

www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#124 NOVEMBER 2000

WIRELESS COMMUNICATIONS

Debugging Wireless Devices

Design2K Winning Projects

Build a
PIC-Based SBC

TPU Programming Basics





LOGIN/REGISTER

SITE NAVIGATOR

Buy Parts Now!

▶ myChipCenter

▶ Knowledge Centers

▶ Guides & Experts

▶ Product Reviews

▶ Communities

▶ Online Tools

▶ Circuit Cellar

▶ Resources

▶ Real Life

SEARCH CHIPCENTER 🔍

ASK US

THE ENGINEERS
TECH-HELP
RESOURCE

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

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



Test Your EQ
8 Additional Questions

CIRCUIT CELLAR ONLINE

THE MAGAZINE FOR COMPUTER APPLICATIONS

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

[This Month](#) [Archive](#) [About Us](#) [Contact](#) [Looking for More?](#)

THE ETHERNET DEVELOPMENT BOARD

by Fred Eady

Part 1: Putting it all Together

Fred moves the Florida room online as he follows through on the recent promise he made in the print version of *Circuit Cellar* to look at some simple, valuable Ethernet hardware. He makes things easy with a step-by-step process to get your Ethernet engine fully functional. This is part one of the series, so look for his upcoming articles to round out the picture.

October 2000



UML IN A PRODUCT'S LIFE CYCLE

by Venu Kosuri

Intended to be an introduction to UML, this article focuses on illustrating how to use its concepts in the development life cycle of a product. UML is a state-of-the-art modeling methodology useful for real-time systems, so if you're still a beginner, Venu will guide you through to the end.

October 2000



A BETTER BATTERY CHARGER

by Thomas Richter

It seems logical that there would be a push for smaller, lightweight, high-capacity batteries with today's outcropping of all kinds of portable equipment. Battery technology is making strides towards enhanced algorithms for faster charging and minimal battery damage. In this article, Thomas looks at the next generation of microcontrollers leading the way past the competition.

October 2000



EVERYTHING CHANGES—Using the *Const* Modifier

Lessons From the Trenches

by George Martin

Sometimes we have the knowledge, but don't utilize all the tools we have available to us. George looks at the forgotten modifier beyond *char*, *int*, *long*, and *float* for writing code in C. Remember the often overlooked *const* qualifier? Well, it can be used to ensure that the data won't be modified during execution, eliminating unexpected changes.

October 2000



ANYGATE IN A STORM

Silicon Update Online

by Tom Cantrell

This month, Tom sets us afloat with Micrel's SY55851U (Anygate). Rather than letting you sink in a sea of ones and zeros, Anygate can act as a lifejacket of sorts, making up for its lack of features with bipolar process and differential signaling for fast action. Musing about his recent reports about ON Semiconductor's OneGate, Tom wonders about the future of gate delay, and ultimately how high prices will climb.

October 2000

Mfr Data Sheets

App Notes

Ask Us

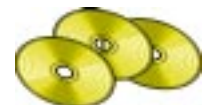
SuperSearch

Search Type:

Part Number

Search for:

Im317



SAVE
DOWNLOAD
TIME

Order your own complete set of *Circuit Cellar Online* archives.

First 6 issues July-Dec 1999 are available.

CD Includes:
All HTML files
Article PDF
Code files

Navigate and use just as you would online.

(no Internet connection required)



RESOURCE LINKS

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

12 **Wireless Devices**
Handling Power Efficiency and Debugging
Brian Branson & David Gonzales


20 **Simplified TPU Programming**
Jeff Loeliger


34 **A PIC17C44-Based Computer**
Duane Perkins

56 **Design2K Winners**
edited by Rob Walker

62 **Applications of PN Sequences**
Tom Napier

68 **Embedded Living**
The "S" is for Speed
Breathe New Life into Your Z180 Designs
Mike Baptiste

74  **Silicon Update**
eZ Does It
Tom Cantrell

78  **From the Bench**
Megawatt Castles Made of Sand
Exploring the Solar Cell
Jeff Bachiochi

Task Manager 6

Rob Walker
An International Blend

New Product News 8
edited by Rick Prescott

Reader I/O 11

Test Your EQ 84

Advertiser's Index 95
December Preview

Priority Interrupt 96
Steve Ciarcia
Upgrade Math

INSIDE ISSUE 124

EMBEDDED PC

40 **Nouveau PC**
edited by Rick Prescott

42 **RPC Real-Time PCs**
Debugging an FPGA Module
Finding the Right Test Case
Ingo Cyliax

48 **APC Applied PCs**
Rabbit Season
Part 3: Network Analysis
Fred Eady

An International Blend



the way I see it, this issue will hit the newsstands just in time for me to provide some last minute voting advice for the readers here in the U.S. Unfortunately, I follow politics just about as closely as I follow the U.S. pole-vault team, that is, every four years I manage to gain interest for a month or so.

This year it just so happened that the Olympic games were winding down as the election fanfare was heating up (which event contains more drama is debatable). But, I won't carry on about the election woes of the U.S., after all, the *Circuit Cellar* audience extends far beyond the territories of the U.S.

A recent check of our web site statistics shows that developers and designers in almost 70 countries (outside the U.S.) have subscribed to *Circuit Cellar* via the Internet (almost makes me want to vote for the candidate who helped "invent" the Internet). If there was an awards ceremony, we'd be listening to the Canadian national anthem and saying thanks to our friends to the north.

Australia would easily take the silver medal with more than twice as many online orders as the United Kingdom, which has the third highest order rate. Just out of the medals would be Mexico with a few subscriptions less than the U.K. However, until ordering a magazine subscription over the Internet becomes an Olympic event, Canada, Australia, and the U.K. will have to settle for bragging rights.

The international reach of *Circuit Cellar* doesn't stop with online subscription ordering. The Design2K contest sponsored by Philips turned out to be one of the most far-reaching contests we've had. As with any *Circuit Cellar* design contest, there was a variety of entries and you'll find the top projects starting on page 56 or on the Internet at www.circuitcellar.com/design2k/winners. Almost half of the projects we received for the Design2K contest were from outside of the U.S. With such an international field of entries, it's no wonder that eight different countries are represented among the winners.

Congratulations to all of the winners in the Design2K contest and thanks to all of you who submitted projects. More than one of the judges commented on the fact that the number of great projects made the judging process more of a challenge than they had expected.

There's no question that Philips did a great job of promoting and supporting the contest, so thanks to Sarah Ward and Kevin Gardner for keeping things running smoothly on their end.

Sure we're still a long way from offering *Circuit Cellar* in the language of your choice, but that doesn't mean you can't earn the respect of designers and developers around the world. Take a look at page 7 and finish your Z183 design by January 15 and you'll be on your way to winning international honor (and some prizes that are a lot more practical than an Olympic medal).

rob.walker@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Joyce Keil

MANAGING EDITOR

Rob Walker

CHIEF FINANCIAL OFFICER

Jeannette Ciarcia

TECHNICAL EDITORS

Jennifer Belmonte

Rachel Hill

Jennifer Huber

CUSTOMER SERVICE

Elaine Johnston

ART DIRECTOR

KC Zienka

WEST COAST EDITOR

Tom Cantrell

GRAPHIC DESIGNERS

Naomi Hoeger

Mary Turek

CONTRIBUTING EDITORS

Mike Baptiste Ingo Cyliax

Fred Eady George Martin

George Novacek

STAFF ENGINEERS

Jeff Bachiochi

Anthony Capasso

John Gorsky

NEW PRODUCTS EDITORS

Harv Weiner

Rick Prescott

QUIZ MASTER

David Tweed

PROJECT EDITORS

Steve Bedford Bob Paddock

James Soussounis

David Tweed

EDITORIAL ADVISORY BOARD

Ingo Cyliax

Norman Jackson

David Prutchi

Cover photograph Ron Meadows—Meadows Marketing

PRINTED IN THE UNITED STATES

ADVERTISING

ADVERTISING SALES REPRESENTATIVE

Kevin Dows
(860) 872-3064

Fax: (860) 871-0411
E-mail: kevin.dows@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, www.circuitcellar.com/subscribe.htm, or subscribe@circuitcellar.com

PROBLEMS: subscribe@circuitcellar.com

GENERAL INFORMATION:

TELEPHONE: (860) 875-2199 FAX: (860) 871-0411

INTERNET: info@circuitcellar.com, editor@circuitcellar.com, or www.circuitcellar.com

EDITORIAL OFFICES: Editor, Circuit Cellar, 4 Park St., Vernon, CT 06066

AUTHOR CONTACT:

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

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

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

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

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

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 by Circuit Cellar®.

The information provided by Circuit Cellar® is for educational purposes. Circuit Cellar® makes no claims or warrants that readers have a right to build things based upon these ideas under patent or other relevant intellectual property law in their jurisdiction, or that readers have a right to construct or operate any of the devices described herein under the relevant patent or other intellectual property law of the reader's jurisdiction. The reader assumes any risk of infringement liability for constructing or operating such devices.

Entire contents copyright © 2000 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 Rick Prescott

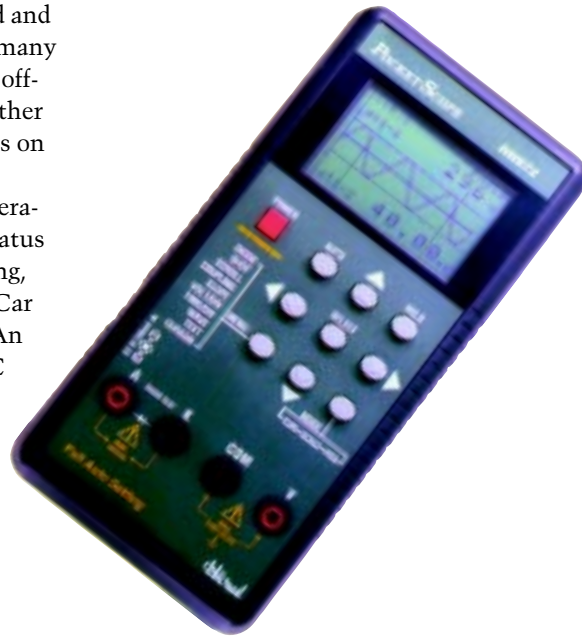
PALM COMPUTING DATA OFF-LOADING

The **HandCar** software for palm hand-helds can read and launch HOBO and StowAway data loggers. Data from many data loggers can be stored in the palm device and later off-loaded to a PC for graphing, analysis, or exporting to other programs. **HandCar** allows users to manage data loggers on location using the palm device.

The software provides functions to verify logger operation, view current measurements, and check battery status (for loggers that support this function). During launching, the software synchronizes the data logger clock. **HandCar** runs on the Palm III, Palm V, and Palm VII organizers. An optional palm-to-PC interface can off-load data to a PC without a docking station.

HandCar software costs \$40.

Allison Technology Corp.
(281) 239-8500
Fax: (281) 239-8006
www.atcweb.com



LITHIUM-ION BATTERY CHARGER CONTROLLER

The **MAX1737** is a standalone battery charger controller for one to four lithium-ion cells. This device features better than 0.8% battery-regulation voltage accuracy, 90% conversion efficiency, and a complete state machine to safely control the charging sequence.

The step-down, switch-mode DC/DC converter uses a small external dual N-channel FET as a power switch and synchronous rectifier to provide several amperes of accurate charging current and sustain efficiency over a wide input voltage range. An internal voltage regulator powers the IC, allowing operation up to 28 V, and the controllers, ability to work with a duty cycle of up to 98% results in low dropout voltage.

The controller regulates the voltage setpoint and charging current with two loops working together to transition between current regulation and voltage regulation. To service the system load during charging, an additional control loop monitors the total current drawn from the input source. This loop can lower the charging current to prevent

overload of the input supply when the system load increases, allowing the use of a low-cost wall adapter.

A built-in safety timer automatically terminates charging after a selectable time limit is reached. Battery temperature is monitored by an external thermistor to prevent charging if the battery temperature is too high or low. The chip charges near-dead cells if the battery voltage is below 2 W per cell. Fast charge, full charge, and fault conditions are indicated via LEDs driven by open-drain outputs.

The controller comes in a 28-pin QSOP package and is specified for the extended industrial temperature range (-40°C to 85°C). Prices start at \$2.85 for 1000. A preassembled evaluation kit is available.



Maxim Integrated Products
(408) 737-7600
(800) 998-8800
www.maxim-ic.com

NEW PRODUCT NEWS

MOTION CONTROLLERS FOR STEPPING MOTORS

With speed, programmable pulse and direction output, and user-selectable profiling modes (S-curve, trapezoidal, velocity contouring, electronic gearing), the **Navigator MC2500** is ideal for applications such as medical automation, materials handling, test equipment, and robotics.

Features include asymmetric acceleration and deceleration, on-the-fly velocity and acceleration changes, trace capabilities for system performance checks and diagnostics, and on-the-fly stall detection. The controller provides multiple

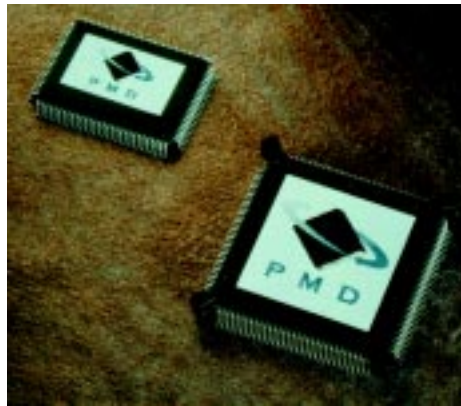
breakpoints per axis to offer precise sequencing and event control by the application program. It accepts feedback from an incremental encoder (up to 5 megacounts per second) or from an absolute encoder or resolver (up to 160 megacounts per second) to read the current axis position.

Input signals include two limit switches (one for each

direction of travel) and one home indicator. One general-purpose programmable input and output signal per axis is provided. In addition, eight general-purpose analog (0 to 5 V) and 256 general-purpose discrete inputs/outputs (16-bit wide) are available.

Consisting of two components, a 132-pin processor and a low-pin logic device, the chipset allows off-loading of resource-intensive motion control functions from the host processor. The instruction set supports more than 130 commands, offering flexibility to designers during application programming.

Engineering samples are available. Prices start at **\$46** in OEM quantities. The Navigator Developer's Kit is offered for **\$995**.



Performance Motion Devices, Inc.
(781) 674-9860
Fax: (781) 674-9861
www.pmdcorp.com

READER I/O

THAT'S THE WAY I LIKE IT

Normally I don't write to magazine editors, I just show my support by continuing to buy. However, after reading the September issue (122) I am moved to write, and I have to say how great I thought this issue was.

Specifically, there were a large number of complete projects, spanning a whole range of subjects and attractive to people with a range of skill-sets and accomplishment. I know that putting together each article involves a lot of work, and even more work to get it together for each magazine issue. This one was the best yet. Thank you.

Andrew Errington

SIMPLE AND EASY

Steve's brief comments on the Napster fiasco in your "Imputed Liability?" column (*Circuit Cellar*, 123) was one of the most logical and common-sense treatments of the hoopla that I have seen, anywhere.

I've been a long time reader and as a kid followed Steve's articles on homebuilt weather stations and such in *BYTE*, I appreciate his common-sense approach to problems. It's very refreshing these days and I am glad that *Circuit Cellar* has retained this philosophy. Keep it up! and thanks for the great mag.

Jim Fitzgerald

A LITTLE HELP?

Does anyone have a schematic for a programmable pushbutton code lock, hopefully using a PIC?

Jon Payne
jppayne@thepaynes.net

Although we don't object to posting reader questions in this forum, the quickest answers to your design questions may be available on the Circuit Cellar newsgroups, which are accessible from our homepage.

ALMOST, BUT NOT ENOUGH

After much anticipation, I received October's issue. My excitement of understanding and possibly using the PCI bus was crushed. It seems the "Catch-

ing the PCI Bus" series has ended with just theory.

I hope Ingo will continue this series because I have a strong desire to build a PCI card and have been pursuing this on my own and I'm making many mistakes.

I started with a Logitech ISA board and added a daughter board so I could play with addressing and the data bus on the ISA slot. Now, I would like to make a PCI card. A guide would be immeasurably valuable. Can someone direct me to a guide for building a PCI card?

Trevor Pearce
tpearce@dwtunnel.com

NAVIGATING A DESIGN CONTEST

After reading Riccardo Rocca's Design99 project article, I think a great competition could be considered which uses Mr. Rocca's chosen application (autonomous GPS). I am thinking of something like a San Francisco to Sydney race, sort of a cross between around-the-world ballooning and engineering school robot wars.

Using dry hull designs and rigid airfoils, extreme seaworthiness could be achieved with craft limited in size to avoid creation of shipping hazards (say, 6 feet and 75 lbs max). A successful entry in this competition would utilize inputs from many disciplines, from oceanography to applied engineering.

To make the completion of the race interesting, a suitably small target would be designated as the finish line. With on-board error-recovery and telemetry provisions, a launch of these intrepid creations might be something like NASA's sending a probe to mars.

Jeff Spellman

There are certainly some logistical problems that would have to be overcome for a contest like this to take place, but we thought it would be interesting to hear what Circuit Cellar readers have to say.

To get the variety of projects that are submitted for Circuit Cellar design contests, the contests generally specify a processor and not an entire application. However, being a Circuit Cellar reader means you can voice your opinion about the way we do things. What kind of design contest would interest you? Send comments or suggestions to: contest.administrator@circuitcellar.com.

FEATURE ARTICLE

Brian Branson
& David Gonzales

Wireless Devices

Handling Power Efficiency and Debugging

Facing the increasing demand for integrated wireless solutions, debugging capability and power efficiency have become crucial. Branson and Gonzales address these issues as well as the idea of global standards that will make life easier.



As the demand for highly integrated wireless solutions continues to increase, silicon providers respond with a broad array of intellectual property, increasingly dense technologies, and an industry-wide focus on System-on-Chip (SoC) design tools and integration methodologies. The trend towards reducing the number of components in these systems clearly has the advantage of reducing cost, power consumption, and manufacturing complexity.

On the other hand, product developers are left with the daunting task of creating complex devices with increasingly reduced visibility of subsystem interaction. These devices use programmable microcontroller (MCU) and

digital signal processor (DSP) cores coupled with embedded memories and a myriad of peripheral modules on a single chip. The proliferating market of highly integrated systems obviously is increasing the need for improved methods of system validation.

SoC design methodologies for programmable cores now include static debug blocks that may be used during the early stages of product development. By including additional debug-related capability on-chip, suppliers offer designers the ability to fully understand the behavior of a given system, including validation of both hardware and software architectures and their interdependence. This is essential for evaluating real-time power consumption in Internet-ready hand-held devices.

This article explores the issues associated with developing power-efficient hand-held wireless devices and the necessary on-chip debug capability needed for rapid product development. Debugging highly-integrated multiple core systems on a single chip will be discussed using the M-Core M341 micro-RISC processor as an example. We'll also discuss an implementation of a real-time debug port based on the IEEE Industry Standards and Technology Organization (ISTO) Nexus 5001 Forum specification.

To better appreciate the problems of developing a low-power, high-performance system, let's look at a cellular handset. A digital cellular handset can be partitioned into three main sections (see Figure 1). The RF section receives and transmits analog and digital information; the analog

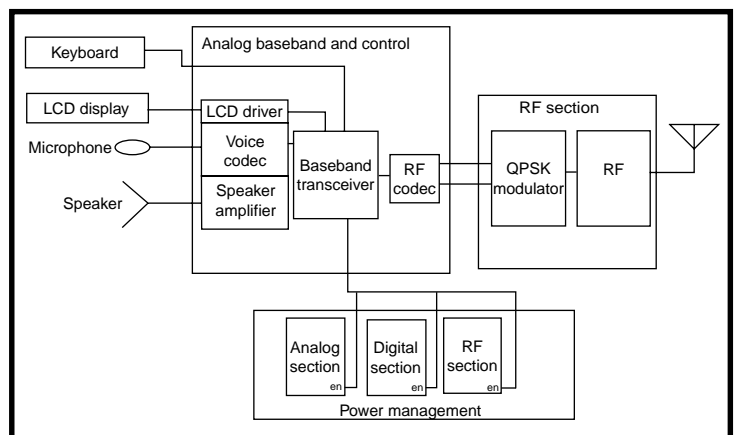


Figure 1—This is a block diagram of the digital cellular handset.

baseband and control section handles intermediate frequency conversion, user interaction, and power control; and the power management section distributes and manages power to all elements of the handset.

In first and second generation digital cellular solutions, overall baseband power consumption is derived from the combination of standby leakage power, active power for time-based protocol software stacks and data (voice) transmission, and system event power induced by an active page, call, or other user-induced event (see Figure 2). The relative periods of standby and active power can be calculated accurately based on knowledge of the wireless protocol. Hence, standby power consumption can be estimated via leakage current information for a given technology and the amount of time the chip stays in this inactive mode.

Active power consumption is more difficult to estimate. But, for repetitive software stacks performing known protocol functions, this too can be accurately determined. Consider then the problem of estimating and optimizing on-chip power consumption during user-induced events such as WAP browsing, high-speed down/up link transactions, or Motion Picture and Entertainment Group (MPEG) structured audio activity.

The embedded system contains all the necessary capability to perform these functions, even in parallel with other events, but their behavior is less deterministic. The software that is written to handle this multitude of system activity must be carefully optimized to improve battery life for a particular application. Prior studies indicate that the three main blocks of the cellular handset each consume 15 to 50 mA of current depending on their states of activity (see Table 1).

REAL-TIME ANALYSIS

Lab bench analysis of prototype systems permits conventional methods of evaluation such as circuit boards with logic analyzer

Table 1—Check out the power consumption numbers.

Task	Digital power	Analog power	RF power
Network access	40 mA	20 mA	40 mA
Call service	20–30 mA	20 mA	50 mA
3G playback	35 mA	15 mA	20–30 mA

interfaces. Typically these boards provide a means for initial powerup and integration of software and hardware modules. Each core in the baseband processor chip is evaluated individually in a static debug form; each is put in a special mode of operation that checks its programmer model register and memory while single-stepping a test program (downloaded from a host computer). After the system passes the “smoke test” where each processor exits reset and performs initialization functions correctly, the task of debugging the real-time kernel and interrupt structures begins.

Debugging a real-time wireless device traditionally requires a logic analyzer monitoring an external bus interface where at each clock cycle a sample of bus activity is recorded. But, this is expensive and physically impossible for microcontrollers and DSPs because they operate above 100 MHz.

When bus interfaces are not available, developers embed printf statements at strategic points. So, data needed is sent to a peripheral port and retrieved by a host processor. The information is minimal and creates intrusive delays in the application. As software layers became more complex, this method became too time-consuming. Now, microcontroller developers are demanding reliable and cost-effective solutions from suppliers.

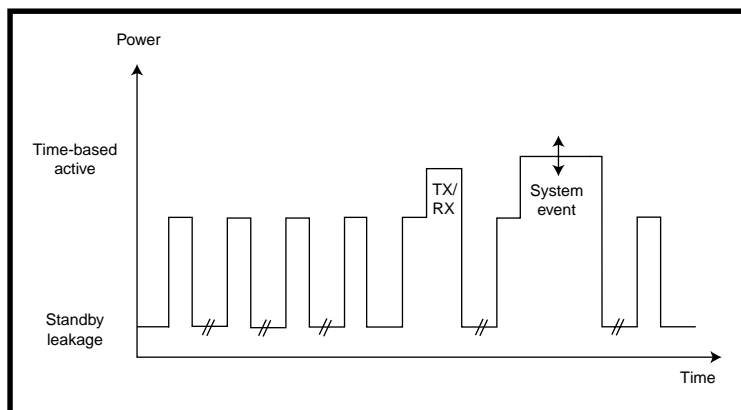


Figure 2—This demonstrates the characteristic power consumption for the cellular handset.

IEEE-ISTO NEXUS 5001 FORUM

During the past two years, a consortium of companies has worked diligently to address the issue of real-time debugging of highly-embedded systems. The automotive, telecommunications, and network appliance industries have driven this effort to reduce time to market for new products. The consortium began with five companies and has grown to more than 25 participating in the definition of a specification, which is now governed by the IEEE-ISTO Nexus 5001 Forum. Current participants include Accurate Technologies, Motorola, Embedded System Products, Hitachi, Mitsubishi, Hewlett Packard, Ashling Microsystems, and more.

The IEEE-ISTO Nexus 5001 Forum’s goal is to define a common set of microcontroller on-chip debug features, protocols, pins, and interfaces to external tools that can be used by real-time embedded application developers. At this time, revision 1.0 of the specification serves as the model for future on-chip debug resources implemented by silicon vendors.

