I've done a lot of things over the years to promote this ideal, and I get a lot of mail from the people who have been influenced by it. One of the inspirations that I look back on with great pride is our College Engineering Program. In this program, college professors register the students in their engineering classes and we give them free copies of *Circuit Cellar* for the semester. I learned at an early age that there were only two ways to learn things: trial and error, or somebody tells you. Our College Engineering Program is my way of giving our educational process a little "somebody tells you" boost.

I get a lot of positive correspondence from the program, but a recent letter from Jack Dillon of Analog Devices summed it up quite nicely. I thought I'd share some of the salient points in it with all of you.

Circuit Cellar has been a tremendous influence in my work and I cannot imagine where I would be today without it. Where I work, we bring in engineering students from a local university and develop them. Actually, it is a two-way learning experience. I am in awe of some of their skills and totally baffled by their lack in other areas. They typically know C and maybe some assembly, but when we get to running an app on an 8051 or other small micro, they really come up short. The look on their face, when you tell them that using `printf` will use up all of the available code space, is priceless. The good news is that their engineering (math, physics, etc.) and general computer skills are superb.

All of the engineering students we have hired have at least seen a copy of *Circuit Cellar*. The students that looked forward to every issue were a step above the rest. I needed to let you know how much influence your magazine has had with myself and the engineering students we work with.

Circuit Cellar has done a tremendous job of presenting the entire picture—lots of hardware and plenty of software, but most importantly, how the two are brought together to make the system work. That, from my perspective is what embedded systems are all about.

Our college program is just one aspect of the continuing *Circuit Cellar* message. Students, regular subscribers, and Internet techies can now enjoy even more applications each month through *Circuit Cellar Online*. We have lots of editorial coming from the winners of our Motorola Design99 contest who will be announced in the December issue and our recently announced Internet PIC2000 contest (running on *Circuit Cellar Online*).

This month we're extending our application goals even further as we inaugurate our regular print magazine contest—Design 2K. This year I am especially proud to announce that it is an 8051-based competition sponsored by Philips. Quite surprising, considering that we've published many 8051-family articles over the years, is that we've never had an 8051-exclusive contest before. The wide range of 8051-core products offered by Philips will no doubt result in some truly esoteric and ingenious entries.

Finally, our motto says this is a magazine by engineers, for engineers. I contend that engineers aren't just born. They are created through a process of technical understanding. The really smart ones typically choose a route around all that trial and error and tend to hang out at *Circuit Cellar* where we like to be the somebody who just "tells you."



steve.ciarcia@circuitcellar.com