The forum is comprised of four groups. The business group constructs and coordinates activities among the companies involved. Companies in the technical group create specifications and coordinate technical activities with the other groups. The third group concentrates on validation. This bunch of companies develops verification methodology for the architectures and tools. The final part of the forum is the API group, which is charged with developing abstraction layers and a software interface for tools and silicon. [1]

One major objective of the forum is to help development tool ven-

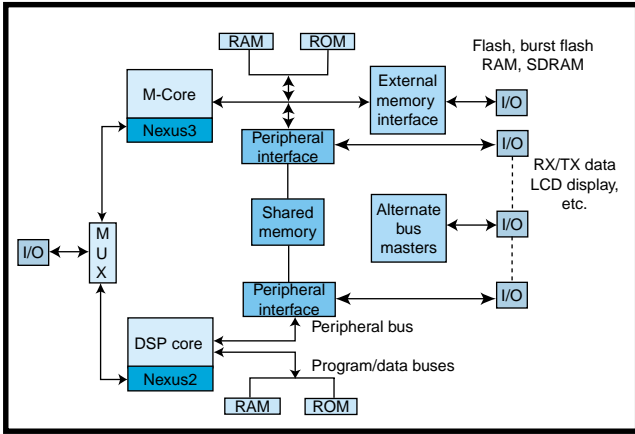


Figure 3—The dual-core cellular handset baseband transceiver is shown here.

dors more easily provide a standard set of tools that may be used on a number of embedded microcontrollers. In the spirit of reusability, many semiconductor vendors have been recognized for having debug ports and tool sets that sufficiently address the static debug requirements of their architectures.

Providing a cost-effective yet powerful migration path to a standard set of dynamic debug features is one of the key goals of the forum. More than 70% of leading embedded microcontroller vendors have dedicated circuits and pins that assist in new product development based on the IEEE 1149.1 Joint Test Action Group (JTAG) four-wire serial interface.

The JTAG pins and protocol help developers with static debug methodologies in a master/slave mode, but there are no means for the embedded microcontroller to initiate real-time information transfers to a host computer. The Nexus standard addresses this need with a scalable set of features whereby existing debug blocks may be used with an extensible auxiliary port. The features associated with this new auxiliary

port focus on real-time transfer of information to and from the embedded microcontroller.

The Nexus 5001 Forum has categorized static and dynamic debug features according to class levels to address various levels of development needs. These classes provide a means for implementing a scalable

debug architecture that can address different market segment requirements. Also, it should be noted that when a product is in development, you want to have as many debug features as possible because of constraints of time to market.

After a device is put into production, however, it may not

be necessary or desirable to have all the development features and pins. You may save by implementing a scalable debug port that meets only the requirements needed for specific stages of the product lifecycle.

EMBEDDED PERFORMANCE

As stated, the heart of the cellular handset is the baseband transceiver that performs all computations relative to call service, Internet web interaction, and handset control. Figure 3 shows a block diagram of a Motorola wireless baseband processor, including separate MCU and DSP core complexes interfaced to separate on-chip RAM and ROM memories and core-specific peripheral and I/O functions.

In order to understand each core's operation and the way cores interact, you should pin-out the internal core buses to external pads, thus achieving good visibility of core bus cycles. However, because of the desire to reduce I/O and package costs, this becomes prohibitive. Nonetheless,

system hardware and software architects still desire a method of understanding stand-alone and integrated core behavior.

WAP DEBUG

Rather than elaborate on the details of the IEEE-ISTO Nexus 5001 specification, we're going to evaluate some of the needs of debugging an Internet-ready handset. The WAP architectural specification focuses on optimizing for efficient use of device resources. But the task of providing a communications protocol as well as an Internet protocol layer dictates that the RAM be 1 to 4 MB and the flash ROM, which holds the kernel, be 256 to 512 KB.

Because the number of external accesses to RAM directly affects power consumption, the microcontroller engine must have an efficient instruction set, resident cache, and Memory Management Unit (MMU) to reduce external bus transactions. One of the important power consumption goals is to write the handset code so that it efficiently uses the cache.

After the cache and MMU are enabled, the interaction of the core and cache is no longer visible unless there is a cacheable instruction or data miss resulting in an external access to fill the cache. This problem is aggravated when you must debug code that exhibits abnormal behavior in real time or there is a need to capture power measurements when running specific code.

The M-Core M341 architecture implements a Nexus 5001 port for accessing user resources using a high-speed output port to transmit real-time program and data information.

The feature set of the Nexus 5001 port is from Class 3, providing static debug capability and real-time process identification, program trace, data trace, and read/write access to M-Core Local Bus (MLB) resources. An efficient mode of transmission must be used so that real-time 32-bit address and data values are reported through a 2- to 8-bit output port (see Figure 4).

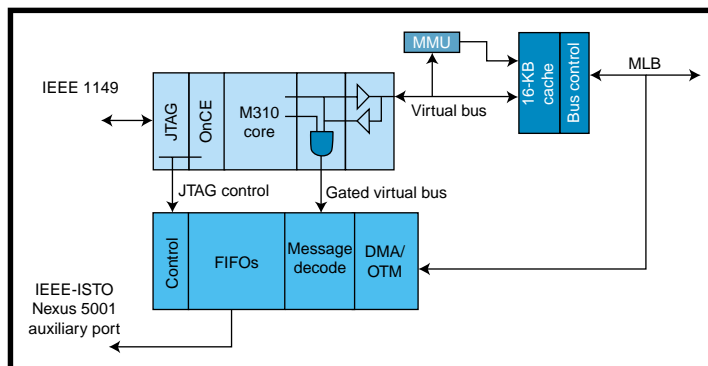


Figure 4—Here's the M-Core M341 processor with the Nexus 5001 debug port.

APPLYING REAL-TIME FEATURES

A set of data packets commonly referred to as public messages in the Nexus 5001 specification has been defined for efficient transfer of debug information between the embedded processor and a development system. Public messages consist of a transfer code, or TCODE, source processor identification number, and the data associated with the particular feature being accomplished. The key ingredient of public messages is their efficiency, which means packets may be variable in length depending on the TCODE.

A JTAG serial interface controls messaging capability. The interface couples with a OnCE static debug block and provides access to all Nexus 5001 registers on the M-Core M341 processor. Messaging is enabled prior to deassertion of the reset pin so that exit from reset can be monitored.

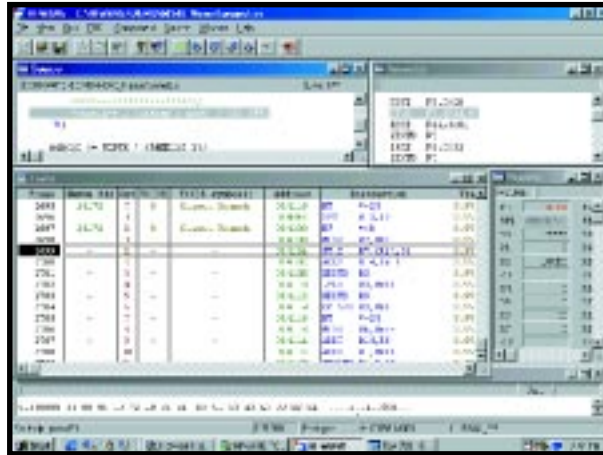


Photo 1—HiWave's Hi-WARE debugger and the Tektronix TLA714 trace buffer are shown here.

MONITORING PROGRAM FLOW

Program behavior includes changes in the program counter because of branching, jumping to subroutines, and servicing interrupts and exceptions. Analysis shows that 12 to 13% of instructions executed in a program are change of flow. Therefore, it is not necessary to report every instruction's address, rather only the change of flow. Where you are relative to a reference

start address and where you are heading when you change program flow should follow the source listing.

Three types of public messages provide program flow behavior. Real-time operating system (OS) debug must have a means for reporting a process ownership identifier. The Ownership Trace Message objective is to give the current value of the data bus when a process writes to a special address (user base address). This is where Nexus 5001

snoop logic comparators capture the data bus. Thus, whenever a context switch of the OS occurs, a process identifier may be transmitted using the ownership trace message. This is a key part of correlating virtual-to-physical address maps of the MMU while sending messages to the source level debugger.

Branch trace messages report when direct or indirect branch instructions are executed. The difference in the

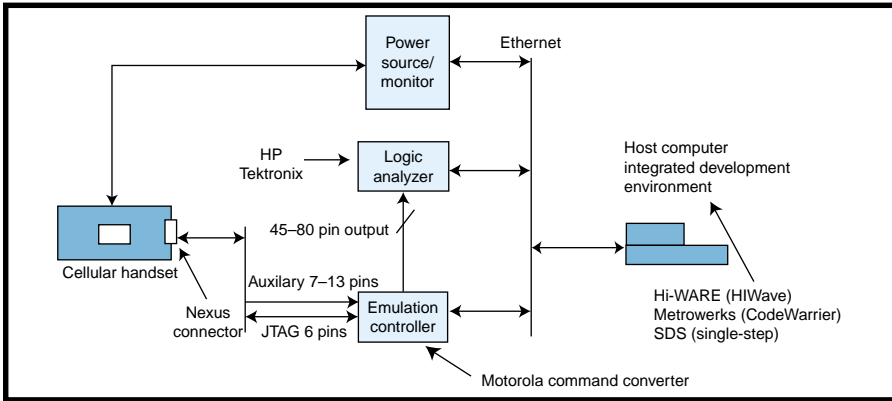


Figure 5—Here's a real-time debug environment.

messages is that during a direct branch, the only information needed is the number of instructions executed since the last change of flow. A reference address using a synchronization message is normally transmitted to discover the program counter's location. After that occurs, all references are made to that address until an indirect change of flow occurs. This reduces the number of bits transmitted in a message. Indirect branch messages report the number of instructions executed since the last change of flow and the

address where the program counter is jumping to, thus establishing a new reference address.

If you need to report specific memory accesses, the watchpoint message does the job. This message triggers hardware comparators and complex access qualifiers that monitor the M-Core virtual bus.

The idea is to set a watchpoint trigger where a signal and message can be transmitted. The message tells which of the watchpoint triggers occurred. This is especially valuable for

debugging variable writes. For example, if you have a global variable that is being modified by a number of processes and you want to pinpoint which of those processes is accessing that variable, the watchpoint message is the tool to use. This feature also asserts an event pinout that may trigger a logic analyzer to capture specific public messages or peripheral signals. For power analysis, you can use the trigger to capture current measurements at specific points in code or data accesses that may be useful for pinpointing power-consuming hot spots.

MONITORING DATA VARIABLES

Trace messages provide a means for reporting real-time data access to memory locations. More data loads and stores are reported than program flow changes. Analysis indicates that as much as 25% of instructions executed in a program are data accesses.

Data messages report stack contents, global and local variables, and peripheral port accesses. To control the number of data messages transmit-

ted, data trace qualifiers include the access type (i.e., read and write or either) and a start and stop address range.

If the data address and access qualifiers are met, data messages are generated and sent to the debug port. This narrows the window of memory locations that may initiate a data message.

Sending only the unique portion of the data address instead of the complete address reduces the output bandwidth requirement for the debug port. Consequently, a data trace message is reconstructed relative to each prior message using a synchronization message as a starting address.

REAL-TIME DATA ACCESS

The M-Core M341 Nexus Port provides access to the MLB-mapped resources via the JTAG port. A ready-for-transfer pin (RDY) was added to increase the transfer rate. Note that calculations show that accesses to the read/write data register allow for a throughput of 1 MB on an M-Core M341 microcontroller operating at a 50-MHz system clock.

Block transfers are possible with only a single setup of read/write control and address registers. This permits 32-bit transfers in 38 JTAG clocks, where each JTAG clock is half of the system clock.

This capability significantly reduces program and data load times and enables you to examine arrays of memory without stopping the application. Data trace messages only report data movement within a well-defined data window, however, read/write access permits accessing values asynchronously. This feature is useful for downloading new filter coefficients or encryption keys when you are testing communication protocols. Another important use of this feature is the retrieval of power data values that may be built into the power management unit of the handset.

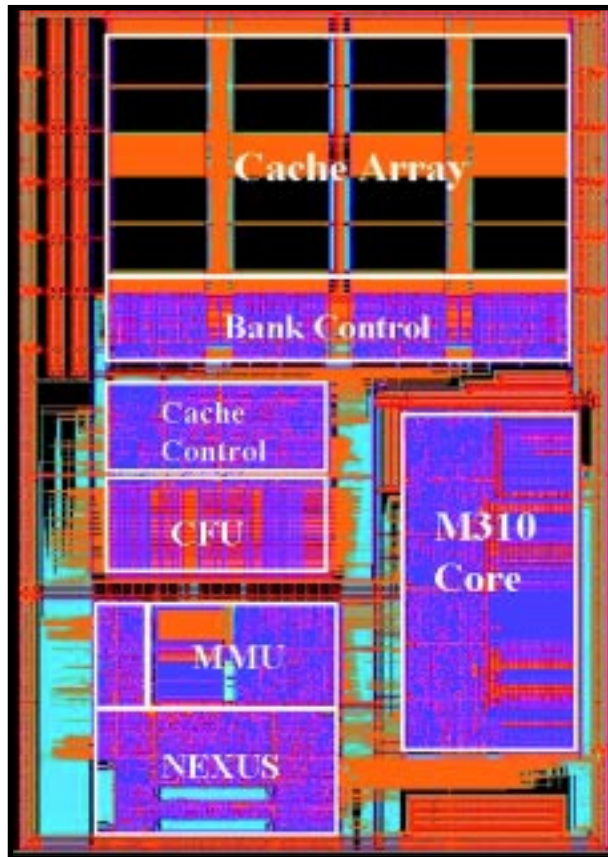


Photo 2—Take a look at the M-Core M341 processor die.

DEBUG TOOL SUPPORT

Two important ingredients of a successful reduction in development time is on-chip circuits such as the Nexus 5001 port and the development tools that support the Nexus interface. The Nexus 5001 specification defines pins, connectors, and the protocol for transferring messages to and from the host computer. However, it is difficult to define stringent rules for Nexus register sizes, bit positions, and other implementation-specific details that may not suit particular semiconductor vendors' architectures. An application protocol interface (API) that abstracts implementation details is the ideal solution for tool vendors.

Figure 5 illustrates a lab debug environment that uses an integrated software tool set coupled with a logic analyzer and power source/monitor for complete handset development. The emulation controller provides the abstraction layer so that an API that provides details of the emulation controller-to-Nexus interface without burdening the tool vendor may be

defined. An FPGA was added to the emulation controller that would reconstruct the full message from the two Nexus Port output pins to a 40-bit message with message trigger signal. This improves utilization of the logic analyzer's trace buffer.

Classic debuggers use a load, arm, go scenario in which the debugger starts the target processor (s) running, and then the debug environment is frozen until the target processor reenters a debug or interrogation mode. To fully exploit the real-time debug capabilities of the M341 Nexus Port, the debugger must permit interrogation of target resources as it executes code in real time.

During initial development of the M341 Nexus Port, a HIWARE (HIWave) debugger was interfaced to a Tektronix TLA-714 logic analyzer (Metrowerks, CodeWarrior, and HP logic analyzers also support Nexus). The HIWave debugger directly interacts with the Tektronix logic analyzer to arm its trace buffer for message captures and later displays the trace buffer contents within the HIWave environment (see Photo 1).

Because the M341 processor has a different instruction set and programmer's model than the DSP56600 architecture, a dual-integrated environment with split windows (one for each processor) debugs the baseband transceiver. One emulation controller communicates with each processor using the JTAG protocol. A semaphore configuration in the dual debugger's control module regulates traffic to the emulation controller so that there are no message collisions when communicating with either processor.

PENALTIES

The additional feature set of the Nexus Port doesn't come without some die area and power penalty. Therefore, during its implementation, all sub-module clocks were gated off for inactive circuitry and the message

decode state machine and logic were enabled/disabled via Nexus control. Special consideration was given to the message queues that reduce power.

The output port was made variable width to accommodate a 2- or 8-bit width. This is important from a development perspective. During lab analysis, the 8-bit port would be used because there was room to add larger connectors on the evaluation cards. But when the handset ergonomics were finalized and the high-density, double-sided surface-mount board was used, we decided that a reduced bandwidth over the output port was feasible.


Overall, the Nexus Class 3 implementation was 7.5% of the M341 processor area (see Photo 2). But considering the size of the complete baseband transceiver, it is small relative to the addition of on-chip memories and the DSP.

STRIVING TOWARD THE GOAL

Cellular-based products that interact with the Internet are growing at a phenomenal rate. The increase in features will lead to more sophisticated portable systems and challenge designers to provide more features that consume less power. Therefore, system validation will play a more important role in SoC design methodologies in order to quicken time to market.

Significant effort is underway throughout the electronics industry to improve tools and methods for designing complex embedded systems. The IEEE-ISTO Nexus 5001 Forum is a testament to this and demonstrates that there is a dire need to standardize a set of features, protocols, pins, interfaces, and tools for rapid development of real-time microcontroller-based products. Originally targeted for automotive applications, the Nexus 5001 Forum has extended the scope of this effort to encompass telecommunications, industrial, and portable handheld products. The problems of real-time visibility and embedded microcontrollers are similar if not identical in most products.

Companies will have special cases for solving specific design issues, so the proposed global standard allows for vendor-defined blocks for special fea-

tures, all addressed by a common protocol. The full specification may be downloaded from www.ieee-isto.org/Nexus5001/index.html. 

Brian Branson is an M3 Core development manager at the Motorola M-Core/ColdFire Technology Center. With Motorola for 15 years, Brian has been involved with M-Core and PowerPC microprocessor development and product integration, as well as fast static RAM and applications-specific RAM designs. He earned a B.S. in Electrical Engineering from Colorado State University and holds eight U.S. patents. You can reach him at brian.branson@motorola.com.

David Gonzales is a senior member of the design team at the Motorola M-Core/ColdFire Technology Center. During the past 22 years at Motorola he has worked on 8-, 16-, and 32-bit microcontrollers and DSPs. He earned a B.S. in Computer Science from St. Edwards University, Texas. He has written more than 40 publications about microcontrollers and holds two patents. You can reach him at david.gonzales@motorola.com.

REFERENCE

- [1] D. Gonzales, "Evaluation of a New Evolution Port Using an M-Core Architecture System," Motorola M-Core Technology Center, Austin, TX.

SOURCES

M-Core M341 micro-RISC processor and OnCE static debug block
Motorola, Inc.
(847) 576-5000
Fax: (847) 576-5372
www.motorola.com

Debug specification
IEEE Industry Standards and Technology Organization:
Nexus 5001 Forum
(732) 981-3434
Fax: (732) 562-1571
www.ieee-isto.org/Nexus5001

Hi-WARE
HIWave Technologies, Inc.
www.hiwave.com

FEATURE ARTICLE

Jeff Loeliger

Simplified TPU Programming

Have you ever thought about using a TPU but were discouraged by the difficulty? Have no fear, Jeff's here! Step by step, Jeff explains the TPU mysteries in detail. After reading this article, you'll be ready to tackle your own custom functions.



Many people assume the Motorola Time Processor Unit (TPU) is difficult to program. One of the reasons is because it is usually described in a hardware-oriented manner. This often discourages people from even trying to write custom functions that could make the TPU a valuable tool.

In this article, I'll use a different approach. I'll present the important facts in a style more suited to programmers. I will not, however, ignore the hardware, because it is fundamental to TPU operation.

TPU OVERVIEW

The TPU is a real-time processor optimized for fast complex I/O signals. The TPU doesn't need the CPU to generate output signals or handle input signals; therefore, it reduces loading on the CPU. In standard microproces-

sors, the CPU controls the timer hardware directly. But, the TPU sits between the CPU and timing hardware.

Traditionally, timers have fixed functionality, but all TPU channels are identical and can perform any timer function. Another advantage is that the TPU is fully programmable and not just configurable. The TPU has its own program memory to write programs.

The TPU is a memory-mapped peripheral with an interface made up of four parts. System Configuration configures things that affect the whole TPU, such as timebase and interrupt configuration. The development and test registers debug and test the TPU. Channel controls configure the way each channel works. And, parameter RAM is used for communication between the CPU and TPU.

Every customer and application has different timer requirements. Because every TPU channel can run any timer function, you can create any mixture of timers. Enhancing the functions for new requirements can be done in software without changing hardware.

The TPU was designed to replace custom ASIC hardware. This simplifies the system hardware and reduces cost.

The TPU can generate a variety of standard and custom input or output functions. The first thing you should do is check if the function you require has already been written. If not, a custom function can be written using TPU microcode. A list of all existing functions is shown in Table 1. Custom functions can take advantage of the TPU's aptitude for generating irregular, complex waveforms.

PROGRAMMING

There are many approaches to learning a new programming architecture. I'd like to start with the pro-

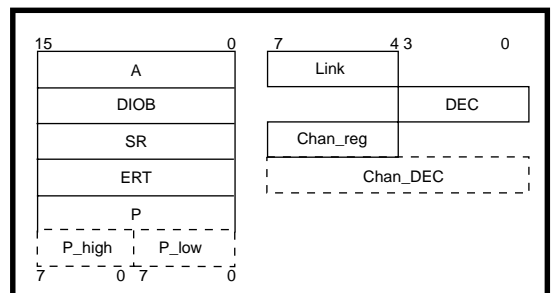


Figure 1—You must understand the TPU programmer model. The registers represented by dotted lines are concatenated from other registers.

grammer's model (see Figure 1). Each of the registers shown in the model has a specific and unique function.

Register A is a general-purpose accumulator used for arithmetic operations. Register DIOB (data I/O buffer) also is an accumulator register. In addition, it can be used as an index register and can access parameter RAM.

Register SR (Shift Register) can be used as a general-purpose accumulator register and has a special feature allowing a shift right by 1-bit operation.

Register ERT (Event Register Temporary) is a mailbox for transferring information to and from the timer channel hardware of each of the channels. The timer channel value is automatically copied to the ERT register every time a new function state is entered. This saves a couple of instructions in code by not having to explicitly access the channel hardware and load the ERT.

Register P (Preload) can be used as a general-purpose accumulator and access parameter RAM. This is the only register that can be accessed as two 8-bit registers, called P_high and P_low.

Register LINK is a special-purpose 4-bit register that enables a function running on a TPU channel to pass a signal to another TPU channel. This

Source	Nickname	Name
TPU "A" mask	PPWA	Period- /pulse-width accumulator
TPU "A" mask	OC	Output compare
TPU "A" mask	SM	Stepper motor
TPU "A" mask	PSP	Position-synchronized pulse
TPU "A" mask	PMA/PMM	Period measurement with addition/missing transition detect
TPU "A" mask	ITC	Input capture/transition counter
TPU "A" mask	PWM	Pulse width modulation
TPU "A" mask	DIO	Discrete input/output
TPU "A" mask	SPWM	Synchronized pulse width modulation
TPU "A" mask	QDEC	Quadrature dedode
TPU "G" mask	PTA	Programmable time accumulator
TPU "G" mask	QOM	Queued output match
TPU "G" mask	TSM	Table stepper motor
TPU "G" mask	FQM	Frequency measurement
TPU "G" mask	UART	Universal asynchronous receiver/transmitter
TPU "G" mask	NITC	New input capture/transition counter
TPU "G" mask	COMM	Multiphase motor commutation
TPU "G" mask	HALLD	Hall effect decode
TPU "G" mask	MCPWM	Multichannel PWM
TPU "G" mask	FQD	Fast quadrature decode
TPU3 mask	SIOP	Serial input/output port
TPU3 mask	ID	Identification
TPU3 mask	RWTPIN	Read/write timers and pin

Table 1—Among the TPU functions, the "A" mask is designed for automotive users and the "G" mask is suitable for general use. Some new functions were added to the TPU3.

powerful feature permits synchronization of multiple TPU channels. For example, if one channel calculates a value required by another channel, a signal can be passed to the second channel indicating that the value has been updated, thus, scheduling the

awaiting channel. When programming the LINK register, remember that it is located in bits 4–7.

Register DEC (Decrementor) is a special-purpose register that implements a unique solution for flow control. It can be used to repeat an instruction a given number of times or to return from a subroutine after a given number of instructions. Repeating an instruction is useful because the TPU can only shift 1 bit per instruction. The decrementor could be loaded and the shift instruction repeated to allow multiple variable bit shifts using fewer instructions. Employing the decrementor to return from a subroutine is commonly used to send a link to a block of channels (see Listing 1).

The starting channel is loaded into Register P while the decrementor determines the number of subsequent channels to be accessed. After that step, the LINK register sends the links to the channels.

Register CHAN_REG (Channel Register) is a special-purpose read/write register that tells you which channel's hardware is being used. You change to another timer channel's hardware with this register. So, a function running on one channel can control multiple TPU pins. Remem-

Listing 1—This example uses the DEC_RETURN feature. The first half shows the sample code for calling the LINK_CHAN subroutine underneath.

```

au p_high := 1. (* link to channels 1-9 *)
au dec := 8.

call Link_chan, flush;
dec_return.

end.

(*****
(* PROCEDURE : Link_chan *)
(* CALLED BY : Various functions. *)
(* ACTION : link up to 8 channels *)
(* PARAMETERS & REGISTERS : *)
(* p_high - start channel to be linked *)
(* dec - number of channels to be linked *)
(*****
Link_chan :
au link := p_high + $00.
au link := p_high + #$10.
au link := p_high + #$20.
au link := p_high + #$30.
au link := p_high + #$40.
au link := p_high + #$50.
au link := p_high + #$60.
au link := p_high + #$70.

```


ber that the CHAN_REG register is located in bits 4–7. The CHAN_REG and DEC registers can be concatenated to form the 8-bit register CHAN_DEC. The latter allows both CHAN_REG and DEC to be accessed in one instruction.

Now that you understand the registers, I'll move on to programming style and the TPU entry table. A key concept in getting to know the TPU is considering the functions as a set of states with an associated entry table. An entry table is similar to an interrupt or exception table on a standard microcontroller in that it tells the TPU where to go when a service request is made.

There are four events that can cause a service request. The CPU can initiate an HSR (Host Service Request), another channel can make a request by issuing an LSR link (Link Service Request), and the channel hardware can detect a capture on the input pin or a match on the output pin (M/TSR, Match/Transition Service Request).

When you start writing a new TPU function, begin with a state transition diagram. This may seem formal, but it is the best way to efficiently map the states to the TPU architecture. The best way to make fast, efficient TPU functions is to have the states mapped directly to TPU entry points. However, it is not always possible to do this in cases where, for example, there are more states than entry points. The entry points are defined in the TPU entry table exhibited in

Listing 2—There are 16 entry points that must be defined for every TPU function. This can be done in a minimum of five states.

```
(* This is for entry points 0 & 1 *)
%entry start_address *; enable_match;
cond hsr1=0, hsr0=1, lsr=x, m/tsr=x, pin=x, flag0=x.
{insert state code here}

(* This is for entry point 2 *)
%entry start_address *; enable_match;
cond hsr1=1, hsr0=0, lsr=x, m/tsr=x, pin=x, flag0=x.
{insert state code here}

(* This is for entry point 3 *)
%entry start_address *; enable_match;
cond hsr1=1, hsr0=1, lsr=x, m/tsr=x, pin=x, flag0=x.
{insert state code here}

(* This is for entry points 4-7 *)
%entry start_address *; enable_match;
cond hsr1=0, hsr0=0, lsr=0, m/tsr=1, pin=x, flag0=x.
{insert state code here}

(* This is for entry points 8-15 *)
%entry start_address *; enable_match;
cond hsr1=0, hsr0=0, lsr=1, m/tsr=x, pin=x, flag0=x.
{insert state code here}
```

Table 2. Every function has its own entry table that points to unique states for that function.

TPU INSTRUCTION FORMATS

Basically, the TPU has five instructions. But, these are instruction formats rather than actual instructions. Each format allows you to execute several instructions in parallel. This is the key difference between normal microprocessor instructions and the microcode used by the TPU. When I explain assembler syntax later, how to achieve parallel operations will

become obvious. Table 3 states an overview of the instruction formats and parallel actions available.

The parallel operations performed by each instruction format were chosen to minimize the code. Because every TPU instruction is executed in two clock cycles, reducing code size increases execution speed. The table showing the instruction formats can seem daunting because of the long list of actions that can be performed in parallel. To ease understanding, I'll explain the actions in greater detail.

Format 1 performs full arithmetic operations including addition, subtraction, and shift by 1 bit. The addition and subtraction can be performed using constants 0, 1, \$8000, or \$FFFF. The parameter RAM access allows you to read or write to any parameter RAM location. Parameter RAM is the memory shared by the TPU and CPU. It passes variables between the TPU and CPU. The former also uses it for local variable storage.

Table 2—The first four entries are states initiated by the host. The remaining 12 entries are states executed because of conditions on the channel.

Entry points	Service requests			Channel conditions	
	Host request (HSR)	Link request (LSR)	Match/transition request (M/TSR)	Pin state	Software flag 0
0	01	X	X	0	X
1	01	X	X	1	X
2	10	X	X	X	X
3	11	X	X	X	X
4	00	0	1	0	0
5	00	0	1	0	1
6	00	0	1	1	0
7	00	0	1	1	1
8	00	1	0	0	0
9	00	1	0	0	1
10	00	1	0	1	0
11	00	1	0	1	1
12	00	1	1	0	0
13	00	1	1	0	1
14	00	1	1	1	0
15	00	1	1	1	1

Format	Format overview	Actions executable in parallel
1	Most commonly used format. It allows an arithmetic operation and a parameter RAM access to occur in parallel.	Full arithmetic operations Parameter RAM access End the current state
2	The only format that can write to the output channel hardware. All output functions must therefore use this format to set up matches on an output channel.	Arithmetic operations (limited shifting and cannot latch condition codes) Negate latches Channel hardware control Software flags Interrupt Write MER End the current state
3	The only format that allows service requests to be enabled by the channel hardware for execution in the TPU.	Conditional branch Software flags Channel hardware control Time base selection Channel hardware service enable
4	Allows modification of software flags and access of parameter RAM.	Jump to subroutine Software flags Parameter RAM access Negate link latch End the current state
5	Allows an arithmetic operation with an 8-bit immediate data value.	Arithmetic operation with immediate data Software flags Interrupt Negate link latch End the current state

Table 3—The TPU Format table shows the five instruction formats available and the actions they can perform in parallel.

End the current state, as the name suggests, allows the termination of the current state. It also can enable the operation of the decrementor.

As part of Format 2, each TPU channel has three negate latches that control its operation. The latches, used to determine the required entry in the entry point table, are Link Service Latch (LSL), Match Recognition Latch (MRL) and Transition Detect Latch (TDL). When a latch is set, it indicates that a service request has been made. After the microcode services the event, it must negate the latch to show that the service request is no longer required.

If using an input channel, channel hardware control defines which edge the hardware should detect. If using an output channel, it sets the state of an output pin and defines which state should be driven after a match occurs.

Software flags allow you to set or clear the two software flags in each channel. Interrupt allows the TPU to request an interrupt from the host CPU and write MER copies the value from the ERT register to the channel hardware during an output function.

Here are the parts of Format 3. As implied, conditional branch allows a conditional branch to be performed.

While executing one instruction, the TPU loads the next instruction. If the first instruction was a branch, the

In Format 5, the arithmetic operation with immediate data action performs addition, subtraction, as well as

prefetch instruction normally would not be used, causing the TPU to stall. An interesting feature of the TPU is that you control the flushing of the prefetched instruction and can prevent stalling.

Timebase selection allows the definition of the hardware channel as an input or output. It allows you to define which timebase the hardware channel will use. Channel hardware service enables the two signals from the channel hardware, MRL and TDL, to request a service.

In Format 4, jump to subroutine actions allows an absolute jump to subroutine. As noted, a flush can be performed for both of these operations.

shift by 1 bit. The addition and subtraction can be performed with an 8-bit immediate value.

TPU HARDWARE

One of the reasons why programming the TPU seems difficult is that you need to fully understand the TPU hardware to program efficiently. This can be true of all microprocessors, especially for the TPU, because of the amount of control the microcode allows you to have over the hardware. So now, let's take a look at the five main areas shown in the block diagram in Figure 2.

The first area that needs to be explained is the microengine. This is made up of two blocks, the execution unit and the control store. The control store is the program memory for the TPU. It supplies instructions to the TPU from either the TPU's internal ROM or from on-chip emulation SRAM. The execution unit is the TPU's equivalent of the CPU in a microprocessor. It is a simple unit that executes an instruction every

two system clocks. It uses 32-bit fixed-length instructions and has a single instruction prefetch queue.

The second area to consider is the timer channel, which is shown in Figure 3. All 16 timer channels are identical and each is associated with its own pin. A unique feature of the TPU, which is extremely powerful, is that each channel uses a greater than or equal to comparator. Normally, timers only use an equal to comparator, which can cause problems. If you are trying to schedule an event close to the current time, you may miss the event and have to wait until the timebase rolls over. This is not a problem with a greater than or equal to comparator (any event missed will happen as soon as possible).

Programmers must adapt to referring to events as past or future. To help, it is convenient to use a circle that represents the full range of the timebase. You can check this out in Figure 4. With the comparator, the future is defined as the period from current time to current time plus

Listing 3—This is what a top-level *include* file should look like. It is good programming practice to list the version and size of each function in the comments.

```
(*****
(* TPUMSKGC.ASC *)
(* Master source file for second standard TPU mask (MOTION
CONTROL). *)
(*****
%org 0.

(* Standard exits to save space in individual functions *)
End_of_phase:      end.
End_of_link:      chan neg_lsl;
                  end.

(*Nickname  Rev  Size *)
%include 'pta.uc' ;function = $F. (* PTA      1.1   63 *)
%include 'qom.uc' ;function = $E. (* QOM      1.1   49 *)
%include 'tsm.uc' ;function = $D. (* TSM      1.1  105 *)
%include 'fqm.uc' ;function = $C. (* FQM      1.1   20 *)
%include 'uart.uc' ;function = $B. (* UART     1.1   67 *)
%include 'nitc.uc' ;function = $A. (* NITC     1.1  35+ *)
%include 'comm.uc' ;function = 9. (* COMM     1.2   50 *)
%include 'halld.uc' ;function = 8. (* HALLD    1.1   30 *)
%include 'mcpwm.uc' ;function = 7. (* MCPWM    1.1   38 *)
%include 'fqd.uc' ;function = 6. (* FQD      1.1   46 *)
(* standard exits 2 *)
(* ===== *)
(* TOTAL          505 *)

(* + = NITC function uses the LINKCHAN subroutine. The size
of 35 includes the LINKCHAN subroutine.*)
```

\$8000 counts. Similarly, the past is defined as current time to current time minus \$7FFF.

The only disadvantage of using a greater than or equal to comparator is that it reduces the size of your time-base by 1 bit. On the TPU, the time-base is effectively 15 bits. You can only schedule events \$8000 counts from the current time, not \$FFFF like with an equal only comparator.

The capture signal blocks a match signal with the Timer Channel (see Figure 3). Because there's one result register for each channel, only one value can be stored per channel. The capture block ensures that if a match occurs after a capture, the match value does not overwrite the capture value. The match value is written by the program, so it doesn't have to be stored in the channel's result register. The capture value, however, comes from an external event and would be lost if the register is overwritten.

The next area is the scheduler. This block is responsible for scheduling each of the channels and is, in effect, an RTOS. There are three priority levels associated with the scheduler,

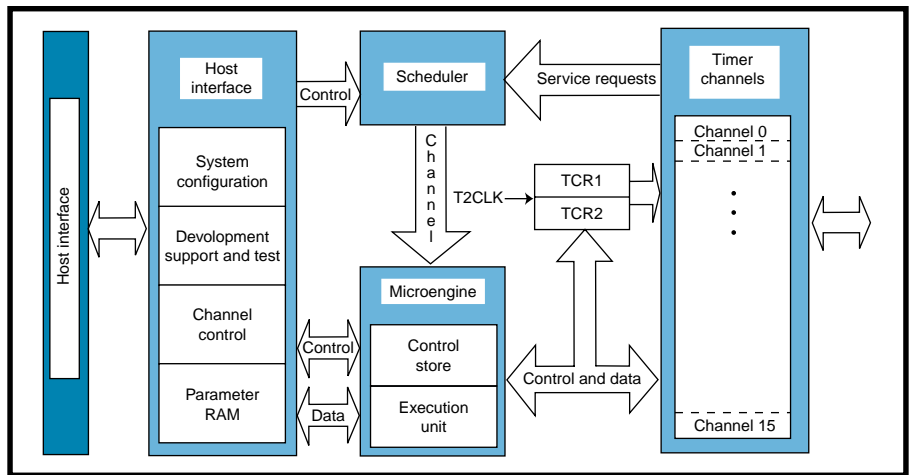


Figure 2—The TPU block diagram shows the five main blocks of the TPU.

and channels are scheduled in a round-robin scheme to guarantee each gets serviced. Each channel is assigned a priority level—high, medium, or low (H, M, or L)—which is scheduled H-M-H-L-H-M-H. The scheduler uses cooperative multitasking whereby each channel must indicate the end of its operation allowing another channel to be scheduled.

The host interface is split into four modules. These modules include system configuration, development

support and test, channel control, and parameter RAM. The system configuration module configures everything that affects the TPU, except the channels. The main things affected by this module are the timebases and how the TPU sends interrupt requests to the host CPU.

The development support and test module provides support for developing and debugging TPU microcode. It allows the TPU to be put into emulation mode, which lets microcode be run from emulation RAM. The channel control module configures and controls the operation of each channel. It allows the configuration of each channel by defining the function to be run and the channel priority. It gives you control of the channel by issuing host service requests (HSR) and setting host sequence registers (HSQR).

A host service request is a request from the host CPU telling a TPU channel to perform a required task. There are two host service request bits for each channel (see Table 4), therefore the CPU can request three different types of service.

There also are two host sequence bits per channel. These are flags written by the CPU and read by the TPU. The parameter RAM has two purposes, it allows the host CPU to communicate with the TPU and it is where the TPU keeps its local variables. Parameter RAM is made up of 100 16-bit locations. Each channel on the TPU has six parameter RAM locations associated with it (except channels 14 and 15, which have eight).

Listing 4—Here's a simple TPU microcode example that shows the addition of two numbers and the storing of their result in parameter RAM.

```

%macro AA 'prm0'.    (*A is a register so it can't be a
                    macro, AA is used*)

%macro B 'prm1'.
%macro C 'prm2'.

(*****
(* ENTRY name:  EX1_ADD          *)
(*              *)
(* STATE(S) ENTERED:  S0        *)
(*              *)
(* PRELOAD PARAMETER :  AA      *)
(*              *)
(* ENTER WHEN :  hsr1=1, hsr0=1. *)
(*              *)
(* ACTION :  Add parameter AA & B putting
            the result in C. *)
(*****
%entry start_address *; disable_match;
cond hsr1=1, hsr0=1, lsr=x, m/tsr=x, pin=x, flag0=x;
ram p <- @AA.

    ram diob <- @B.

    au p := p + diob;
    ram p -> @C;
    end.

```


Listing 5—The TPU microcode for a function to produce a PWM output waveform is listed here.

```

%macro HIGHTIME          'prm0'.
%macro PERIOD            'prm1'.
%macro NEXT_RISING_EDGE 'prm2'.
(*****
(* ENTRY name:  EX2_INIT          *)
(*              *)
(* STATE(S) ENTERED: S0          *)
(*              *)
(* ACTION :  configure channel    *)
(* calculate next rising edge time and store in parameter *)
(* calculate next falling edge time and setup pin          *)
(*****)
%entry start_address *; disable_match;
cond hsr1=1, hsr0=1, lsr=x, m/tsr=x, pin=x, flag0=x;
ram p <- @PERIOD.

chan tbs := out_m1_c1,
      (*configure pin for output using TCR1*)
      pin := high,          (*initialize pin to high*)
      pac := low,           (*drive pin low at next match*)
      enable_mtsr.         (*enable match service requests*)

ram diob <- @HIGHTIME.

au p := ert + p;           (*calculate next rising edge*)
ram p -> @NEXT_RISING_EDGE. (*store value*)

au ert:= tcr1+diob;        (*calculate next falling edge*)
chan write_mer,            (*setup next edge time*)
      neg_mr1, neg_tdl, neg_lsl; (*negate all latches *)
end.
(*****
(* ENTRY name:  EX2_HIGH          *)
(*              *)
(* STATE(S) ENTERED: S1          *)
(*              *)
(* ACTION :  calculate next rising edge time and store in parameter *)
(* calculate next falling edge time and setup pin          *)
(*****)
%entry start_address *; disable_match;
cond hsr1=0, hsr0=0, lsr=x, m/tsr=1, pin=1, flag0=x;
ram p <- @PERIOD.

au p := ert + p;           (*calculate next rising edge*)
ram p -> @NEXT_RISING_EDGE. (*store value*)

ram diob <- @HIGHTIME.

au ert:= ert+diob;        (*calculate next falling edge*)
chan write_mer,            (*setup next edge time*)
      pac := low,           (*next edge drive pin low*)
      neg_mr1, neg_tdl, neg_lsl; (*negate all latches *)
end.
(*****
(* ENTRY name:  EX2_LOW          *)
(*              *)
(* STATE(S) ENTERED: S2          *)
(*              *)
(* ACTION :  calculate next rising edge time and
store in parameter *)
(* calculate next falling edge time and setup pin          *)
(*****)
%entry start_address *; disable_match;
cond hsr1=0, hsr0=0, lsr=x, m/tsr=1, pin=0, flag0=x;
ram p <- @NEXT_RISING_EDGE.

au ert:= p;               (*load next falling edge*)
chan write_mer,            (*setup next edge time*)
      pac := high,          (*next edge drive pin high*)
      neg_mr1, neg_tdl, neg_lsl; (*negate all latches *)
end.

```

Each channel uses its dedicated RAM area to store the local variables of the function running on it. The function running on each channel can access all parameter RAM, enabling exchange of data among channels.

The goal of writing TPU functions is to make them able to run on any channel. Because the functions running on the TPU channels must coordi-

nate parameter RAM usage, each channel should write only to its own dedicated area. The function running on each channel, however, can access all of the parameter RAM.

The next parts of the block diagram are the timebases. The TPU has two 16-bit timebases called TCR1 and TCR2. TCR1 is driven by the system clock and can be one quarter as fast as

Listing 6—Here's the TPU microcode for a function to measure the high time of an input signal.

```
%macro RISING_EDGE 'prm0'.
%macro HIGHTIME1 'prm1'.

(*****)
(* ENTRY name:  EX3_INIT                               *)
(*                                                     *)
(* STATE(S) ENTERED:  S0                               *)
(*                                                     *)
(* ACTION :  Configure channel.                         *)
(*****)
%entry start_address *; disable_match;
cond hsr1=1, hsr0=1, lsr=x, m/tsr=x, pin=x, flag0=x.

    chan  tbs := in_m1_c1, (* configure pin for input using TCR1 *)
          pac := low_high, (* wait for rising edge *)
          enable_mtsr. (* enable capture requests *)

    chan  neg_mr1, neg_td1, neg_lsl; (* negate all latches *)
          end.

(*****)
(* ENTRY name:  EX3_START                               *)
(*                                                     *)
(* STATE(S) ENTERED:  S1                               *)
(*                                                     *)
(* ACTION :  Store rising edge time in parameter RAM.   *)
(*           Configure channel to wait for falling edge. *)
(*****)
%entry start_address *; disable_match;
cond hsr1=0, hsr0=0, lsr=x, m/tsr=1, pin=1, flag0=x.

    au    p := ert; (*store rising edge time in parameter RAM*)
    ram   p -> @RISING_EDGE.

    chan  pac := high_low; (*configure channel to detect falling edge*)
          neg_td1; (*negate capture latch*)
          end.

(*****)
(* ENTRY name:  EX3_END                               *)
(*                                                     *)
(* STATE(S) ENTERED:  S2                               *)
(*                                                     *)
(* ACTION :  Get falling edge time.                    *)
(*           Calculate high time and store in parameter RAM. *)
(*           Configure channel to wait for rising edge. *)
(*           Request interrupt to host CPU.             *)
(*****)
%entry start_address *; disable_match;
cond hsr1=0, hsr0=0, lsr=x, m/tsr=1, pin=0, flag0=x;
ram p <- @RISING_EDGE.

    au    diob := ert - p; (*get falling edge time & calc. high time*)
    ram   diob -> @HIGHTIME1. (*store high time in parameter RAM*)

    chan  pac:= low_high, (*configure channel to detect falling edge*)
          neg_td1; (*negate capture latch*)
          chan cir; (*interrupt host CPU*)
          end.
```

the system clock. TCR2 can be derived from the system clock, clocked by an external source, or operate in gated mode.

ASSEMBLER SYNTAX

The TPU assembler syntax is a mixture of normal microcontroller syntax and Pascal. The difference in writing TPU programs is that several instructions assemble as one microinstruction:

```
au p := a + diob;
ram p-> prml;
end.
```

A semicolon separates instructions and a period signifies the end of a microinstruction. Another difference is that each instruction starts with an identifier to let you know which part of the TPU hardware the instruction uses.

To support the different versions of the TPU and configure functions, the TPU assembler uses commands known as assembler directives. These directives are like instructions, but do not produce code. Three of the six assembler directives used by the TPU are required for TPU programming.

The TPU supports a simple form of macro substitution with the `%macro` directive:

```
%macro HIGH_TIME 'prm0'.
    %macro A 'prm1'.
    %macro PERIOD 'prm3'.
```

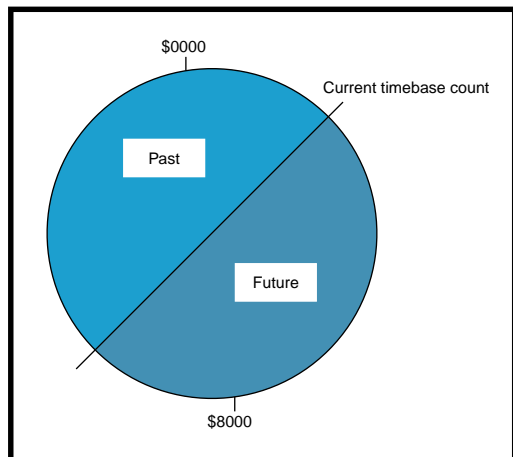


Figure 4—This is a graphical representation of the greater than or equal to comparator. The circle represents one complete cycle of the timebase.

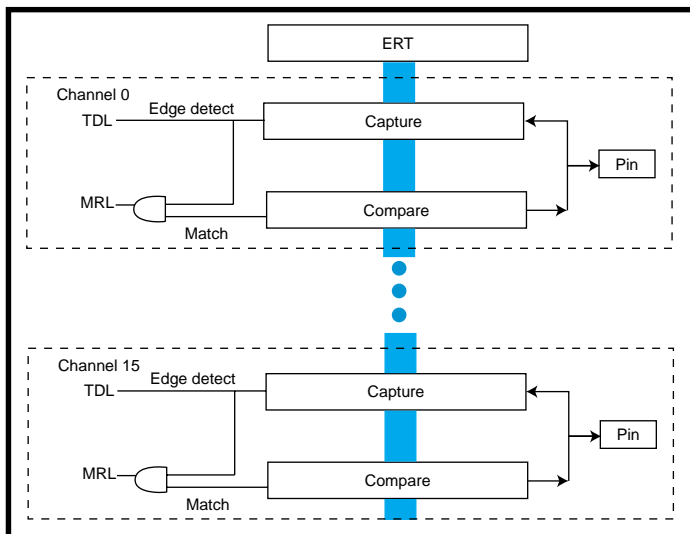


Figure 3—Two of the 16 channels are shown here.

This directive normally used to give the parameter RAM values a meaningful name.

The `%entry` directive is the most complicated and most important. This directive defines the entry table for a given function. All entries for a function must be defined using the `%entry` directive to prevent an error during assembly (see Listing 2).

Finally, the `%include` directive enables the inclusion of source code from another file. This directive usually creates a complete set of TPU functions from several single function files. Normally, there will be one top-level include file that contains every `%include` directive.

In addition, the top-level include file contains two microinstructions that are commonly referred to as the standard end instructions. To save space, the standard end functions are only put in memory once, and then referenced by all functions; this saves inclusion in every function. The top-level include file for the standard TPUG mask set is shown in Listing 3.

TPU TOOLS

No programmable processor is useful without tools to support it. The most important tool is the assembler. For the TPU, there are three choices. The first two are Motorola's TPUMASM V. 3.33, available

free on the Motorola web site, and TPU-MASM V. 4.04 sold by Motorola and TAS, available as source code under the GNU license. Neither supports the TPU2 nor TPU3. But, future versions of TAS are planned to support both. And, a previous compiled version of TAS for Windows 95/98 and WindowsNT is available on the 'Net.

Two debuggers are available for the TPU. An unsupported, DOS-based debugger called

TPUBUG is free from the Motorola web site. And, as part of the Lauterbach tools, there is debug support for the TPU3 on the MPC555.

The simulator is the last tool. There is one simulator available for the TPU from Ashware. This product provides full development support for all TPU versions.

WRITING FUNCTIONS

You made it this far, now you can get to the fun part of actually writing functions. I wrote three functions to give you a taste of TPU programming, including a simple function, an output function, and an input function.

The first function you'll write will simply add two numbers from parameter RAM and put the result in a third location ($A + B = C$). This function is shown in Listing 4. Notice that in the entry directive, one parameter load operation can be specified. In this example, the P register is loaded with the value A. The first instruction loads the B value into the DIOB register and the second instruction performs the addition, writes the result to the C parameter, and ends the state. Hence, the second instruction has three instructions within one microinstruction.

All three can be done in one microinstruction because the TPU does the addition first, the write to parameter RAM second, and the end instruction last. The TPU follows this combination of instructions in this order no

Host Service Request bits (HSR)	Resulting action
00	No host service request
01	Host service request 1
10	Host service request 2
11	Host service request 3

Table 4—This shows the decoding of the Host Service Request bits. The host CPU can request three different service requests.


matter how they are written. For clarity, it's helpful to write the instructions in the order they are executed by the TPU.

The next function produces a PWM output waveform (see Listing 5). This is made up of three states—initialization, PWM_high, and PWM_low. The initialization state configures the channel and sets up the first edge. The PWM_high state calculates the falling edge time, the next rising edge time, and sets up the next falling edge. The PWM_low state uses the previously calculated value and sets up the next rising edge. As stated, with timer channels, you can only schedule events \$8000 in the future, so values used must be less than \$8000.

Listing 6 shows the input function that measures the high time of an input signal. Initialization, start, and end are the function's states. The initialization state configures the channel and sets up the input edge detection. The start state is entered when a rising edge is detected. It saves the time of the edge and configures the channel to look for the falling edge. The end state calculates the length of the high time, writes it to parameter RAM, and sends an interrupt request to the host CPU.

CHALLENGE

As you expected, the TPU is complicated, and powerful when mastered. I hope this article interested you enough to write more complex and useful functions. My examples are basic, but you may try making the edges configurable using the host sequence bits, adding noise immunity, making the TPU handle error conditions, or making the functions work with values greater than \$8000.

It is your challenge now. Happy programming! 

Jeff Loeliger is a senior staff engineer in the Advanced Vehicle Systems Division at Motorola SPS, Europe. With more than twelve years of experience in microcontrollers, he has become a recognized expert on the Motorola TPU. You may reach him at jeff.loeliger@motorola.com.

SOFTWARE

The source code for the example functions is available on the *Circuit Cellar* web site.

REFERENCES

- J. DiBartolomeo, "TPU: A Coprocessor for Timing Function," *Circuit Cellar* 102–105, 1999.
- J. Loeliger, "RC Servo Control via TPU," *Circuit Cellar Online*, December, 1999.
- "Time Processor Unit Macro Assembler Reference Manual," Motorola, Inc., TPUMASMREF/D2, 1994.
- "Time Processor Unit," Motorola, Inc., TPURM/AD, 1990.
- M. Bannoura and A. Dyson, TPU Microcode for Beginners, AMT Publishing, Australia, 1998.
- R. Soja, "Inside Motorola's TPU," *Doctor Dobbs Journal*, December, 1996.

SOURCES

TPU tools

Motorola, Inc.
(800) 521-6274
(512) 328-2268
Fax: (512) 895-4465
www.mot-sps.com

TAS

TPU Assembler
www.loeliger.freesever.co.uk

TPU debugger

Lauterbach, Inc.
(508) 303-6812
Fax: (508) 303-6813
www.lauterbach.com

TPU simulator

Ashware, Inc.
(503) 553-0271
Fax: (508) 553-0547
www.ashware.com

FEATURE ARTICLE

Duane Perkins

A PIC17C44-Based Computer

According to the market, single-board computers and PIC microcontrollers don't mix. But, it would be nice, wouldn't it? If you agree, tune in, because Duane found the right mixture so we can execute from external RAM.



Single-board computers that use PIC microcontrollers are not readily available, partly because all except the top-of-the-line PIC17Cxx microcontrollers do not allow execution from external memory. This precludes execution from RAM, which means that a program cannot be downloaded and executed. However, the PIC17Cxx microcontrollers can be configured for a microprocessor or extended microcontroller mode, much like the Motorola 68HC11, which greatly facilitates program development. In this article, I'll explain how to build a PIC17C44-based computer that can be programmed to execute from external RAM. I call it the SBC17C44.

Although the SBC17C44 is designed around the PIC17C44, any of the PIC17Cxx microcontrollers with 8-KB or smaller program memory can be used. There are two boards, the motherboard and daughterboard. The motherboard includes the external memory, address latch, and device selection decoder. The daughterboard provides the device interfaces, including RS-232/422/485 serial I/O level conversion, an LCD, a keypad, and a piezo buzzer. It also includes the power supply. The boards are large with wide

traces, making it possible to etch the boards at home using the simplest possible tools and materials.

THE MOTHERBOARD

The motherboard features a 16-bit address bus, 16-bit data bus, 8 KB of external program memory (ROM or RAM), and 8 KB of data memory (RAM) (see Figure 1). A 34-pin header provides access to the low-order 8 bits of the address bus and data bus, 10 I/O pins, five chip-select lines for external devices, and the OE and WR lines. A 6-pin header provides for connection to a serial I/O level converter.

The three high-order bits of the address bus are used to select the memory chips or external devices. 2000:3FFF selects the 16-bit program memory; 4000:5FFF selects the 8-bit data memory. 6000:7FFF, 8000:9FFF, A000:BFFF, C000:DFFF, and E000:FFFF activate CS3, CS4, CS5, CS6, and CS7, respectively.

J1 provides power from a 5-V supply. You can connect a reset button to the third and fourth terminals. J2 is a 34-pin header that connects the motherboard to a daughterboard. It provides connections to the buses, I/O lines, and control lines as discussed above. J3 is a 6-pin header used to connect to J5 or J7 on the daughterboard for serial I/O level conversion.

THE CLOCK OSCILLATOR

A crystal or ceramic resonator can clock U1 or a pulse train can be applied to OSC1 (pin 14 of U1). A 3-pin SIP socket is provided for Y1. If a ceramic resonator with integral capacitors is used, C10 and C11 should not be used.

The frequency can be 33 MHz or lower, however it must not be above 25 MHz if EPROM is used for external program memory. Use 70-ns SRAM for higher frequencies.

PORT ASSIGNMENTS

The PIC17C44 has five ports with a total of 33 I/O pins. Note that a pin is specified herein as RX<N>, where X is the port designation letter and N is the bit designation of the pin. RX<M:N> designates a range of pins, where M is the high bit and N is the low bit. Most pins can be configured dynamically

under program control as inputs or outputs. Many pins have alternative uses that are invoked by configuration bits in internal memory or in various control registers.

Port A is 6 bits (2 bits are required for serial I/O). Port B is 8 bits—each is available for input or output. Port C has 8 bits used for the low-order address/data bus. Port D has 8 bits used for the high-order address/data bus. And lastly, Port E has 3 bits (ALE, OE, and WR).

RA<1:0> can be interrupt or data inputs. RA<3:2> are Schmitt Trigger inputs or high-current, open-drain outputs. RA<4> is the serial TX line and RA<5> is the serial RX line; these can be general-purpose inputs if serial I/O is not used. RB<7:0> can be used in accordance with the Microchip specs.

In microprocessor or extended microcontroller modes, RC<7:0> is used as the low-order address/data bus and RD<7:0> for the high-order address/data bus. These multiplexed pins drive the address latch when a TABLRD or TABLWT instruction is executed. These are available for input or output.

In these modes, RE<2:0> is used for the ALE, OE, and WR lines. ALE (active high) activates the address latch. OE (active low) signals the memory chips and external devices to drive the data bus. And, WR (active low) signals the memory chips or external devices to write the data on the data bus (read and put data on the bus when high). And again, in microcontroller mode, these are available for input or output.

Lastly, U2 decodes RA<15:13> and pulls one of its outputs low.

MEMORY

Each of the three memory sockets will accept a 28-pin, 8-KB memory chip.

These can be static RAM (6264) or EPROM (2764). An address in the range 0000:1FFF is the range of the PIC's internal program memory. U5 and U6 are selected by an address in the range 2000:3FFF. And, the final memory socket, U7, is selected by an address in the range 4000:5FFF. Note that using RAM chips in all three sockets allows a program to be downloaded and executed in the extended microcontroller mode.

The SBC17C44 operates in the extended microcontroller mode. In this mode, the program code can reside in both internal and external memory.

The internal program memory is ROM. During program development, it is convenient to have a program loader

and commonly used functions in internal memory. The application program can be downloaded to RAM and can call the commonly used functions, avoiding the need to assemble or link them every time the application program is changed.

The PIC17C44 has a 16-bit instruction code size and 16-bit buses for fetching instructions. To fetch instructions from external memory requires a 16-bit address latch that's implemented on the motherboard by U2 and U3. RE0/ (ALE) is the address latch enable line, which is pulsed low during every instruction cycle.

If an instruction is fetched from a location in the range 2000:3FFF, the latched address selects a word in U5 and U6. U2 decodes bits 15:13 and pulls V1 low, thus enabling U5 and U6 via the CE pins. /OE is low, thus enabling U4 and U5 via the OE pins to put the instruction word on the AD bus.

And, note that /WR remains high so that the R/W pins are high and the memory is read rather than written. These control signals from Port E are present on every instruction cycle irrespective of the address.

TABLARD AND TABLWR

The TABLRD and TABLWR instructions allow external memory to be read or written. The loader in internal memory writes downloaded program code to external memory. Data memory in U7 can be read or written. These instructions use the address latch to determine the address of the instruction word or data byte by latching an address held in a 16-bit SFR pair called the table pointer (TABLPTRH and TBLPTRL).

The instruction, or data, is read into or written from another SFR pair called the table latch (TABLATH and TABLATL). Sixteen-bit data can be read from internal or

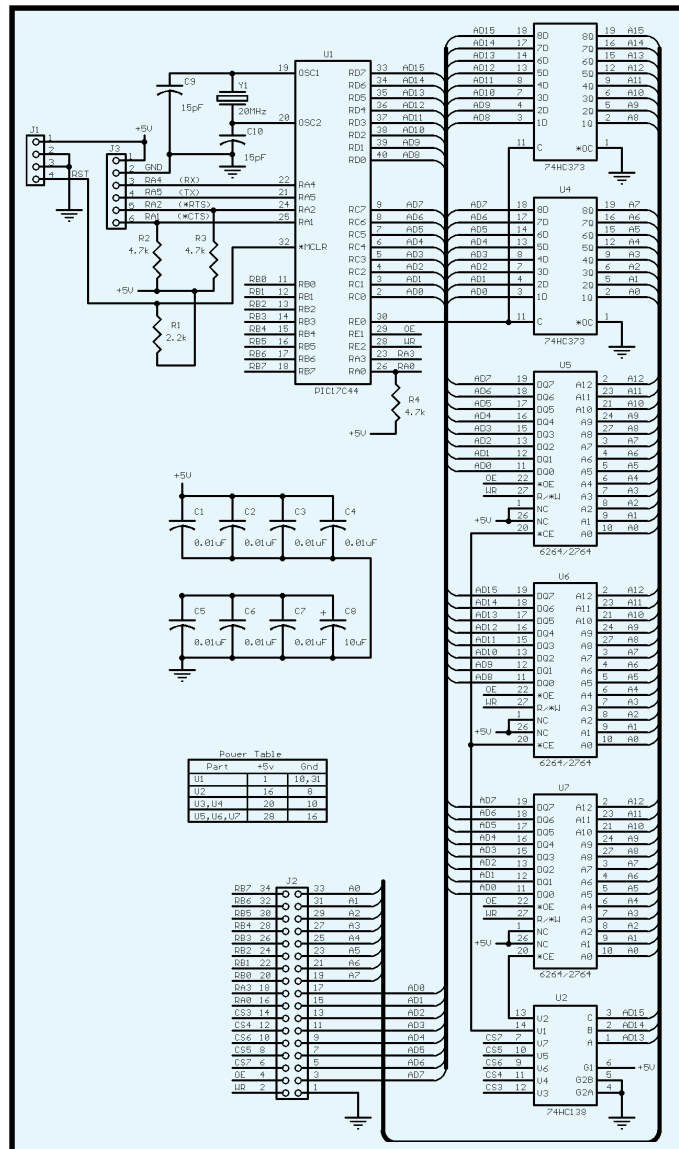


Figure 1—Take a look at the motherboard schematic.

6000-7FFF	CS3	Available
8000-9FFF	CS4	Available
A000-BFFF	CS5	LCD (lines 1 and 2 of 4 x 40 LCD)
C000-DFFF	CS6	Lines 3 and 4 of 4 x 40 LCD
E000-FFFF	CS7	Keypad or piezo buzzer

Table 1—Here are the device selection address range assignments.

rupt request by the PIC17C44. RA3 is an input used as the RS line for the LCD and also clears the CS7 and WR latches when pulsed low.

One of the CS lines will pulse low when a TABLRD or TABLWR instruction is executed with an address in the range 6000:FFFF. The WR line pulses low when a TABLWR instruction is executed. OE pulses low during every instruction cycle, but is not used. The low-order bytes of the address bus (RA<7:0>) and the address/data bus (RAD<7:0>) are not used.

THE LATCHES

Device selection is accomplished by executing a TABLRD or TABLWR instruction (see Table 1). RA<15:13> are decoded to pull one of the CS lines low for 1 Tcy.

There are three latches for CS5, CS6, and CS7, and one for WR. Selecting CS5 or CS6 clears the CS7 latch. To clear the CS5 and CS6 latches, select CS7. To clear the CS5 and CS6 latches without setting the CS7 latch, hold RA3 low.

external program memory, but should not be written by the application code. It should be used only for constants that are coded in the source program.

THE DAUGHTERBOARD

The daughterboard consists of four independent sections that can be separated for mounting (see Figures 2 and 3). These are the power supply, the RS-232 level converter, the RS-422/485 level converter, and the keypad/LCD/piezo buzzer interface.

J1 pins 1 and 4 provide for power and ground from J13. Pin 2 connects to an open-collector transistor that can drive a piezo buzzer. An active-low interrupt request can connect to pin 3.

J2 is a 34-pin header that connects to the motherboard. J3, a 16-pin header, connects to an LCD. Note that

the pinout is the mirror image of the LCD pinout so that a matching 16-pin header connector can be used on the solder side of the LCD board.

The 9-pin header that connects to a matrix keypad is J4. The pinout probably will not match that of your keypad. J5 and J7 are 6-pin headers to connect one of the level converters to J3 on the motherboard.

Next is J6, a DB-9 connector for RS-232 serial I/O. J8 is a 5-pin header for RS-422/485 serial I/O.

And, J9 selects RS-422 (jumper pins 1 and 2) or RS-485 (jumper pins 2 and 3). Finally, J10 and J11 can be jumpered to bias the line in a mark state when using RS-485.

RB<7:0> are used as the data bus for the LCD and keypad. RA0 is an output that can be used as an active-high inter-

The WR latch is set when any TABLWR instruction is executed. The CS7 latch can be cleared by pulling RA3 low (level activated). The WR latch can be cleared by pulsing RA3 low (edge triggered). In addition, RA3 is also used as the register selection line for the LCD.

CS5 or CS6, which are latched in U2, select the LCD. Only one can be selected at any given time and absolutely must be cleared before the other can be set. U2B selects E0 and U2A selects E1. WR is latched in U3A. RS is the state of the RA3 pin.

When reading, the LCD drives the data on Port B. And when writing, the data must be on Port B. U1 (PIC12C508A) polls for either selection. When either goes high, it enables the LCD. When writing, U1 pulses E0 or E1 high according to the selection latch outputs, then waits for both latch outputs to go low before it resumes polling for either high. When reading, it sets and holds E0 or E1 high until both latch outputs are low, then resumes polling for either high.

The PIC-17C44 should be programmed to delay long enough to allow the PIC-12C508A to execute the instruction

sequence required for the LCD operation before changing the RW latch or the data bus.

The keypad decoder (U4) is selected by CS7, which is latched in U3B. U4 drives the data on Port B. After reading the data, U3B must be cleared by pulsing RA3 low so that the data output lines of U4 return to hi-Z. When CS7 is selected by a TABLRWR instruction, an open-collector transistor activates the piezo buzzer. Before executing either a TABLRD or TABLWR instruction, RA3 must be set high and RB<7:0> must be hi-Z. Pulse RA3 low to deselect CS7. When a key is pressed on the keypad, the DA output of U4 goes high, setting RA0 high. You can program this to invoke an interrupt.

Model	Code	Program memory	Data memory	External program memory
Small	1	≤2 Kb	≤8 Kb	Yes
Compact	2	≤2 Kb	>8 Kb	No
Medium	3	>2 Kb	≤8 Kb	Yes
Large	4	>2 Kb	>8 Kb	No

Table 2—Here are the memory model designations.

THE PIC12C508A

A PIC12C508A is shipped with a calibration constant in ROM at 0x1FF, coded as a MOVLW instruction. Before erasing a new PIC12C508A, read the program memory with a PIC programmer and note the hex value of the low-order byte of the code at 0x1FF.

The PIC12C508A must be configured for an internal 4-MHz RC oscillator. Although the frequency is not exactly 4 MHz, it is close if the calibration constant is stored in OSCCAL.

MAKING THE PC BOARDS

The first step in construction is to make the PC boards. You need a negative for each side of the two boards. You can make your own with Kepro

RF-2024 reversing film or can have them made at a photo shop that makes lithographic negatives. You can download the artwork from the *Circuit Cellar* web site.

Download all mirror.lsr files to a printer (files can't be displayed or printed by Windows). To make your own copies, use transparency film for figure5.lsr-figure9.lsr. If you're not making negatives, print the files on paper.

You will need two-sided photosensitized boards, such as Kepro S2-712G. Registration is critical during exposure. Using a wooden board as a base, place a blank PC board on the base and

one of the negatives on top of the board, emulsion side down, so that the pattern is within the borders of the board. Using a sharp awl, mark the board at one of the two registration hole pads marked with crosses and remove the negative. Then, with a no. 60 bit, drill the board at the mark.

Remove the debris and again place the negative on the board and use a plastic-headed bulletin board tack to hold it in place. Mark the board at the other registration hole pad and the two mounting hole pads at the opposite edge and drill the registration hole. Place the negative on the board and hold it in place with two tacks. Put a clean sheet of glass on top of the negative to hold it in contact with the board, with the edge against the tacks. Expose the board according to the directions for the material used. Then, remove the glass and negative, flip the board over, and repeat the process with the other negative, making sure the emulsion side is down.

Enlarge the two holes to 5/16" and drill 5/16" holes where previously marked at the other two corners. Place 1/4" 6-32 machine screws in the holes and secure them with machine nuts. This allows the developer and etchant

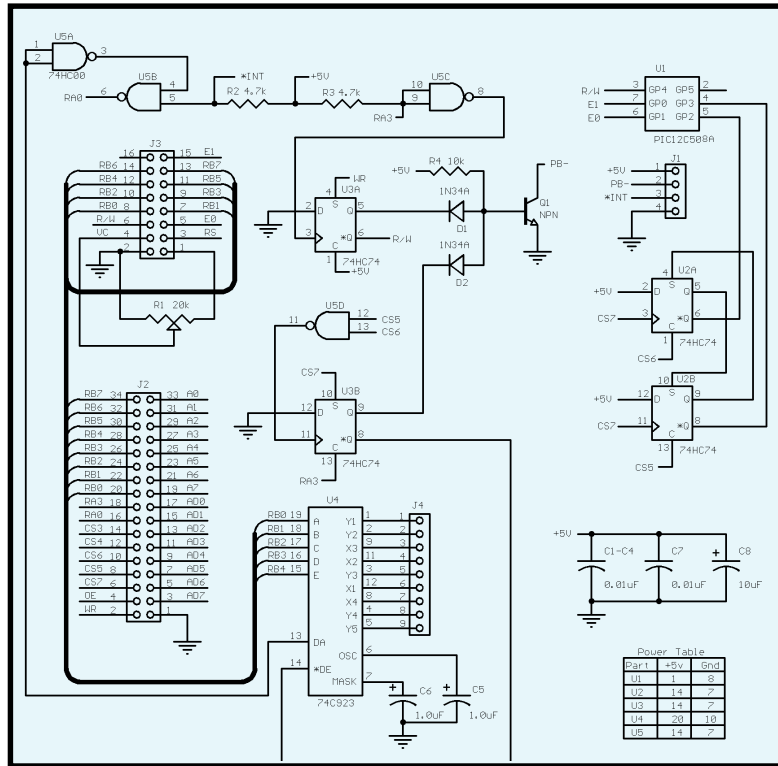


Figure 2—The LCD/keypad/piezo buzzer circuit is shown here.

to flow under the board. Develop, etch, and tin the board, then remove the machine screws. Use a no. 60 bit to drill holes through all pads. If you want to mount the auxiliary boards separately, use a hacksaw to cut the boards apart along the guidelines.

Next, solder the feedthrough conductors. Use 24-gauge bare tinned-copper wire. Be careful not to solder a conductor in a pad intended for a component lead. Feedthroughs always connect traces on both sides of the board, but some pads are intended for component leads. Because none of the feedthroughs are under a component, they can be soldered after the components. However, that approach is more difficult. Be sure to make good solder connections, because poor connections can be difficult to trace.

After that, solder the components starting with the smallest and proceeding according to size, again avoiding bridges. Do not solder ICs directly to the board, instead, use sockets. For development purposes, it is advantageous to use a ZIF socket for U1. I recommend the JDR Microdevices 40-6554-10 socket. Remove the rosin with acetone. Do not immerse the board; use a small brush and let the acetone

drip off the edge. When you're finished, check for solder bridges and unsoldered pads.

TESTING THE BOARDS

When testing, connect power to J12 on the daughterboard and test for 5 V on J13. Check for shorts across pins 1 and 2 of J1 on both boards. Also, connect J13 to J1 on both boards. Connect J2 on both boards with a 34-wire ribbon cable with header connectors.

With no chips in any of the sockets, test for proper voltages as follows. Check for 5 V on the 5-V pins

and on other pins connected to 5 V (see Figures 1-3). Then, connect the positive lead of a voltmeter to 5 V and use the negative lead to check for a ground on the GND pins (see tables) and other pins connected to ground (see figures).

Program a PIC from tryports.hex. Install U1 and Y1 in their sockets on the motherboard, but no other chips. Apply power and check for about 2.5 V on 20. Ports B, C, D, and E should be counting. In addition, check for about 2.5 V on all pins of all sockets where these signals should appear. An oscilloscope should show a pulse train on all these pins, doubling in period with each higher bit of each port.

Next, install the other chips in their sockets on the motherboard. Program a PIC from memory.hex. When power is applied, all bytes of all three memory chips will be written and verified. Port B's bits all should output a square wave at decreasing frequency as probed from low to high order (use a voltmeter).

RA<2> should pulse low at a frequency of 14 Hz with a 20-MHz crystal in Y1. If a failure occurs, the program will freeze and RB<7:0> will output the bits read. And RA<2> will be high if program memory failed or low if data memory failed. Also note that RA<3>

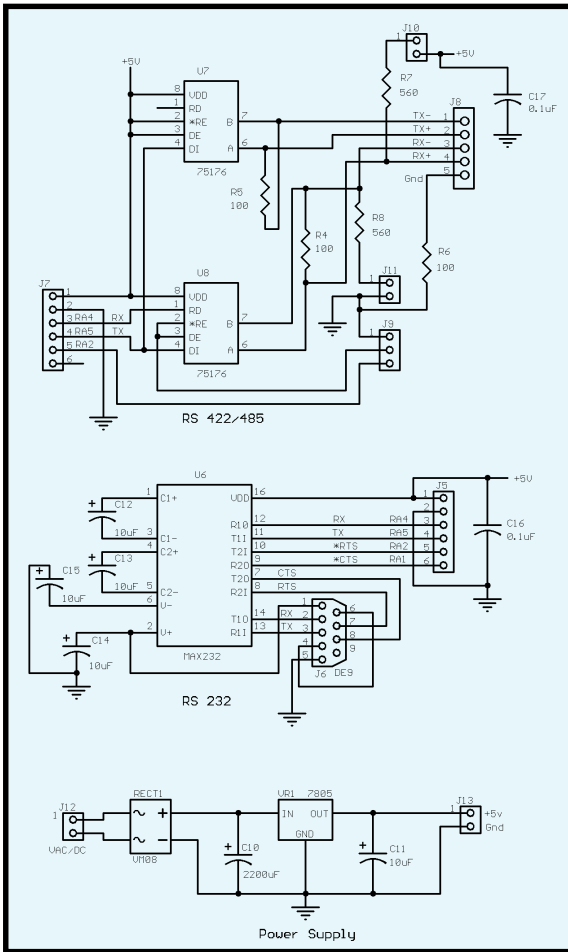


Figure 3—This schematic shows the serial I/O level converter and power supply.

will be pulled to ground if the low-order byte failed. If that doesn't occur, $RA<3>$ will be hi-Z.

Program a PIC from `devices.hex`. The CS lines (pins 6, 8, 10, 12, and 14 of J2) will go low sequentially for one program cycle ($F_{osc}/4$).

You must assemble the program for the PIC12C508A from `aux17C44.asm`, but first change the calibration constant to what you just determined. Program PIC12C508A from `aux17C44.hex` and install it and the other chips in their sockets on the daughterboard. Then, run additional tests using the supplied `.asm` and `.ext` source code files following the next instructions.

THE SOFTWARE

Programs can be assembled for internal or external program memory. A program to run in external memory requires a program in internal memory with code at the reset vector (zero) where execution starts. This code can be as simple as a branch (`call` or `goto`)

to code in external memory, but be sure to set `pccal` to a range of 20–3Fh before the branch instruction.

If any interrupts are to be enabled, appropriate code beginning at the interrupt vector is also required. It is convenient to program a PIC17C44 with `sbc17C44.hex`, which provides a program loader, interrupt service code, and a number of commonly used functions that can be called by the application.

A program assembled for internal memory can include `sbc17C44.int`. This includes initialization code and a few functions. Interrupt code also will be included if service is defined prior to the `#include` line (see Table 2).

The model defaults to small if not defined before `sbc17C44.int` code. RAM at 2000:3FFF can be used for data if the compact or large model is specified.

Program code can reside in external program memory if the small or medium model is specified.

A program coded for external memory can include `sbc17C44.inc`. `intsvc` must be defined as 0 if interrupts are not enabled or with the label of the interrupt service if interrupts are enabled. The label "main" must be included as the entry point of the application code. "Endcode" must precede the end line. See `key2lcd.ext` and `clock.ext` for examples. See `monitor.lsr` for more information. 📄

Duane Perkins is a self-taught engineer who has made computers and electronics his avocation since retiring in 1980. In recent years, he has specialized in PIC microcontrollers. You may reach Duane at dmp Perkins@compuserve.com.

SOFTWARE

The software is available on the *Circuit Cellar* web site.

NOUVEAU PC

Edited by Rick Prescott

SERVER APPLIANCES

The low-profile **Little Dragon** series supports Intel Pentium III processors up to 700 MHz as well as economical Intel Celeron processors. Components include an Intel chipset and an Intel 82559 fast Ethernet controller.

The Little Dragon series initial offering includes 1U and 2U units in bare-bones or full-system configurations. The 1U design is 1.75" high and includes one full-length

PC1 slot and three drive bays. The 2U is 3.5" high and includes two full-length PC1 slots and four drive bays.

The compact design of the server appliance allows for more systems per rack. Its steel case was designed to meet the needs of high-density installations that require quick service access and mounting options typically needed by ISPs and application service providers (ASPS).

The Little Dragon fits open-frame or enclosed cabinet-style racks. It is ideal for applications where high packaging density, performance, easy operation, and quick exchangeability are required.

Evaluation units of the 1U (bare-bones configuration) cost \$399.



ITOX, Inc.
(732) 390-2815
Fax: (732) 390-2817
www.itox.com

SLIM-LINE PANEL PC

The **PPC-102T** is a Panel PC for web-based kiosks, ATMs, patient monitoring, and control panel applications. Intended for systems integrators and OEMs, the system can be bundled with a configured Windows CE image and a Windows CE 2.12 full license version to speed development time for integrating it into other products.

The unit comes with a 10.4" TFT LCD, Pentium MMX, Cyrix or AMD processor, solid-state disk, socket for another DiskOnChip flash memory disk, onboard Ethernet, internal and external hard disk ports, and 32 MB of RAM. All other ports for audio, keyboard, mouse, and serial or parallel peripherals are included. A touchscreen is optional. The unit comes in a 13.5" x 10.4" x 2.4" fireproof industrial plastic case.

With Windows CE installed, the PPC-102T costs approximately \$2000 in OEM quantities.

Advantech Technologies, Inc.
(800) 866-6008
(949) 789-7178
Fax: (949) 789-7179
www.advantech.com/



Ingo Cyliax

Debugging an FPGA Module

Finding the Right Test Case

It's a good thing Ingo enjoys problem solving because seeking out a test condition this time is crucial, not to mention rummaging for old manuals and having a bit of patience along the way. It's all in the name of debugging.



I'm a problem solver at heart.

This fact about myself actually took me a long time to discover. One of the reasons it never occurred to me was because it used to frustrate me to work on things that don't operate properly. What I didn't realize at the time was that this frustration can also be the necessary energy needed to facilitate the process. Also, I'm much more patient these days.

So, let's dig into a problem I've been working on. The hardest thing about it was tracking down a test condition that exhibits the problem. In a sense, finding a good test case is the trick to effective problem solving. It helps narrow the scope of the problem.

REPLICATE AND ANALYZE

The system I'm debugging is an application for a client who's using an FPGA-based I/O board to enhance the real-time performance of their system. I won't go into detail about the actual project, but the problem had to do with not being able to successfully load the FPGA on the I/O module. Of course, this problem appeared to be nondeterministic at first. The board

would sometimes load, but not every time. What's worse, it would work fine in a seemingly identical test system that the client provided for me to help identify the problem.

The client was using a Windows 95-based environment to test the application. Of course, when they sent me the files to download to the FPGA module, it would work fine on my system. Furthermore, when they took the board from one system to another, it would experience no problems. Obviously, there was something peculiar about the actual system that made it fail.

The system was a PC/104-based Pentium module that had to run from a power supply through a PC/104-based DC/DC power converter. The FPGA module would plug into this stack and run a custom design to perform whatever function it needed. Because this was the system they were having trouble with, they sent me one of their spare systems so that I could try to replicate the problem and analyze it.

When the system arrived and I started to hook it up, I discovered a small gotcha. It uses 2.5" disk drive cables (2.5" drives are laptop disk drives). Luckily I had some on hand to install. The test setup uses a Windows95 environment to load the application. After loading a Windows95 environment on a spare 2.5" disk drive and installing their design files on it, I was ready to begin testing.

It worked! Well, the download to the FPGA worked, but their design and related test program did not. But, the diagnostic program that comes with the FPGA modules loaded and ran fine.

One of their engineers told me this wasn't as far as their system goes. It fails when trying to load the FPGA design. I wasn't able to duplicate the problem they were seeing. What now?

They got the download to work when the disk cache was turned off. This is done with the `smartdrv.exe` program/driver under DOS. `smartdrv c` turns the read cache on for drive C and `smartdrv c-` turns the cache off.

I tried various combinations in DOS mode and under Windows and was finally able to replicate this behavior. At first I had the disk cache off and that's why the loads worked reliably.

ROLL UP THE SLEEVES

This was strange indeed. Up until that point, I suspected that the problem might have been related to bus loading and the fact that the CPU module uses 3.3-V signaling on the PC/104

bus. PC/104 uses TTL-based signaling levels, but a 3.3-V signaling level should work with most 5-V TTL or CMOS devices.

This theory of mine was put to rest with a quick look at the datasheets of

the FPGA and PLD used on the FPGA module to make sure that VIL/VIHO thresholds were OK.

Another theory was that there may have been an I/O address conflict. The FPGA module uses 16-bit I/O addressing and a write to a 16-bit decoded address as the signal to the module to reset and reinitiate the FPGAs. Naive I/O modules might use only 12 bits of address decoding (0x000–0x3ff), so there's a small chance that there will be a conflict if the addresses of such an I/O board or resources repeat throughout the 64-KB address space. But because I was able to control its behavior by turning the disk cache on and off, this was probably not the case.

It was time to roll up my sleeves. I needed to get the logic analyzer out and try to capture what was going on with the bus. This would tell me whether there was an I/O conflict or a signaling level issue.

Logic analyzers are amazing machines, but can be tricky to use. In the hands of an experienced engineer they can be a valuable tool. Of course, I still have a lot to learn about using logic analyzers, because there are still options and modes I haven't explored.

If you're unfamiliar with what a logic analyzer does, I'll give you a short description. Think of a logic analyzer as a digital recorder. There are usually several digital inputs. They usually come in a 16-bit set. A "pod" interfaces a set of test probes to a ribbon cable that goes into the logic analyzer. The pod has inputs for the digital signals, as well as ground connections and one or more clock inputs. Each pod usually has about 16 data bits.

The logic analyzer usually uses one of the clocks on the pods to sample the inputs. This way it can sample the data synchronously to the device under test (DUT). Synchronous sampling is important because you need to have a view of the DUT just like other components on the device. In the case of a processor, it will also sample all of its inputs synchronously to a bus clock.

The logic analyzer records the data it samples from the pods into its memory. When the memory is full or an acquisition runs, the analyzer stops. You can display the memory in several

Listing 1—Take a look at the FPGA programming chain.

```
/*
 dumpbits.c - routine to config PF2000 series module
 Author: Ingo Cyliax
 Copyright 1998, 1999, Derivation Systems Inc.

 Created: 1998/06/07
 Last Revision: July 24, 1999 - implement usleep loop
*/
...
#ifdef DOS
    if(!(fp = fopen(file, "rb"))){
#else
    if(!(fp = fopen(file, "r"))){
#endif
        perror(file);
        return(-1);
    }
#endif
    iopl(3);
#endif

again:

#ifdef DEBUG
    fprintf(stderr, "about to send prog\n");
    getchar();
#endif

    outbyte(0, io|0xa800);          /* re-program pulse*/
    /*
        wait for device to clear
    */
    usleep(10000);

#ifdef DEBUG
    fprintf(stderr, "about to send data\n");
    getchar();
#endif

    /* send the data */
    while((c = gets1s9(fp)) != -1){
#ifdef DEBUG
        fprintf(stderr, "%02x", c);
#endif
        while(!(inbyte(io) & 0x80))
            ;
        outbyte(c, io);
    }

#ifdef DEBUG
    fprintf(stderr, "sent data\n");
    getchar();
#endif
    ...
```

different ways. One way is to look at it as a timing diagram. This makes it easy to examine things like bus or communication protocols. Another mode is state analysis. Here the data is displayed as a hex or binary one line at a time. It looks like a listing. State analysis is useful if you're tracing software and want to look at the actual data going back and forth on the bus.

However, because the logic analyzer's memory is fast, it is also small (usually around 4-KB or 8-KB words). This is not enough memory to record something like bus transaction information. At 8 MHz, 4 KB of memory will fill up in 512 μ s.

TRIGGER A START OR STOP

To aid in looking for problems, logic analyzers have sophisticated triggers. A trigger allows you to program a condition based on the inputs that will start or stop data acquisition. For example, you can set up a trigger to look for a write to a certain address or range of addresses. Some analyzers even let you set up sequential conditions. This could be a complex series of events, like when an interrupt and a write to a certain address occurs.

Triggers, like I said, are used to either start or stop an acquisition run. Typically, the trigger is in the middle of the buffer, but it can be programmed to be either at the end or the beginning. The middle is convenient if you want to look at the conditions around a trigger event. The analyzer would collect data and read it into the buffer in a FIFO order. When the trigger occurs, it will read another half buffer of data and, voilà, you have the data leading up to the condition that caused the trigger.

The analyzer I was using also has some conditional timing information. You can specify conditions based on whether or not a glitch has occurred. A glitch is a logic transition that is shorter than the clock rate.

Analyzers can do glitch captures down to a nanosecond time range. Glitches and edges can be detected relative to time constraints. For example, you can tell it to trigger on events like if a setup time was too long or too short for a specification.

Listing 2—This is the `usleep()` function ported to DOS.

```
...
#ifdef DOS
#ifndef usleep
usleep(n)
    int n;
{
    unsigned long jiffies;
    jiffies = n*1;
    while(jiffies--);
}
#endif
#endif
...
```

This is where I started. I first set up the analyzer to look for I/O accesses on the PC/104 bus that maps to the I/O module. The analyzer was programmed to look for the following addresses:

0x0270–0x0271—configuration data (wr), status (rd)
 0xaa70–0xaa71—reconfigure (wr), nop (rd)

I needed to look for timing violations. On a fast machine, sometimes bus timing on the ISA bus or PC/104 is violated. So, I checked that the read and write pulses to the board were long enough to register with the FPGA on the board. This checked out OK.

In the next test, I programmed the analyzer to look for any accesses to the FPGA module and let the system go

through several reboot cycles with the cache on and off. The only time the board was being accessed was when I ran the program to configure the FPGA. Although this was not conclusive, at least it looked like none of the plug-n-play or PCI bus devices were being mapped at the module addresses.

My logic analyzer also lets me program the logic threshold levels for the inputs. This allows me to check for minimum TTL levels that the bus has to transmit in order to be TTL-compatible. This turned out to be OK, too.

HEAD SCRATCHING

Because I wasn't sure what was happening to the FPGA when it didn't program correctly while disk caching was on, I decided to start looking at the FPGA in more detail.

Listing 3—Here's the implementation for `mysleep()`.

```
...
#ifdef DOS
#include <time.h>
#ifndef mysleep
mysleep(n)
    unsigned long n;
{
    clock_t x,y;

    n = n / 1000; /* convert to msec. */
    x = clock();
    while(1){
        y = clock();
        if((y - x) > n)
            break;
    }
    return ;
}
#endif
#endif
...
```

The FPGA programming chain looks like the one shown in Listing 1. The lead FPGA is set up to program in asynchronous peripheral mode. In this mode, it will use chip selects and look like a chip on a bus. The chip selects are provided by a PLD on the module that acts like an address decoder.

The address decoder generates the chip selects for configuration accesses to the lower address. During a read, the FPGA will provide a status bit on D7 of the address bus. This bit will be a one if the FPGA is ready to receive a configuration word.

The PLD also decodes the program pulse *PROG. When the higher address is accessed with a write, a *PROG pulse is generated to all of the FPGAs, which causes them to go into a reconfiguration cycle.

But there also is the *INIT signal, which is shared among FPGAs. This signal is used to retard the configuration process. After the FPGA receives the *PROG pulse, it will tri-state all of the outputs that aren't being used for configuration and clear the SRAM memory used to store the configuration data. As this is taking place, the *INIT signal stays low and the FPGA is catatonic and doesn't respond to status reads. Also, if the signal is held low externally, the FPGA will wait until the signal has been released before progressing to the configuration download phase.

So, the *INIT signal makes sure that all of the FPGAs that are in a configuration data chain are synchronized and ready to receive configuration data at the same time. This is necessary because FPGAs with different sizes take different amounts of time to clear the memory. A large FPGA takes longer than a smaller-capacity FPGA. The time is specified in the data book.

Armed with this knowledge, I added the *PROG and *INIT signals to the logic analyzer inputs and captured some traces using the start of programming (i.e., the write that causes the *PROG pulse).

I captured the *INIT and *PROG timing and compared them to the datasheet to make sure that the *PROG pulse met the minimum pulse width (which it did). I also had to make

sure that the *INIT pulse would release, indicating that the FPGAs are ready to start configuring.

It turned out that the latter definitely was a problem. Because I was using state analysis, the buffer would fill up before the *INIT line went high. Remember how I determined that a 4-KB buffer would fill up in 512 μ s? The FPGA types used on the boards may take up to 1 ms to clear their memory, therefore I couldn't capture this event in one buffer.

Luckily, my analyzer also has a timing mode. In this mode, it doesn't sample at each clock tick, but records the transition time of each signal. For data that is slow (less than 100 MHz) and does not change, the buffer will be able to hold more time than in state mode. In this case, I was able to capture the *INIT signal at about 600 μ s.

OK, so the timing checked out. At this point, I programmed the logic analyzer to look for any configuration data writes to the module while the

*INIT was still low, indicating that the board wasn't ready to accept the data. This turned out to be it! When the disk cache is on, the first configuration write happens at about 400 μ s, which is before the *INIT signal goes high. Without the disk cache, the first write comes in more than 1 ms. Gotcha! It's a software problem.

I verified that the problem didn't exist under Linux. The first configuration write came 10 ms after the program pulse, which is safe. Linux always has its disk cache on by default.

Well, I guess that could be seen as both good news and bad news. The good news was that, because I'm the designer of the module, I was glad it wasn't a hardware problem with the board or a compatibility problem with the CPU card. Hardware problems are usually expensive because you have to fix PLDs and swap them (or worse, make PCB changes and run a new batch of boards).

The bad news, however, was that I'm also the person who wrote the software to configure the module. So

the hardware guy in me was pointing the finger at the software guy, but in this case I wore both hats.

SOFTWARE GUY RETURNS

The configuration program was written in C originally to run under Linux (or POSIX-style RTOS). Remember I mentioned that it ran fine under Linux? Because several of my clients want to run or develop under DOS or Windows, this program has been ported to run as a DOS command line application that can also run under Windows in 16-bit mode. This was done using an old version of the Microsoft C compiler (V. 5.0), and was initially seen as a quick fix.

Because the problem is in the initialization phase of the FPGA, let's look at the code segment that's responsible for this. Check out the code segment in Listing 1. The program opens the configuration data file in binary mode. For the DOS version, you have to look at all the define blocks that are outlined using the DOS constant. Then, after opening the file successfully, the pro-

gram sends the program pulse to put the FPGAs on the module into the configuration state.

It waits 10,000 μ s, or 10 ms, for the FPGAs to clear the configuration memory. It then reads the first configuration word using the `gets1s()` function from the configuration file. Then it checks the status register of the FPGA and waits for the ready bit (D7) to be high before proceeding to write the configuration data.

When I first looked at this problem, I must have stared at this section of code for what seemed like forever. After learning what the timing problem was, it was easy to see what was happening. When the disk cache was on, the first read from the configuration files took less time than when the cache was off. This meant that `usleep()` was not working properly because the delay was always shorter than the specified 10 ms.

`usleep()` is implemented using system calls under Linux and many Unix-like systems. Under Linux this call is not a real-time quality of service, because it

guarantees that the calling process will sleep for at least the specified amount of time. It may be a bit longer, but that's alright for this application. However, when the program was ported to DOS, the call was implemented as a busy loop (see Listing 2).

Now the problem was clear. On a faster processor, an uncalibrated delay loop will run faster than on a slow processor. The complication was that, on this particular processor, there was a threshold effect with the file system speed. With a slower file system access (i.e., when the cache is off), the speed was slow enough to work. On a slower processor, it would work regardless of the file system access time.

Well, the fix was easy in concept, but tricky in practice. Isn't that always the case? It has been about two years since I last compiled this program for DOS. The machine I used doesn't exist anymore so I didn't have a copy of the ancient Microsoft C compiler needed to get it up and running. The Visual C++ development system, although capable of compiling command line code, doesn't have the libraries I needed to make it run in regular DOS. And I didn't have the patience (even though I said I have more these days) to trudge through all of the Microsoft disks looking for the proper SDK or DDK to add to the libraries, even if they do exist somewhere. Tracking down hard disks that contained an old installation of the V. 5.0 compiler also proved to be a daunting task, taking up a couple of afternoons.

When the compiler was found and installed on my current Windows system, I was able to compile the program to its old glory and verify that it was working just as poorly as before. But now I had to track down some old library documentation to find out what calls I could use under DOS to implement the `usleep()` call. I found some old manuals scattered in a box in my closet that suggested using the `clock()` call. The implementation for `mysleep()` now looks like Listing 3.

The call `clock()` returns the number of milliseconds that have elapsed. I tested it under a couple of versions of DOS and in Windows95 and 98 and it seemed reliable. `mysleep()` now

converts microseconds to milliseconds and calls `clock()`, comparing the elapsed time until the time has expired. Although the timing resolution is coarser than under Linux, it's conservative. Besides, the timing resolution for `usleep()` under Linux also is coarsely implemented.

I then verified the new version of the loader on several systems and used the logic analyzer to ensure that the timing was consistent.

Several lessons can be learned from these trials. Besides the obvious knowledge of not exhaustively testing your code after porting it, there's the lesson that even software folks should learn to use a logic analyzer. It's an invaluable tool for tracking down funky timing and flaky behavior.

Another lesson you can take from this is to never throw away old software or disk drives. You never know what you're going to need. I will copy all of my disk drives to CD-ROMs as soon as possible so I'll have an archive. Also, it's not a bad idea to keep old documentation. I couldn't find the

documentation I needed on Microsoft's web site, and was glad I held on to my old manuals because my memory of the contents has long been flushed out.

I'm not sure what to do with the old manuals. I'd like to scan and store them on CD-ROMs as well, but this could be time-consuming. Perhaps I'll borrow one of those auto-feeding scanners for future use. 📄

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

SOURCE

Compiler

Microsoft Corp.
(425) 882-8080
www.microsoft.com

Fred Eady

Rabbit Season

Part 3: Network Analysis

Rabbit season isn't over yet, as Fred continues with his series. This Rabbit's been through the briar patch and back, hiding along the way. But, in this next installment he eventually leads us into the clear blue sky.



Things have been harey around the Florida room since the last time we shared carrot bits together. I thought I was going to have to call in Marvin the Martian (one more character intent on destroying Bugs) to take care of this situation. Marvin didn't have to use his "space modulator" on the Rabbit, but he should've used it on me! I'll tell you about that later, but right now the Rabbit Semiconductor-based Ethernet project is ARPing and echoing data.

FLUSHING A RABBIT...

What I mean by this is the hunting kind of "flush." As you already know from previous installments on this subject, real-life rabbits like to think they can become invisible by being still. Well, electronic rabbits don't fall too far from the tree. Photo 1 shows my quick and nasty Rabbit development board-to-embedded Ethernet card interface. I simply added some double-row headers that came with the Rabbit Development Kit to the monkey board and put wire between the CS8900 Ethernet IC and the Rabbit's I/O pins. When I plugged the

Rabbit-controlled Ethernet board into the Florida room's 192.168.1.0 Ethernet segment, nothing happened. Maybe there is something to this invisible stuff.

Remember a rock group called The Tubes? They wrote a song that warned, "Don't fall in love" (you never know when things could change). Last month I gave the same advice in the caption for Listing 1. To paraphrase, don't fall in love with it because it might not stay the same. I've often heard such advice given to other bands because musical styles are always shifting. Looking at the new Listing 1, it seems that we (The Tubes and I) were right. The code did change. I didn't trash the array idea, just improved it. After days of chasing a wild rabbit around the Florida room, I decided to apply logic and a powerful network analysis tool to the Ethernet-IP-ARP-UDP data structures.

Starting at the top of Listing 1 you find your user-assigned Rabbit IP and MAC definitions. In today's competitive world, part of the MAC address should always be determined and assigned by the IEEE. You won't be working with external vendors here, so "RABBIT" will be your MAC address. Using a readable name rather than obscure hexadecimal numbers will make things easier when you examine the packet dumps later. The only restriction on the content of the CS8900A-CQ MAC address is that the very first bit must be 0. The "R" in RABBIT equates to 0x52, so you're OK

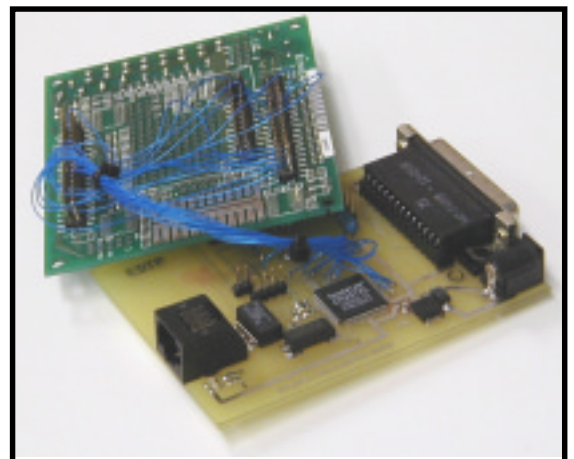


Photo 1—Wire wrap on the Rabbit end and solder techniques using vias on the CS8900A-CQ end made the embedded Ethernet board quickly intelligent.

in that area. The CS8900A-CQ individual address register holds the Rabbit's MAC address. For the purposes of ARP, UDP, and IP, your Rabbit IP address will be 192.168.1.3.

In Listing 1, the Ethernet packet header bytes are aligned in memory just as they occur physically. The first word in the Ethernet packet header area is reserved for the packet length. This is provided by the CS8900A-CQ and isn't in the standard Ethernet packet. Although not part of the Ethernet transmission, it's used in the Ethernet data transmission process.

The only way to make the

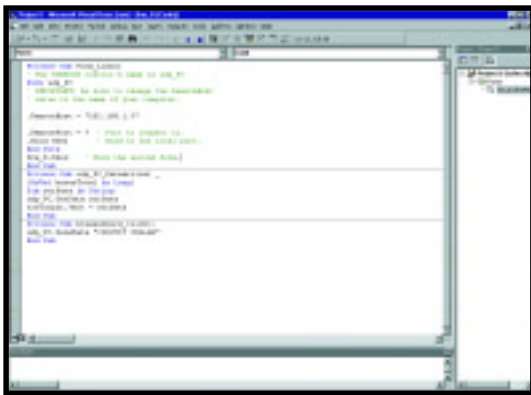


Photo 3—In this view the ARP cache is empty. By typing "ARP" without switches, you get additional details on other ARP switches that may be useful to you.

embedded Rabbit Ethernet visible to its counterpart in an IP sense is to issue an ARP request and make the Rabbit twitch. If the Rabbit Ethernet electronics receive the ARP request and the IP address matches your long-eared friend's address of 192.168.1.3, the Rabbit will certainly go into motion. In this case, a twitch is really an ARP reply. After the Rabbit senses and replies to the ARP request directed at its IP address, it is no longer invisible and other rabbit hunters can take pot shots at it on the network.

An ARP packet consists of the Ethernet packet header followed by 28 bytes of control information and sender/receiver MAC and IP addresses. A UDP message or UDP datagram is composed of the Ethernet packet header, the IP packet header, the UDP packet header, and the actual data. If you look closely at Listing 1, you'll find that the IP data area

(ip_data_begin) consists of the UDP packet header and UDP data area (udp_data_begin). The actual data in the UDP datagram is found in the UDP data area, which turns out to be a subset of the IP data area.

SNIFF OUT THE RABBIT

Before I get into the software details of acknowledging ARP requests, I'd like to explain why Marvin should've zapped me with his ray gun.

The ARP routines went together easily, as did the initial IP receive routines. I put together a simple VB6 Winsock program to send data to the embedded Ethernet board and then assembled the necessary software logic and Dynamic C routines to return (or echo) the incoming UDP datagram to the sender. I went through three days asking myself, "Why the heck won't this thing answer?", and then decided to run the code line by line to double-check the math and bit manipulations with my HP-16C.

Using the Dynamic C debug facility to verify or defy my HP-16C hand calculations, I finally traced the problem to a miscalculation of the IP and UDP checksums. A few hours later, I had the IP checksum problem rectified. I applied the same logic to the UDP checksum calculation and assumed (first mistake) that the UDP numbers were crunching correctly, too.

Another day passed and I was desperate for this invisible Rabbit to show itself. By now, I had stomped all over the briar patch. I was beginning to think the CS8900A-CQ IC was dead or the code I wrote killed the Rabbit. In any case, this project was starting to leave a bad taste in my mouth. I needed a tool to diagnose the Ethernet pieces my embedded Ethernet board was rejecting. It was time to break out the SNIFFER.

From here on I will be showing

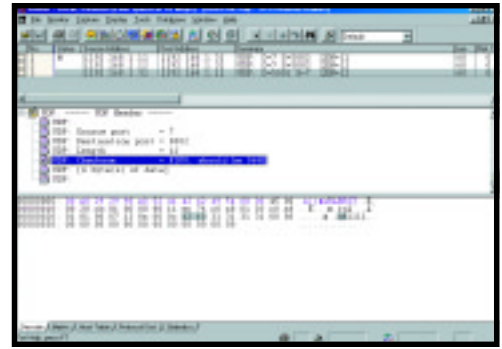


Photo 2—This is really nice! Notice that checksum is highlighted and so is the checksum word in the packet dump area. And now you know why I used "RABBIT" as the MAC address.

you screen shots of captured real-time Ethernet packets, thanks to Network Associates' SNIFFER. For those of you who are not familiar with SNIFFER, it's basically a PC, SNIFFER software, and a special NIC (Network Interface Card) that plugs into an Ethernet segment and monitors all of the traffic on the segment. In addition, the SNIFFER software breaks the packets down into their smallest parts and automatically identifies them to you as well. Using the special SNIFFER-approved NIC, I can also show you details at the physical wire level. Thus, I can generate and respond to Ethernet packets with the CS8900A-CQ/Rabbit 2000 IC combination and capture all of the transactions with the SNIFFER for later viewing and analysis. To summarize, what I see on the Rabbit's Ethernet segment is exactly what you see.

As it turns out, I was inserting a couple of 16-bit values that I should not have into my UDP checksum calculation. Photo 2 is the SNIFFER

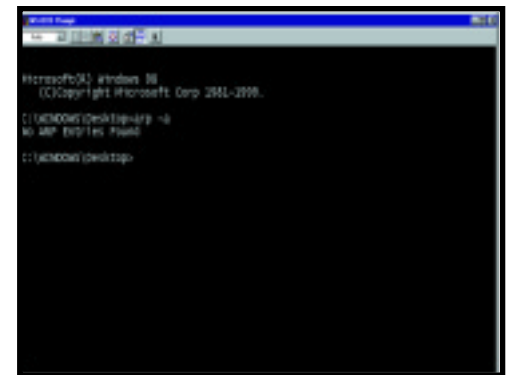
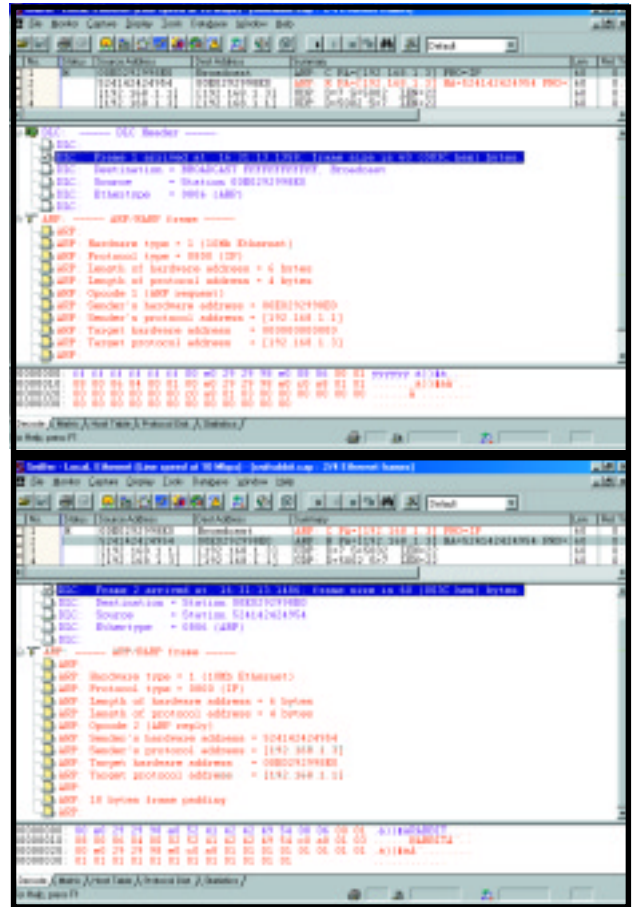


Photo 4—This is pretty simple stuff masking some complex logic. All you do is place Bill's Winsock control on the form and fill in the blanks and suddenly you're sending packets!

Photo 5—As if the SNIFFER folks weren't making it easy enough, they went ahead and color-coordinated everything too. Placing the cursor on a byte in the dump highlights its description, and vice versa. **Photo 6**—The MAC address really stands out in the carrot field.



view that pointed out my bad UDP checksum. After I saw that message, I knew exactly where to look for rabbit fur in the snow. Because I'm on the subject of bogus checksums, let's look at how the IP and UDP checksums are derived.

BOUNCING A CHECK

The first gotcha I ran into was that I defined the checksum variable as an integer. After all, the IP checksum is defined as a 16-bit sum of the parts with a one's complement twist. In actuality, the checksum variable ended up being a long (32-bit) variable. The trick to doing it right is to add any bits above the 16-bit line back to the lower 16 bits before complementing. Listing 2 is the IP checksum code snippet I cut from the `ip_received()` function.

The first order of business is to clear the checksum fields that came with the received packet. After they are cleared, the IP header checksum is calculated by totaling the following 16-bit words in the IP header:

- IP Version/header length (byte) and
- IP Type of service (byte)
- IP Packet length
- IP Datagram IDIP fragment offset
- IP Time to live (byte) and
- IP Protocol (byte)
- IP Source address
- IP Destination address

Any value beyond 16 bits is then added back into the lower 16 bits and

the total is complemented. For instance, suppose the total resulted in 0x294C0. The raw un-complemented checksum would become 0x94C2. The complemented and final checksum would then be 0x663D, which is placed in the IP checksum field defined as `packet[ip_checksum]` and `packet[ip_checksum+1]` in your Dynamic C Rabbit Ethernet driver code.

The UDP checksum is calculated in the same fashion, but the trip takes a turn or two in the process. Again, the first order of business is to clear the existing UDP checksum field. The checksum calculation then proceeds as follows:

- IP Source address
- IP Destination address
- IP Protocol (one byte only)
- UDP Length

Now, here's where things get a little tricky. The above total is added to the total beginning at the UDP source port field and continues for the length of the UDP packet. The weird

Listing 1—The key to this whole thing is understanding how all of the headers work with each other to define the protocols and deliver the data.

```
//      ETHERNET RABBIT STYLE

//      DEFINE THE RABBIT'S IP ADDRESS
// ALL CURRENT IP ADDRESSES ARE 4 OCTETS (BYTES)

#define rip0 0xC0 //first octet of IP address 192
#define rip1 0xA8 //second octet of IP address 168
#define rip2 0x01 //third octet of IP address 1
#define rip3 0x03 //fourth octet of IP address 3

// DEFINE THE RABBIT'S 48 BIT OUI (ORGANIZATIONALLY UNIQUE
// IDENTIFIER)
// THIS IS THE MAC ADDRESS OR HARDWARE ADDRESS

#define rmac0 0x52 // R
#define rmac1 0x41 // A
#define rmac2 0x42 // B
#define rmac3 0x42 // B
#define rmac4 0x49 // I
#define rmac5 0x54 // T

// GLOBAL DECLARATIONS

char packet[2048];
char rabbit_ip[4];
char rabbit_mac[6];
char swapper[6];
unsigned int
    i, incoming, udplength, packetlen, packettype, return_code;
unsigned int
    ppaddr1, ppaddrh, ppdata1, ppdatah, checksum_H, checksum_L;
unsigned int isqdata1, isqdatah, packetlen1, packetlenh;
unsigned int packetstatus1, packetstatush, packetlen_in;
unsigned int
    bstat, x, y, z, porta_data, pp_port_addr, pp_port_data, temp;
unsigned long checksum, checksumtemp;

// ETHERNET PACKET HEADER

#define len_H      0x00 //this is for PacketPage
#define len_L      0x01
#define mac_des0_H 0x02 //Ethernet destination MAC address
#define mac_des0_L 0x03
#define mac_des1_H 0x04 //Ethernet destination MAC address
#define mac_des1_L 0x05
#define mac_des2_H 0x06 //Ethernet destination MAC address
#define mac_des2_L 0x07
#define mac_src0_H 0x08 //Ethernet source MAC address
#define mac_src0_L 0x09
#define mac_src1_H 0x0A //Ethernet source MAC address
#define mac_src1_L 0x0B
#define mac_src2_H 0x0C //Ethernet source MAC address
#define mac_src2_L 0x0D
#define type_H     0x0E //packet type
#define type_L     0x0F // ARP PACKET
#define arp_packet_type 0x0806
#define ar_resp_len 0x2A //arp response frame length
#define ar_hw_type  0x10 //hardware type
#define ar_proto_type 0x12 //protocol type
#define ar_hw_len   0x14 //hardware address length
#define ar_proto_len 0x15 //protocol address length
#define ar_oper     0x16 //ARP operation
                        (1=request, 2=reply)
#define ar_sndr_hwadr 0x18 //senders
                        hardware address
```

(Continued)

thing about it is that the UDP length field gets counted twice. As you can see from Listing 2, when the words and bytes are totaled, the same 16-bit checksum is converted to a 16-bit version just like I did with the IP.

THE ART OF ARP

Now that we have a SNIFFER to seek out Ethernet problems, let's turn our attention to interpreting the data provided via the CS8900A-CQ electronics. To make interpreting the Dynamic C Ethernet code easier, I've prefixed PacketPage register offsets with PPO (PacketPage Offset).

PacketPage ports are all prefixed with PPP, so ppoLineCtl is the line control register. Each PPO is defined by its offset. In the case of ppoLineCtl, its PacketPage offset is 0x112. This is all laid out in the complete listing available on the *Circuit Cellar* web site. The offsets can be found in the CS8900A-CQ datasheet.

Listing 3 is the receive-event handler. I included a snippet at the top of Listing 3 that I took from the CS8900A-CQ startup and initialization routines. Notice that the Receiver Control Register (ppoRxCtl) has been loaded with bits that allow individual MAC addresses (RxCtl_IND_A) and broadcast addresses (RxCtl_BCAST_A) to pass through the CS8900A-CQ and be accepted for scrutiny by the Rabbit's code.

The first four bytes the Rabbit sees are two bytes of packet status that you won't use and two bytes of packet length information that you will use. After storing the packet length data in the first word of your Ethernet packet array, you then proceed to get all of the bytes that make up the Ethernet packet using the packet length you received earlier. The result is an array the size of a packet length containing the Ethernet MAC addresses and ARP, or IP packets that match your layout of header memory definitions. How did you know to get this particular Ethernet packet? In this scenario, the answer is based on ARP. Let's define the players.

As you know, each IP-addressable machine on a network is called a host. Your network consists of two active hosts and a passive monitor, the

Listing 1—continued

```
#define ar_sndr_ipadr      0x1E    //senders IP address
#define ar_targ_hwadr     0x22    //target hardware address
#define ar_targ_ipadr     0x28    //target IP address

// IP PACKET HEADER
#define ip_proto_icmp     0x01    //ICMP protocol type
#define ip_proto_tcp      0x06    //TCP protocol type
#define ip_proto_udp      0x11    //UDP protocol type
#define ip_verhdr_len     0x10    //IP version and
                                   header length
#define ip_tos            0x11    //IP type of service
#define ip_packet_len     0x12    //IP packet length
#define ip_dgram_id       0x14    //IP datagram ID
#define ip_frag_offset    0x16    //IP fragment offset
#define ip_ttl            0x18    //IP time to live
#define ip_protocol       0x19    //IP Protocol
#define ip_checksum       0x1A    //IP header checksum
#define ip_src_addr       0x1C    //IP source address
#define ip_des_addr       0x20    //IP destination address
#define ip_data_begin     0x24    //IP data area

//UDP HEADER
#define udp_echo_port     0x07
#define udp_src_port      ip_data_begin //UDP source port
#define udp_des_port      udp_src_port + 2 //UDP destination port
#define udp_len           udp_des_port + 2 //UDP header and data length
#define udp_checksum      udp_len + 2 //UDP checksum
#define udp_data_begin    udp_checksum + 2 //UDP data area
```

SNIFFER. Each host on your network and the SNIFFER are physically connected to a small 4-port NETGEAR 10BaseT Ethernet hub. The PC host is a WIN98 desktop running a Visual Basic Winsock application that simply sends a "CIRCUIT CELLAR" message to the Rabbit-controlled Ethernet module. The PC host's VB program knows that the Rabbit is at 192.168.1.3. The PC is located at 192.168.1.1. In your example code, after the Rabbit receives the UDP message from the PC, it immediately echoes the message back to the sender. As expected, the VB6 code is a piece of cake (see Photo 3).

When every player is powered on and no communications activity has been initiated by any host, the PC's ARP cache is empty. That means there are no stored IP addresses mapped to any stored MAC addresses that the PC host knows about. I issued the ARP -a command as shown in Photo 4. The -a switch in the ARP command displays current ARP entries by interrogating the host's current protocol data. Because there's no Dynamic C code to implement an ARP

cache on the Rabbit side, and none to enable the Rabbit's Ethernet engine to generate an ARP request, you can safely assume that the Rabbit and CS8900A-CQ are equally as IP/MAC dumb as the PC at this point.

Now, everything is set. The next move is to start the SNIFFER in capture mode so you can see all of the events from the instant you click on the VB Winsock application's "Send UDP Packet" button, until the message is bounced back by the embedded Ethernet carrot chopper.

Turn your attention to the SNIFFER shot in Photo 5. That simple mouse click on the VB form generated four Ethernet events. The first event was an ARP request, which was generated by the PC host. The SNIFFER has made this easy to figure out. Without going into thorough detail, the topmost frame of Photo 5 tells you the MAC address of the sender (0x00E0292998E0). It also informs you that the packet is a broadcast. Not only is this a broadcast, but it's also an ARP request looking for the MAC address of the

owner of IP address 192.168.1.3.

Looking at Listing 1 in the Ethernet header area, it's obvious that the DLC (Data Link Control) header area in the SNIFFER shot is actually the Ethernet header. Hopping over to the ARP area of the SNIFFER ARP shot, you quickly see that everything is known except the MAC address of the target

host. Finally, the bottom SNIFFER window gives the bit-bangers in the audience something interesting to look at. Note the series of 0xFFs (indicating a broadcast) in the MAC destination fields of the SNIFFER dump. Also, notice that you don't see the word "RABBIT" in the ASCII part of the dump. That's because our rabbit is still

and, thus, invisible.

To our Rabbit, the ARP request was like a shotgun blast that came too close for comfort. Our invisible electronic fur ball was forced to twitch and, thus, return the MAC information requested by the PC's ARP request. The SNIFFER shot in Photo 6 speaks for itself. At the DLC level, you see the SMC 9432TX NIC MAC address (0x00E0292998E0) and a new source MAC address RABBIT, or 0x524142424954. Notice that the Rabbit sent this ARP reply and the destination is the information that arrived from the source machine that issued the ARP request. The Rabbit is on the run, as you can see from the ASCII equivalent of the Rabbit's MAC address in the Ethernet header and the ARP packet. The CS8900A-CQ automatically adds padding to the packet to meet the minimum length requirements. In this case, the SNIFFER points out that the frame needed 18 more bytes to be legal.

Looking again at Listing 3, you can see the Dynamic C code getting the bytes into an array and then checking the incoming packet's fields in an attempt to identify what type of protocol the packet contains. ARP packets are fairly straightforward and if any of the checks fail under case 0x0806, it means the packet doesn't match. Because it's useless, it is subsequently discarded.

RABBITS HATE SHOTGUNS...

Especially when they're double-barreled! The ARP request and resulting reply by the Rabbit enabled the host station on the PC to zero in with barrel number two. Check out Photo 6 again and note that Event 3 does not contain any references to MAC addresses. That's because the Rabbit's IP address and MAC address have been assimilated by the PC's Winsock functions and placed in the PC's ARP cache (see Photo 7). Also, 0x0800 in the type field denotes the third packet as an IP packet. IP packets use IP addresses. The IP layer depends on the DLC layer to handle the physical NIC-to-NIC addressing.

If this were a real hunting trip, the next shot fired by the VB Winsock application would put Bugs in the

Listing 2—After you know what the fields and their offsets mean, it all falls logically into place. Suddenly the mist lifts and you find yourself in Ethernet land.

```
//IP CHECKSUM ROUTINE
checksum=0;
for(i=0;i<2;++i){packet[ip_checksum+i]=0x00;}
checksum=checksum+(packet[ip_verhdr_len] << 8 | packet[ip_tos]);
checksum=checksum+(packet[ip_packet_len] << 8 |
packet[ip_packet_len+1]);
checksum=checksum+(packet[ip_dgram_id] << 8 |
packet[ip_dgram_id+1]);
checksum=checksum+(packet[ip_frag_offset] << 8 |
packet[ip_frag_offset+1]);
checksum=checksum+(packet[ip_ttl] << 8 | packet[ip_protocol]);
checksum=checksum+(packet[ip_src_addr] << 8 | packet[ip_src_addr+1]);
checksum=checksum+(packet[ip_src_addr+2] << 8 |
packet[ip_src_addr+3]);
checksum=checksum+(packet[ip_des_addr] << 8 | packet[ip_des_addr+1]);
checksum=checksum+(packet[ip_des_addr+2] << 8 |
packet[ip_des_addr+3]);
checksumtemp=checksum >> 16;
checksum=checksum+checksumtemp;
checksumtemp=checksum & 0x0000FFFF;
checksum=~checksumtemp;
checksum_H=(checksum & 0x0000FF00) >> 8;
checksum_L=checksum & 0x000000FF;
packet[ip_checksum]=checksum_H;
packet[ip_checksum+1]=checksum_L;
//UDP CHECKSUM ROUTINE
udplength=(packet[udp_len] << 8 | packet[udp_len+1]);
checksum=0x00;
for(i=0;i<2;++i){packet[udp_checksum+i]=0x00;}
checksum=checksum+(packet[ip_src_addr] << 8 | packet[ip_src_addr+1]);
checksum=checksum+(packet[ip_src_addr+2] << 8 |
packet[ip_src_addr+3]);
checksum=checksum+(packet[ip_des_addr] << 8 | packet[ip_des_addr+1]);
checksum=checksum+(packet[ip_des_addr+2] << 8 |
packet[ip_des_addr+3]);
checksum=checksum+packet[ip_protocol];
checksum=checksum+(packet[udp_len] << 8 | packet[udp_len+1]);
i=0x00;
do{
    checksum=checksum+(packet[udp_src_port+i] << 8 |
packet[udp_src_port+i+1]);
    ++i;
    ++i;
    --udplength;
    --udplength;
    }while(udplength != 0);
checksumtemp=checksum >> 16;
checksum=checksum+checksumtemp;
checksumtemp=checksum & 0x0000FFFF;
checksum=~checksumtemp;
checksum_H=(checksum & 0x0000FF00) >> 8;
checksum_L=checksum & 0x000000FF;
packet[udp_checksum]=checksum_H;
pIacket[udp_checksum+1]=checksum_L;
```

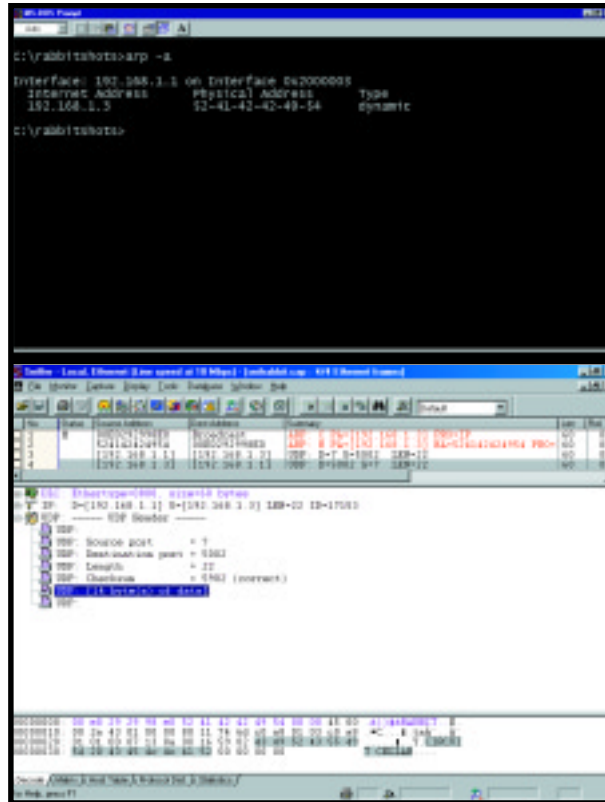
Listing 3— *It doesn't take much to kick start the CS8900, but after you get it started it drives like a fine sports car.*

```
// initialize the CS8900
ppwrite(ppoLineCtl,LineCtl_10BASET);
ppwrite(ppoTestCtl,TestCtl_FDX);
ppwrite(ppoRxCfg,RxCfg_RX_OK_IE);
ppwrite(ppoRxCtl,RxCtl_RX_OK_A | RxCtl_IND_A |
        RxCtl_BCAST_A);
ppwrite(ppoTxCfg,TxCfg_ALL_IE);
// Receive Event Handler
void Receive();
void Receive() {
DATAIN;
WrPortI(PDDR,&PDDRShadow,pppRxTxData0+1);
ioread();
packet[0]=packetstatus;
WrPortI(PDDR,&PDDRShadow,pppRxTxData0);
ioread();
packet[1]=packetstatus1;
WrPortI(PDDR,&PDDRShadow,pppRxTxData0+1);
ioread();
packetlenh=porta_data;
packet[0]=packetlenh;
WrPortI(PDDR,&PDDRShadow,pppRxTxData0);
ioread();
packetlenl=porta_data;
packet[1]=packetlenl;
packetlen=packetlenh << 8 | packetlenl;
packetlen_in = packetlen;
i=2;
do{
        WrPortI(PDDR,&PDDRShadow,pppRxTxData0);
        ioread();
        packet[i]=porta_data;
        ++i;
        --packetlen;
        WrPortI(PDDR,&PDDRShadow,(pppRxTxData0+1));
        ioread();
        packet[i]=porta_data;
        ++i;
        --packetlen;
    }while(packetlen != 0);

packettype=(packet[type_H] << 8 | packet[type_L]);
switch(packettype){
    case 0x0806:
        arp_chk_hwtype();
        if (return_code) {arpchk_proto();}
        else { break;}
        if (return_code) {arpchk_hwlen();}
        else { break;}
        if (return_code) {arpchk_protlen();}
        else { break;}
        if (return_code) {arpchk_request();}
        else { break;}
        if (return_code) {arpchk_ip_1();}
        else { break;}
        if (return_code) {arpchk_ip_2();}
        else { break;}
        if (return_code) {arpchk_ip_3();}
        else { break;}
        if (return_code) {arpchk_ip_4();}
        else { break;}
        reply_to_arp();
    case 0x0800:
        ip_received();

    default:
        trash_packet();
    }
}
```

Photo 7—And, the shotgun is still loaded. **Photo 8**—I used an easy-to-read data package here. The idea is to convey to you where the data really is.



boiling pot. Event 3 of Photo 6 shows you that the host station at 192.168.1.1 (PC/VB application) sent a UDP datagram to the host station at 192.168.1.3 (the carrot chopper). The standard echo port is UDP destination Port 7, therefore, to answer any incoming echo

requests, the Rabbit must know what port to listen on. The first define statement under the Rabbit code // UDP header in Listing 1 sets the Rabbit listening port number, and the incoming destination port field from a remote host is verified in the Rabbit's reply_to_arp() function of your Rabbit C code. In Photo 6, Event 3, the source port is, of course, the UDP port used by the VB Winsoc program running on the PC host. A quick look at Photo 3 shows that the .Bind 5002 instruction is a blood relative of the S = 5002 source port definition shown by the SNIFFER. At this point, if the code is right and the checksums are calculated correctly, the Rabbit has no

other alternative than to echo the message in the UDP datagram. Photo 8 is the SNIFFER view and Photo 9 is the VB host view after the dust settles. The good news is that our fast cotton-tailed friend is electronic and bounces back just like the rabbits you knock over in an amusement park shooting gallery.

THAT'S ALL, FOLKS

In addition to echoing characters, you now have the knowledge that will enable you to send hexadecimal values within Ethernet packets that can be interpreted as commands to force your Rabbit to perform via the Ethernet. What you do currently with serial and parallel ports can also be done with the Ethernet port because now you know that Ethernet isn't complicated, it's embedded. ☺

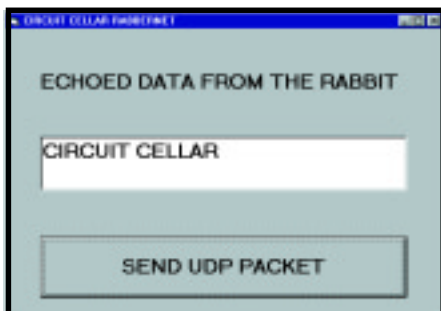


Photo 9—According to the SNIFFER timestamp, this message was transmitted and echoed back in .031 seconds.

Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.

Design2K Contest

Blaze a Trail for the new Millennium.

**CIRCUIT
CELLAR®**
THE MAGAZINE FOR COMPUTER APPLICATIONS



PHILIPS

Let's make things better.

For complete abstracts, go to www.circuitcellar.com/design2k/winners.

Grand Prize

DDSGEN
Robert Lacoste

Chaville, France
robert_lacoste@yahoo.fr

The DDSGEN is a full-featured DDS-based generator that can generate sinusoidal and square signals from 0–120 MHz with a resolution of up to 0.001 Hz. The DDSGEN supports a variety of modulation modes (AM, FM, PM, shaped keying, FSK, PSK), as well as wobulation (programmed jitter). The DDSGEN can also be extended by daughter boards to implement a full-featured Arbitrary Signal Generator (ARB) and pulse generator. And last but not least, its cost is reasonable. The DDSGEN is primarily built around an AD9852 DDS chip from Analog Devices and is controlled by a pair of Philips 87LPC764 microcontrollers (one main, one dedicated to the user interface).

The DDSGEN features include:

- onboard user interface (2 × 16 LCD, keyboard, rotary encoder)
- all functions can be remotely controlled through a RS-232 connection
- 300-MHz internal clock frequency, standard 100-ppm stability or OVCXO based 3 ppm option
- 0 to 120 MHz output frequency with 0.001 Hz resolution from 0 to 999 KHz and 1-Hz resolution from 1 MHz to 120 MHz
- sinusoidal output from 0 to ±3 V, 12-bit resolution, programmable offset of 0–3 V
- low jitter squarewave output, 3.3- and 5-V compatible
- fully digital AM, FM, PM (amplitude, frequency, and phase modulation) up to 5 KHz with programmable depth, 10-bit resolution; internal modulation generator option
- shaped keying (0 to 100% amplitude modulation based on a digital signal), with programmable slope rate
- digital ramped FSK (Frequency Shift Keying) between any two frequencies; immediate or programmable change rate
- digital ramped PSK (Phase Shift Keying) between any two phases (resolution 12 bits); immediate or programmable change rate
- linear wobulation between any two frequencies, programmable repetition rate
- 0- to 30-MHz clock with 0.001-Hz resolution, 12-bit amplitude resolution, 1 × 8k-word signal or 8 × 1k-word signals. Serial download of the waveforms through the RS-232 port to an onboard EEPROM
- future optional pulse generator: 10-ns maximum resolution, 24-bit length register/32-bit repetition register
- optional high-precision 3-ppm OVCXO system clock
- future optional ARB (Arbitrary Signal Generator)

The software for the DDSGEN is mainly written in C, thanks to the freeware SDCC-optimizing cross-compiler. The dynamic structure of the DDSGEN embedded software is of the classic (but field-proven) interrupt driven variety. After initialization, a main program manages the user interface and stores in a shared RAM buffer all parameters that need to be loaded into the DDS chip.

An interrupt routine, executed each time the DDS chip asks for new values (usually every 200 μs), executes an A/D conversion of the external modulation input and recalculates the frequency and amplitude (and/or phase if a modulation is requested) on the fly and uploads the modified parameters into the DDS chip.



Grand Prize

Design2K
Contest

For complete abstracts, go to
www.circuitcellar.com/design2k/winners.

QuizWiz

Paul Kiedrowski

Fort Worth, TX
kiedro@swbell.net

A common practice for automatic scoring of multiple-choice quizzes or tests is to use a commercially available system based on a desktop card reader machine, which requires that students mark their answers on preprinted forms of specific size and layout. This method is relatively expensive because of the cost of equipment and score cards, and therefore is usually used only for critical testing.

In most cases, because only one centralized scoring machine is available to the teacher, it is not located in the classroom where it would offer the most convenience. Perhaps more importantly, the most useful time to evaluate test results would be immediately afterwards so that teachers could promptly give feedback and discuss the most commonly missed questions. This is especially desirable for periodic quizzes where the intent is to allow the teacher to quickly gauge the classroom's learning progress.

To answer the above needs, a new scoring device was developed based on the Philips 87LPC764 microprocessor. The QuizWiz uses a single reflective opto-sensor to perform the scanning detection process. To preserve battery power, the opto-sensor LED is only active when QuizWiz is pressed against the paper, which depresses a mechanical switch located on the bottom.

Normal battery current is about 25 mA when all circuits are operating, 15 mA when not scanning, and 20 μ A during shutdown. Using three AAA batteries in series, with a typical capacity of 1000 mAh, a teacher can score approximately 100 quizzes for 30 students (i.e., 3000 scans).

QuizWiz uses a simple 3-chip design (processor, 5-V converter, and RS-232 interface). The 87LPC764 is a good match for the required features, and all of its pins and most of its features are used in this project. To minimize cost further, no external crystal is required, because the processor conveniently includes an internal 6-MHz RC oscillator.

For access to quiz scoring details in real time, as they are scanned, the user may connect a PC to the QuizWiz using a standard RS-232 serial port connection at 9600 bps. The QuizWiz automatically detects the presence of the serial port connection, and power usage is reduced when not connected. When the serial port is connected, the PC will receive the results of each quiz as they are scanned and completed. The questions that are wrong will be reported as well. A standard terminal emulator program can be used on the PC, and the results can be copied and pasted into other programs.



Grand Prize

InLine MIDI Monitor

Robert Morrison

Star, ID
rdm@boi.hp.com

The Musical Instrument Digital Interface (MIDI) is the RS-232 of the musical world. However, much like RS-232, MIDI has its downside. If something doesn't work, it's difficult to find out why. The InLine MIDI Monitor was designed to quickly diagnose problems unique to the MIDI environment. The monitor looks like a cable that plugs into a MIDI port and displays the status of the line. The device can be battery powered for quick debugging while on the road, or run from a wall transformer to provide continuous MIDI line monitoring during a performance.

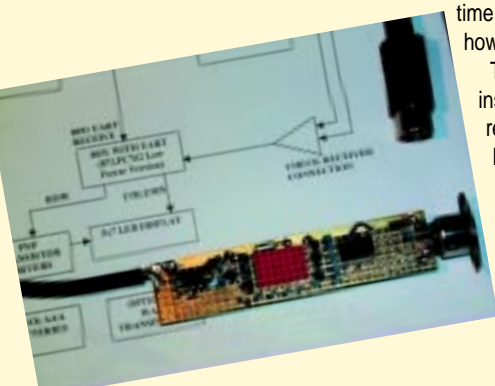
As an amateur classical pianist and synthesizer player, I have a MIDI-compatible piano and MIDI-compatible synthesizer equipment. My biggest nightmare is beginning a solo performance by playing a dramatic opening chord and hearing no sound, except for the clunk of the synth keys. All too often, a cheap MIDI cable prevents data from getting from the keyboard to the rackmount synthesizer box. What's worse is that I will have to check many things before determining the problem. Is it an incorrect channel setting of the keyboard or synth? An amplifier or mixer line cable that didn't get plugged in? Maybe a bad MIDI cable?

The MIDI protocol allows every note or other command to be assigned to one of 16 channels, so if there is a channel mismatch between the keyboard and the synth, no sound will occur. Wouldn't it be nice if I could be reassured that the connections were all working so I could concentrate on the performance? That's why I designed the InLine MIDI Monitor—an unobtrusive, battery-powered, 87LPC762-controlled monitor that provides visual information about the MIDI connection.

The InLine MIDI Monitor checks for several things and provides its status on a small 5 x 7 LED array. The MIDI Monitor checks for receiver connectivity, active messages, channel information, and note integrity/noise. Because MIDI is a current loop, if the receiving end of the cable is not properly connected, it is easy to detect—this verifies that the rackmount synthesizer is actually present (cable is connected). That may not sound like a big deal, but unless you've done time as a band roadie, you don't know what a rat's nest of cables are involved in an instrument setup, and might not realize how easy it is to leave some cables unconnected or connected to the wrong device.

The MIDI Monitor will also check for channel settings. This is one of the most common goofs in making sure that MIDI instruments are talking to each other. I will set the keyboard to transmit on channels 0 and 1, but set the synthesizer to receive on channels 1 and 2! If I'm lucky, I will figure it out quickly, hunt down the MIDI channel configuration for both the keyboard and the synth and set them to match.

To solve the problem, the MIDI Monitor will display (in slow sequence) the active channels on the line so I can verify that the transmitting device is functioning and what channels it is sending on. It will also watch for the MIDI active messages that are sent periodically, and display an idle status (IDL) if no keyboard activity is occurring (it displays inactive status (IA) if there are no MIDI active messages at all). With such a powerful tool checking the MIDI cable connection, I can rest easy and concentrate on the performance!



Second Prize Winners

TV Timer

Lionel Theunissen
Brisbane, Australia

The TV Timer is a low-cost channel control designed for use in the hospital and hospitality industries, or any situation where access to specific television channels needs to be controlled. Existing solutions such as set-top decoder boxes or scramblers provide functionality but can be quite expensive. The TV Timer has an on-screen status display, is operated via a custom infrared remote control, is self-contained within the television, and best of all, only costs about \$40 to build. The TV Timer allows access control of “pay” channels while allowing transparent access to free programming.

Projector Assisted Sculpture Turntable

Nathon Van Noy
& Mark Patterson
Provo, UT

To obtain an accurate likeness, a sculptor must utilize some form of reference material. Photographs are the most widely used reference material, but they offer a limited dimensional perspective and can cause distractions when the artist looks away from the sculpture to study the photograph. However, by joining a rotary table mechanism and a slide projector, the rotary table actuated slide projector provides a continuously correct three-dimensional perspective of a slide-projected image to the sculpture in progress.

The Geo-Mite

David Penrose
Bedford, NH

The Geo-Mite is a small microprocessor-controlled vibration alarm system. It uses a sensor that was developed to perform acoustic surveying for the oil industry. This sensor can detect small vibrations and generate an output voltage proportional to the vibration. The Geo-Mite converts this output to a binary condition and then counts the number of these vibrations occurring each second. This level of vibration is displayed on an LED bar graph on the front of the Geo-Mite. If this level exceeds a programmable threshold, the Geo-Mite will sound an alarm and send an X-10 On command to a selected device. The unit also can operate without the X-10 interface and function as a simple audible vibration alarm.

Pocket Logic Analyzer

Michael Kroon
Horsnby Heights,
Australia

With the ever increasing performance of microelectronics, the home user and small entrepreneur cannot afford to keep pace with the need for special test equipment. Getting all of today's new microcontrollers to operate and execute code often requires a means of being able to view the relative timing of different signals in a system to get it going. With the requirement for high-speed clocking and configurable settings, the heart of this eight-channel serial interface analyzer is a CPLD from Lattice Semiconductor. These devices are extremely fast, which results in a system that can sample a high-speed bus. All signal acquisition is achieved by hardware and thus the microcontroller is free to handle the interfacing, setup, and control functions.

For complete abstracts and photos,
go to www.circuitcellar.com/design2k/winners.

For complete abstracts and photos, go to www.circuitcellar.com/design2k/winners.

FPGA on a USB Cable

Michael DeVault
Penfield, NY

The FPGA on a Rope board and device driver provide a simple method to quickly configure and control a user-definable FPGA connected to a Windows 98 PC through a USB port. The design consists of the USBFPGA hardware (printed circuit board), and a Windows 98 compatible WDM device driver. An example application and FPGA template design are included to demonstrate the control and status functions of the USBFPGA device. These functions allow you to develop custom applications that communicate with your defined FPGA hardware via the USB port.

SatPoint

Kenneth Trussell
Sandersville, GA

SatPoint is a portable satellite tracking device with an illuminated pointer that gives specific location information. By connecting the unit to a PC via RS-232, it can download up to 100 satellite passes and data that may span several days. The data is stored in nonvolatile memory so the unit can be turned on and off as needed without having to reload the tracking data. The PC sends the current date and time to SatPoint at the beginning of the data set. SatPoint sets its internal date and time and maintains that information automatically so that it is ready to track real-time at any time. Pushbuttons and a two-line by 20-character LCD interact with the operator and allow the selection of various modes of operation.

The Yard-Stick

David Penrose
Bedford, NH

The Yard-Stick is a tool for the landscape planner, builder, architect, accident investigation professional, or anyone who needs to measure and record the outline of an irregular area. The device combines the familiar rolling wheel measuring instrument with an accurate digital compass, internal memory, display, processor, and RS-232 interface. The resulting tool can measure and record the outline of multiple areas, measure distances and direction between these areas, and then transfer this information to a PC for display and calculation. This information can then be used to produce both graphical and textual description of the scene. This data can be overlaid on photographic information to produce a complete record of the area.

Ultimate Clock and Message Display

Chuck Cateora
Aurora, CO

The clock and message display presents the date, time, and an optional user-defined message in a scrolling marquee fashion. A large 320-LED display continuously scrolls the date, time, and user message across the display using a dot-matrix font. Bright LEDs and a large 8 × 8 font make the display easily readable from far away. The pushbuttons on the front of the clock allow for easy setup and changes. The clock is based on the 60-Hz AC commercial power grid for timekeeping accuracy and uses a "super cap" instead of batteries for backup.

The Geo-Sentry

David Penrose
Bedford, NH

The Geo-Sentry is a programmable real-time event recording and display system. It interrogates a series of small seismic sensors to detect foot or vehicle traffic. The system is designed to monitor these sensors and send X-10 commands when the level of activity from a sensor indicates that a person is walking in the area or that a vehicle is moving in the driveway. By associating a different X-10 command sequence and On time with each sensor, the outdoor lighting near the driveway and house can follow a person's movement. This adds safety for the walker and security for the homeowner.

Electronic-Lab

Mariano Barron Ruiz &
Javier Martinez Perez
Zaldibar, Spain

The Electronic-Lab enables educational personnel to simulate analog and digital practice exercises with real devices. It provides the mounting surface for practices, the power source for the circuits, two signal generators, a clock generator, a 16-bit word generator, an I²C bus controller, a frequency meter, an oscilloscope for up to eight analog inputs, an oscilloscope for up to 16 digital inputs, and a 16-bit pattern detector. With the practice handbooks that are included, instructors can make the best use of PC resources to teach the student and to generate, monitor, and register the signals, as well as verify fixed-function SSI/MSI integrated circuits and programmable logic devices.

THIRD PRIZE WINNERS

For complete abstracts and photos, go to www.circuitcellar.com/design2k/winners.

Machinist's Tachometer

This project is a sophisticated optical tachometer that uses a PhilipsP87LPC764 as the only integrated circuit. Because tachometer signals are relatively low frequency, it uses a reciprocal counting scheme whereby the frequency (and thus rotational speed) is calculated from the time it takes to receive an integral number of input pulse edges. This allows an update rate that usually is in the 2 Hz range with 1 RPM resolution.

Spehro Pefhany

Toronto, Canada — speff@interlog.com

Easy Altimeter

The Easy Altimeter is an inexpensive altimeter project that also includes a thermometer and barometer. The altimeter has a resolution of about 1 m and calculates data faster than most GPS units, making it practical for outdoor hobbies such as hiking or biking. It can store up to 10 hours of data, which then can be downloaded to a PC.

Radek Vaclavik

Roznov Pud Radhostem — radek.vaclavik@onsemi.com

PC-based Machinery Vibration Analyzer

Machinery vibration analysis is used to routinely check the health of machinery and also to understand the cause of vibration. Vibration is often a symptom of an internal defect and can be an early predictor of developing defects. This low-cost 51LPC-based data acquisition system can be used with a portable PC to read vibration signals (acquired from an accelerometer). The signals are then plotted, saved, and analyzed using Fast Fourier Transform algorithms.

Ariel Quezada

Cochabamba, Bolivia — ari_quezada@yahoo.es

Weathermon

Controlled by a simple user interface of one LED and four pushbutton switches, the Weathermon incorporates several Dallas 1-Wire bus devices to implement three weather sensors. The Weathermon reads the ambient temperature, wind direction, and wind speed data from a weather station assembly and displays the processed weather data as text on a video monitor or television receiver.

Peter Ampt

Victoria, Australia — pampt@alphalink.com.au

The Bit-Banger: USB to I²C bridge

The Bit-Banger implements a USB to I²C (single master) bridge, as well as some general purpose I/O, which is accessible via USB through a set of API calls. The design is intended to be used with a series of I²C feature boards to provide a simple method for engineers to access a variety of functions (e.g., an LCD interface, digital I/O, 7-segment displays, keypads, or ADCs) through a USB interface, using a simple set of function calls (API).

Michael DeVault

Penfield, NY — michael@devasys.com

I²C-MMI

Developing an embedded system can be fun. Doing it twice is called experience. Redeveloping the same kind of software over and over is a waste of time and money, not to mention that it can get boring. The I²C-MMI is a dedicated a low-cost preprogrammed chip that directly handles all the elements of a classic user interface, is fully reusable, communicates with the host micro through a standard nondedicated I²C bus, and can offload as many functions as possible from the main micro.

Robert Lacoste

Chaville, France — robert_lacoste@yahoo.fr

Stick Shift Auto Racing Simulator

The Stick Shift Auto Racing Simulator is a modification to a commercially available steering wheel joystick interface that provides a realistic auto racing environment for computer, yet requires no special modifications to the computer game. The addition of a clutch pedal and a stick shift assembly allows the player to experience an engine stall or grinding gears, just like you would in a standard transmission automobile.

Robert Morrison

Star, ID — rdm@boi.hp.com

I²C Menu

The I²C Menu is an I²C interfaced unit for simple and flexible menu-type selection, including the ability to write application-specific text to the LCD module. The project uses a 2 x 16 LCD, a buzzer, backlight control, contrast regulation, two soft keys, and two scroll keys. To save application code space, the menu structure and setup values for the unit can be preloaded into the EEPROM.

Robert Klingbeil

Midrand, South Africa — klaxe@mweb.co.za

Strobe Clock

The Strobe Clock is a pen-sized pocket clock that displays the time with the wave of the wand by using LEDs aligned in a vertical column. By turning the LEDs on or off at the proper positions, any number of characters can be displayed. Using only five LEDs, it achieves a vertical resolution of five and an unlimited horizontal resolution.

Mircea Hossu

Mississauga, Canada — mhossu@hotmail.com

R/C Radio

Using the Micrel single-chip RF receiver, this R/C radio and receiver provides an inexpensive "throwaway" product for the increasingly popular class of micro R/C airplanes, which are flown the length of a typical front yard or large living room. The whole device (with the receiver) costs under \$5 to manufacture in any kind of quantity. This is a welcome improvement from the \$150 or so that it costs for conventional high-power four channel radios that are currently used for micro aircraft.

Mark Antonelli

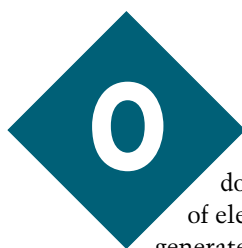
Oxnard, CA — jamesc@venturalink.net

FEATURE ARTICLE

Tom Napier

Applications of PN Sequences

This month, Tom takes us into the world of testing data transmission systems. He says his subject, ideal pseudo-random number sequences, proved “infinitely intriguing,” so get ready for some very interesting information.



One of the paradoxical constructs of electronics can be generated with just a shift register and an exclusive OR gate. With the right connections, a shift register having N stages can generate a sequence of ones and zeros that repeats every $2^N - 1$ bits. Apart from this repetition, the sequence passes every standard test for randomness. This makes it the ideal test input for data transmission systems.

Pseudo-random number (PN) sequences are widely used in electronic devices where an apparently random but actually predictable signal is required. The PN sequence emulates all possible normal data inputs, but because an exact copy of it can be independently generated at the receiver, any discrepancies between the input and output can be immediately detected. This is the basis of so-called Bit Error Rate Testers (BERTs), which detect bit errors occurring in data transmission.

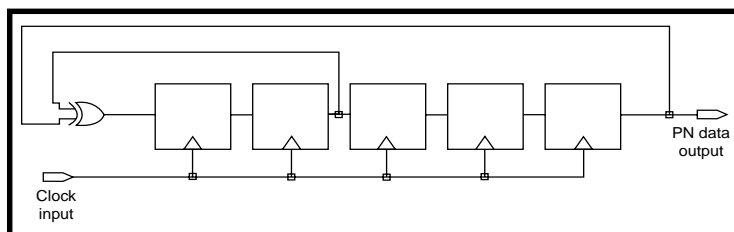
Another use is to make a data signal more random. A binary datastream can be exclusive ORed with a PN sequence before transmission. This may be done to make interception more difficult, to eliminate long strings of ones and zeros that might disrupt the receiving equipment, or to spread the data spectrum over a wider band. In spread spectrum systems such as ubiquitous GPS receivers, PN sequences phase modulate the carrier frequency.

All these applications depend on the receiving equipment being able to generate a matching PN sequence to remove the encoding applied at the transmitter. If the same bit sequence is exclusive ORed twice with a data signal, the result is the original data. There is a tricky way of doing this.

PN sequences also have uses in analog systems. If a digital PN sequence is low-pass filtered, the result is a good approximation of Gaussian analog noise. Such noise has many advantages over that generated by conventional methods. Because the input signal is digital, the output amplitude is well-defined. Low-pass filtering a normal noise source reduces its amplitude in proportion to the square root of the filter bandwidth. With digitally generated noise, changing the clock rate changes the noise bandwidth without affecting the amplitude. This makes it easier to generate test signals having a well-defined signal-to-noise ratio.

Classically generating analog noise from a PN sequence gave a noise bandwidth about 5% of the clock rate, a few megahertz at most. Some years ago a product I was designing needed a source of fixed amplitude Gaussian noise with a bandwidth up to 100 MHz. By using a FIR filter to compensate for the roll-off of the PN sequence spectrum, I came up with a noise generator with a useful bandwidth of 50% of the clock rate. One

Figure 1—A five-stage shift register and an exclusive OR gate generate a 31-bit pseudo-random sequence.



digital circuit board replaced a boxful of analog noise generators, filters, amplifiers, and RMS level meters. (This design was granted a U.S. patent.)

PROPERTIES OF PN SEQUENCES

Here are some of the fascinating properties of PN sequences. First, ones and zeros occur in almost equal numbers. Why almost? Well, the complete cycle of bits contains an odd number, there is always one more 1 bit than 0 bits. With a long enough sequence, this discrepancy becomes irrelevant; whichever bit you look at, the next bit is almost equally likely to be a one or zero. If you look at groups of successive bits, all possible patterns occur equally as often. This means that you can convert groups of bits into analog form and get a random voltage that takes all possible values equally often.

A particular case of this "window" property arises when you look at the complete shift register. In the course of $2^N - 1$ clock periods, this cycles through all possible N-bit states except the all-zeros state. Each state is equal to, say, a right shift of the last state, except that one bit is shifted out and is replaced by a new bit. The new bit is the exclusive OR of the bit shifted out coupled with another bit of the shift register. And because there are $2^N - 1$ distinct states, a PN shift register is sometimes used as a pulse counter, or clock divider.

A PN sequence has a perfect autocorrelation function. If you exclusive OR two identical time-shifted PN sequences and compute the average, you get a negligible output unless the time shift is zero. This makes PN encoded data suitable for measuring the time delay in a signal path. By shifting the reference sequence until you get a match, you can tell how far away the transmitter is even if the received signal is noisy. There is a range ambiguity if the time delay is longer than

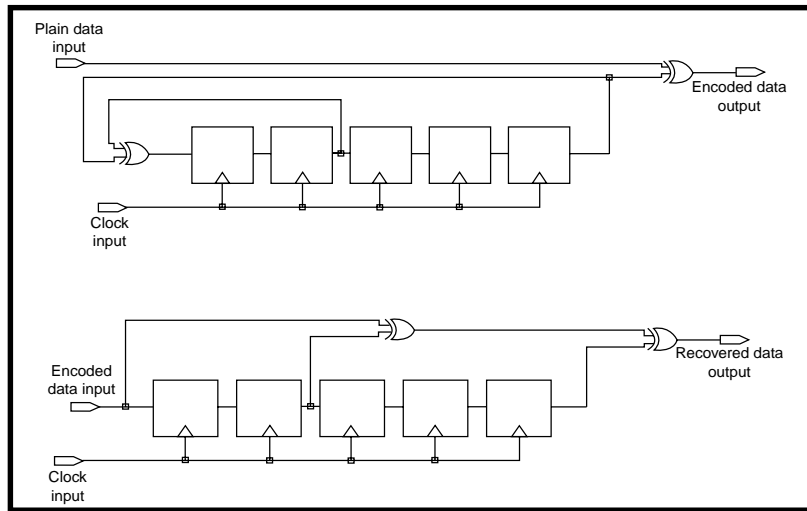


Figure 2—Two shift registers (one with feedback and one without) randomize and standardize a binary signal.

the sequence, but this rarely matters because there is no limit to the length of the PN sequence you can use.

Another interesting property is that when you exclusive OR a PN sequence with a delayed version of itself, you get an identical sequence with yet a different delay. It is easy to tap a time delayed datastream if you need a delay shorter than the shift register length. However, you can generate a sequence delayed by any number of clocks up to the $2^N - 1$ length of the cycle. All you need is to exclusive OR the right set of shift register bits.

I used this property to generate effectively uncorrelated sequences without extra hardware by tapping into the same sequence at points millions of bits apart. I found an algorithm to compute the correct set of taps for any given delay, but I'm unable to find an algorithm that translates taps into time delays, other than by brute cycling through the sequence. Does anyone know of one?

BUILDING A PN GENERATOR

I presented an oversimplification of the PN sequence generator's construction. Not all $2^N - 1$ sequences can be generated by exclusive ORing the outputs from two shift register stages. For some values of N, more than two stages must be exclusive ORed to generate the maximum length sequence. For a given N, there are usually several combinations of taps that generate

different maximal length sequences. In some cases these are time reversals of each other. When testing equipment, you must be sure which sequence was used at the transmitting end.

With telemetry there is a tendency to standardize easily generated sequence lengths, partly because a shift register can be clocked faster

when only one gate delay is needed. Table 1 lists the sequences shorter than 2^{24} , which can be generated with a two-input exclusive OR gate. In each case, the inputs of the exclusive OR gate are connected to both the last shift register stage being used and the stage given by the tap number (counting from the input end). The output of the exclusive OR gate goes to the shift register's input pin.

Figure 1 shows the five-stage shift register, which generates the 31-bit sequence:

1000010101110110001111100110100

The 74HC164 8-bit shift register makes a convenient building block for experimenting with PN sequences.

When a PN generator is turned on, the shift register must be loaded with a nonzero value, otherwise it will generate zeros forever. When operating, it generates an all-ones pattern once per cycle. This can be used to synchronize other equipment, such as an oscilloscope, to the data pattern.

In practice, it is better to detect the $N - 1$ zeros state, which also occurs only once per cycle. Detecting this state is easy when the shift register is constructed from ECL parts, because you can use the wired OR capability of ECL to make a negative AND gate.

In other logic families, the easiest solution is to use an up counter, which is reset by any one. This counter reaches its $N - 1$ state only

when $N - 1$ zeros occur. The same counter can test for N zeros, the "hung" state, and restart the shift register if it occurs. In one application, I combined the sync and restart functions by parallel loading the shift register with the next valid state every time the sync state was reached, thus anticipating any all-zeros state. [1]

Having modulated a datastream by a PN sequence, you might imagine that synchronizing the decoding sequence at the receiver would be a tough job. However, you only have to feed the data into a shift register that has an exclusive OR gate connected to the same taps as in the transmitter (with no feedback). The output from an exclusive OR gate connected to this signal and to the input will then be the original data.

Figure 2 shows the transmitter and receiver connections using the same five-stage register as Figure 1. There is one disadvantage. Because each output bit is the exclusive OR of three input bits, any bit error generated in

Stages	Tap	Length
2	1	3
3	1	7
4	1	15
5	2	31
6	1	63
7	1	127
9	4	511
10	3	102
11	2	2047
15	1	32767
17	3	131071
18	7	262143
20	3	1048575
21	2	2097151
22	1	4194303
23	5	8388607

Table 1—The shift register taps for maximal length sequences are listed here.

transmission will cause three output bits to be in error.

BIT ERROR RATE TESTING

One major application of PN sequences is the measurement of bit errors in data transmission systems. This is done either to confirm that the equipment meets the maker's specification or to monitor its performance

to detect potential failures. The job is simple if the equipment can be taken off-line for testing. A suitable PN sequence is supplied in place of the usual data source. The data recovered at the other end of the system is compared with the original PN sequence and any discrepancies are noted.

If the equipment must be tested in operation, then either PN data must be injected in an unused channel or some regularity of the data (e.g., a frame synchronization pattern) must be used to detect errors.

A BERT generates a binary datastream at a user-selected clock rate. The data pattern can be selected from several PN sequence lengths. There are many ways of formatting binary data. These range from non-return to zero through bi-phase code, which carries its own clock, to modified-Miller encoding, which minimizes the signal's mean DC component.

The output amplitude also can be switched to conform to various standard levels. Some specialized BERTs also generate the framing information

required when testing packet-switched networks and telephone systems.

A BERT contains a receiving, decoding, and error counting section that works at the same clock rate but is otherwise independent of the transmitter section. Sometimes both ends of a transmission path are at the same location and a single BERT can both transmit and receive data. More often the input and output are many miles apart and two BERTs are used.

LOCKING TO THE DATA

The receive section of the BERT has two mechanisms to handle transmission time delays. It can synchronize to the clock phase of the input signal and it can synchronize to the PN data pattern. A BERT may be used to measure error rates as high as 20% so it needs a more sophisticated synchronization system than just explained. However, the essentials are the same.

Because in a BERT the data is a raw PN sequence, passing the incoming data through a decoding shift register should generate all ones or all zeros. If

Listing 1—This code fragment for a PIC16C57 uses two 8-bit registers, low and high, to emulate a 15-stage shift register. TOP = 6, the farthest left bit in the shift register. TAP = 0, the other input to the XOR operation. This code is general. It can be extended to any shift register length and tap position by using more registers and changing TAP and TOP as required.

```
CLRF    TEMP,1      ;Clear test register
BTFSF   HIGH,TOP    ;Test leftmost SR bit
BSF     TEMP,TAP    ;Set bit in tap bit position
MOVF    LOW,0       ;Fetch SR byte
XORWF   TEMP,1      ;XOR tap bit with leftmost bit
BCF     3,0         ;Clear carry bit
BTFSF   TEMP,TAP    ;Test XOR output
BSF     3,0         ;Set carry
RLF     LOW,1       ;Shift carry into register
RLF     HIGH,1      ;Complete 16 bit shift
```

there are few enough bit errors, an error-free sequence longer than the generating shift register occurs often. This seed is detected by counting output zeros and is transferred to a second shift register. There it generates a clean and completely independent PN sequence that acts as a reference for the input signal.

Any discrepancy is a bit error and is counted. After some fixed number of input bits have been received, the error count is stopped. The result is

divided by the number of bits received and displayed as the bit error rate.

The input and output can get out of sync. For example, equipment under test may drop one bit. Because of the autocorrelation property of the PN sequence, such a bit slip results in an immediate change in the detected error rate to close to 50%. This alerts the BERT of the need to synchronize to the input signal again.

Error rate testing can be a slow business. Because of the random nature

of bit errors, the accuracy of an error count is roughly the reciprocal of the square root of the total count. To measure the true error rate to about 10%, you need to wait for 100 errors to occur. Because the test specification may require the error rate to be less than one in a billion bits, a full-scale error test may take weeks. At 64 kbps, for example, 100 errors will accumulate in 18 days.

For completeness I'll mention the mathematical notation commonly used to specify the taps on a shift register. A set of taps is listed as a polynomial expression where the exponents of a delay operator (x) specify which taps are exclusive ORed to generate the new input.

The input itself is often denoted as 1, so a polynomial such as $x^5 + x^2 + 1$ represents the taps on the five-stage shift register of Figure 1. The plus signs represent the exclusive OR operator (e.g., $x^n + x^n = 0$). Right and left shifting is represented by multiplying and dividing the expression by x . With this notation, you can prove all kinds of neat results.

SOFTWARE EMULATION

To build a PN data generator, you don't need a hardware shift register. Even the tiniest computer chips can handle shift and exclusive OR functions and, thus, emulate a hardware generator, albeit at a lower speed.

I've programmed PIC chips to generate an FM signal modulated by an external data input. It's easy to add a PN data generator for self-test. I've even built a 64-kbps BERT using a 16C57 chip. This generates both 127- and 32767-bit test patterns using the same algorithm, only the address in the index register changes. Listing 1 shows a code fragment that can generate almost any length of PN sequence.

DISPLAYING PN DATA

Short PN sequences fit on one line of a scope trace and can highlight problems such as baseline shift. If the scope is synchronized to the bit clock rate rather than to the data pattern, it will display an eye pattern. This will reveal whether the transmission path has too

little bandwidth or too much noise or phase shift to transmit reliable data. This is a valuable tool for adjusting data filters and equalizers to optimize the signal-to-noise ratio.

What I've discussed here this month merely scratches the surface of the theory and application of PN sequences. They are to electronics what the Mandelbrot set is to mathematics, simple to generate yet infinitely intriguing. 📄

Tom Napier was a principal engineer in the Signal Recovery Group of the Aydin Corporation for eight years. There he developed better ways of receiving signals from a spacecraft and designed a BERT with a built-in noise generator to test data receivers. Now, he is a consultant and writer.

REFERENCE

- [1] Tom Napier "Ideas for Design 520: Self-Starting Data Generator," *Electronic Design*, July 24, 1995.

Mike Baptiste

The “S” is for Speed

Breathe New Life into Your
Z180 Designs

Whether your design needs a few software tweaks or more extensive change, upgrading to the 'S180 will put you in the fast lane. Mike shows us that it's a valuable addition to the Z180 family, one that makes HCS-II users smile.



If you keep up with the latest happenings at Zilog, you probably think you picked up a 1999 issue of *Circuit Cellar* (Zilog released the Z8S180 CPU over a year ago). However, this column is timely because I recently upgraded the HCS-II controller to use the new Z8S180 CPU, and *Circuit Cellar's* Driven to Design Contest is based on the Z180 core. I'm going to stray from my home automation focus, but fear not, it's only temporary.

The Z8S180 is an enhanced Z180 core offering faster execution, more robust serial UARTs, more efficient DMA, more power-saving modes, and lower EMI. Before ordering some for your existing Z180 designs, read on. This upgrade may not be as easy as popping out your old Z180 and putting in an 'S180, especially if you want to get the most out of your upgrade.

UNDER THE HOOD

The 'S180 adds a few new elements to the existing Z180 feature set. For example, it expands the number of

power-down modes available for applications requiring power conservation. The new Standby mode effectively shuts down all of the CPU (including the oscillator) until an external Reset, bus request (BUSREQ), or interrupt is received. This total shutdown reduces power consumption to approximately 50 μ A for the CPU.

Because the oscillator is turned off, exiting Standby mode requires time for the oscillator to stabilize. The 'S180 will wait 2^{17} clock cycles before resuming operation. The recovery signal must remain asserted for all of the recovery time or the device will stay in Standby mode. This can be in several milliseconds, even at 20 MHz!

If the long recovery time is an issue and your design uses an external oscillator device, the 'S180 allows you to enter a quick recovery Standby. This only takes 64 clock cycles to recover the system.

The other power-saving mode is Idle. This keeps the oscillator running to allow for a quick recovery, but turns off Clkout so any other parts in your design tied to Clkout will stop, thus reducing overall power consumption.

The 'S180 adds an option to reduce the internal oscillator drive, which can reduce EMI. The default oscillator drive is already reduced by 30% compared to the Z180. If you enable this option (bit 6 of the Clock Multiplier Register (0x1E)), remember that the reduction is in addition to the 30% reduction in the default drive. If you have a series resistor in your oscillator, you may want to remove it as a result of the reduced default drive.

The I/O pins are now auto-latches, which can prevent excessive supply currents because of floating inputs. By using a latch circuit on each pin, the state of an input is maintained, even if the external source is removed. The same applies when an output is turned off using /OE. The state will be maintained, but the pin state will be overridden by any external assertion because the auto-latches are weak (i.e., maximum of 10 μ A of leakage current).

One problem you may encounter is if a pin is pulled to ground via a pull-down resistor. If the auto-latch is latched high (VDD) when the signal is

removed (i.e., in a tri-state situation), the input may not flip to GND. This occurs if the external pull-down is not strong enough and the auto-latch and external pull-down then form a voltage divider. See the Resources for information on ensuring that pull-downs are strong enough. The Z8S180 errata states that pull-down resistors must be no weaker than 15 kΩ to ensure that they can overpower the auto-latch.

The 'S180 also enhances the DMA support by allowing the DMA channels to be linked. This reduces the involvement of the CPU during DMA transfers and allows for faster data transfer. If the DMA channels are tied to the same high-speed device, you can switch between them. While one channel is busy getting data, you can program the next buffer address and byte count into the inactive DMA channel without waiting for the first transfer to finish.

When the current transfer is over, the CPU automatically switches to the inactive channel and immediately starts the transfer that's already programmed. In effect, you can set up the second transfer anytime during the first transfer instead of having to interrupt the CPU after the first transfer to set up the second.

By now you're probably wondering why you started to read this. So far the improvements are pretty tame. But I like to save the best for last.

SPEED BOOST

The Z180 CPU uses an internal clock divider that takes the oscillator frequency and divides it in half. Just like yesterday's PCs, the operating frequency of your device may not be enough for today's needs. That was the case with the HCS-II.

A number of users had large control programs for their HCS-II, and running these would cause the HCS-II to slow down because XPRESS programs are essentially one big loop of IF statements. The interrupt-driven, second-based timers would become less accurate because the CPU was swamped with stuff to do (program evaluation, network traffic, etc).

The HCS-II boards have 18.432-MHz crystals, so they run at 9.216 MHz internally. I needed a simple way

to boost the performance of new and existing HCS-II boards. This meant no hardware changes beyond chip swapping.

The 'S180 CPU can operate at full-oscillator frequency or at the original half frequency if you don't want to change the internal operating speed. It also can use a clock multiplier so you can double the external oscillator frequency. In my case, doubling the frequency by eliminating the divider was the ticket. The 'S180 cannot go beyond 33 MHz internally so I couldn't use the clock doubler even if I wanted to. But the clock doubler comes in handy for devices with slow oscillators (i.e., less than 16 MHz) and even in new designs so that cheaper, low-frequency crystals can be used.

Doubling the clock speed of the HCS-II was going to help a lot. But it wouldn't be as easy as dropping in the 'S180. The existing hardware and software would need to be altered. The HCS-II was designed at a time when static RAM that was faster than 100 ns was too expensive. The firmware used a single wait state for memory access, which further slowed down execution speed, but also allowed slower, less expensive RAM chips to be used.

Note that to remove the wait state at 18.432 MHz, I had to replace the RAM and EPROM with devices that were at least as fast as 100 ns or more. Because inexpensive 70-ns chips are available, I decided to go with these to provide a comfortable timing margin (though the 70-ns DIP RAM can be hard to come by as a result of supply shortages). I had to replace the EPROM anyway because of software changes, so adding a couple of RAM chips to the upgrade was not a big deal.

When increasing system speed, one area of concern is the response time of I/O devices. The 'S180 allows up to four wait states for I/O, just like the Z180. If you double the internal clock, your I/O will be twice as fast, unless you can double the I/O wait states. Because the HCS-II was already at

Register name	Hex address
ASCI0 extension control (AXEXT0)	12h
ASCI1 extension control (ASEXT1)	13h
ASCI0 time constant low	1Ah
ASCI0 time constant high	1Bh
ASCI1 time constant low	1Ch
ASCI1 time constant high	1Dh
Clock multiplier	1Eh
CPU control register	1Fh
DMA I/O address register Ch 1B	2Dh

Table 1—The new 'S180 CPU adds a number of new control registers to handle the new added features. These registers default to values that make the 'S180 act just like a Z180.

three wait states at 9.216 MHz for I/O, I could only increase them to four with the 'S180.

I needed six to keep the timing the same, so I feared I wouldn't be able to double the internal speed. However, extensive onsite testing showed four wait states would be tolerable with the existing peripheral set. Of course, as Murphy's Law would have it, a few users with many I/O cards (and thus, a heavily loaded bus) have reported some instability, which was resolved by upgrading the 74LSxx glue logic to the faster 74Fxx family.

Doubling the clock speed of any program will probably require a number of software changes. If your system uses a real-time OS like the HCS-II, you'll have to tweak the low-level code. Hopefully your RTOS utilizes a ticks variable, which specifies the number of clock ticks per unit of time. If it does, it can just be doubled and your OS timing should stay the same. If it doesn't, the changes can be trickier and, because they are such low-level changes, the risk increases.

The other timing issue is delay loops. I usually try not to hardcode timers in the code, but sometimes it happens anyway. The toughest part of this upgrade was going through all 200 pages of the HCS-II firmware and tracking down all the delay loops that were time-critical. Changing them usually meant doubling the timer delay value, which was easy. However, finding them was tricky. Thank goodness the code was well commented!

SUBTLE SERIAL

The 'S180 contains some subtle enhancements to the two onboard serial ports (ASCI0/1). The ports now

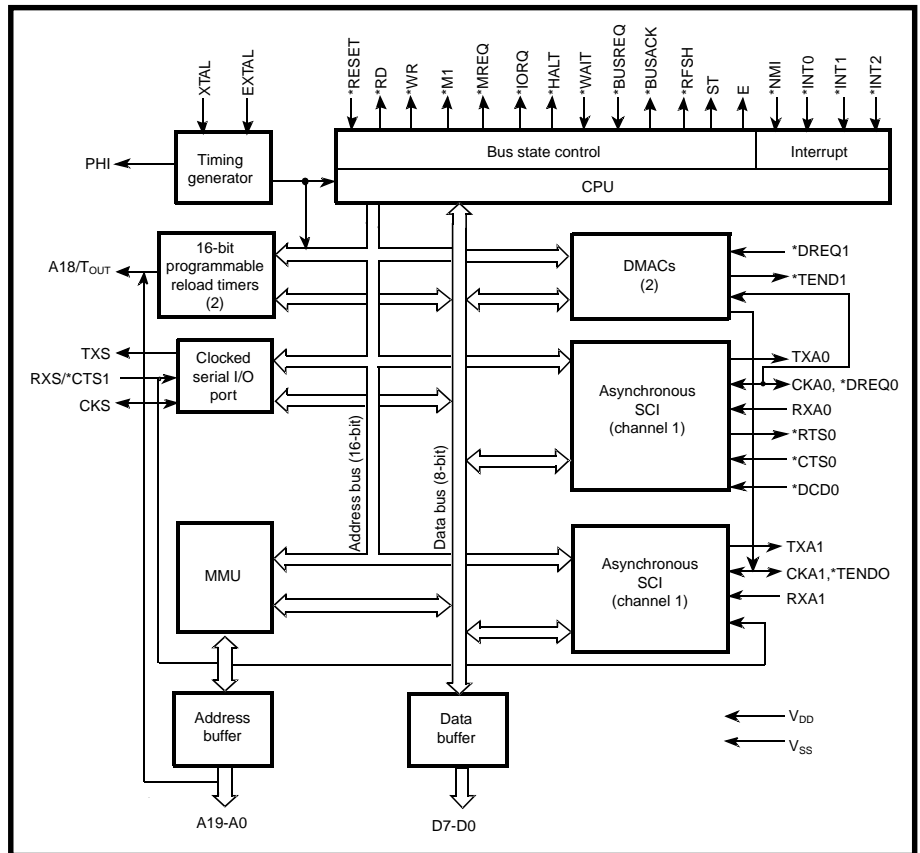


Figure 1—Here's the functional block diagram of the Z8S180. There's not much different here because the enhancements are inside.

have a better baud rate generator (BRG) that will allow data transfer rates up to 512 kbps. The Z180-style BRG is still available if you don't need the faster BRG speeds.

The transmit and receive FIFO buffers have been doubled. These FIFO buffers sit between the data access registers (TDRx and RDRx) and the shift registers. Each serial port has a receive FIFO buffer that can store four characters before an overflow occurs. The transmit buffer now has a two-character FIFO.

The expanded FIFOs can boost communication performance because your program can use fewer interrupts to service a serial I/O operation. Because the FIFOs are "under" the data access registers, it probably appears that these changes won't affect your current code. You might think you'd only have to change your code if you want to reduce the interrupt interval in your serial routine.

However, if your device uses an RS-485 network, the bigger transmit buffer will probably cause problems. Chances are your code waits 1.04 ms

after the last character is loaded into the shift register (not the TDRx register), and then turns off the RS-485 transmit enable line. However, because the transmit buffer is now two characters deep, you'll have to wait twice as long to disable the RS-485 transmit drivers to give the double FIFO a chance to empty.

A common symptom of this problem is that your network packets seem to drop their last character or the last character is corrupted. Because this delay loop needs to be accurate, the HCS-II disables interrupts during the delay, which is now 2.08 ms. I wish there was an interrupt flag indicating when the shift register was empty because that's a lot of wasted CPU time in a multiple task system. Of course, Microchip PICs are the same way, so Zilog is not alone in the unfriendly RS-485 camp.

WHOOPS!

There are a few things the 'S180 does that you may not expect. These are covered in an errata published by Zilog in 1999.

The first issue relates to the RXS/CTS1 input for the second ASCII port. Though not explicitly stated, it appears that this pin is not connected internally, or a problem with the RXS/CTS1 input multiplexer prevents it from being properly read. Thus, the RXS/CTS1 input will always read as a 0 (via bit 5 of the CNTLB1 register when bit 2 of STAT1 is set to 1).

If your device relies on CTS1 for handshaking, you will have to reroute it to an unused input. Of course, if you rely on CTS1 to automatically handshake serial transmission, you'll have to change your code because CTS1 = 1 prevents Transmit Data Register Empty (TDRE) from going high when the transmit buffer is ready for more data. You'll simply have to add an extra software check for the new CTS1 input instead of relying on the CPU to do it for you internally via TDRE.

Another change is how the 'S180 handles receive overruns. The Z180 ASCIs will resume serial reception after an overrun as soon as the software reads a received character. The 'S180 disables ASCII reception after an overrun and it can only be enabled again by writing a 0 to the ERF bit in the CNTLA register.

This difference can have drastic results in upgraded systems for obvious reasons. If, for some reason, your device allowed incoming data to overrun, the worst case would be to have corrupted data at that point in time. Now, your device will stop receiving data! Of course, this ensures that your device will have a robust serial error handling routine. That's a good thing, but only if you're prepared to do the programming for it.

WRAP UP

The 'S180 is a nice addition to the Z180 family. Moving to an 'S180 can be as simple as a few software tweaks, though some situations may require more substantial changes. At the very least, you'll want to review the new registers to see if you need to set any (see Table 1). But, you may be able to get away with the default values.

Upgrading the HCS-II with the 'S180 CPU had a dramatic effect. XPRESS second timers were much

more accurate because interrupt delays were reduced. Even though the RS-485 network stayed at 9600 bps, the serial performance improved. The HCS-II can send out the next network packet much faster after it finishes receiving and processing data from a previous query. You could see it speed up by simply watching the traffic on a serial terminal. The packets were going out with less lag time between them.

XPRESS programs run faster, which means that inputs can be sensed faster.

An XPRESS program is just a big loop of IF statements. Since I doubled the clock and eliminated the memory wait state, the XPRESS program loops much faster, so input state changes are sensed earlier. This was observed by toggling an output (which drove an LED) on every XPRESS loop. Combined with a large XPRESS program, the LED would blink noticeably.

The upgrade is a hit with HCS-II users because all they had to do was replace the RAM, EPROM, CPU, and a few logic chips. Your situation may vary, but the upgrade shows that a chip like the 'S180 can extend the life of existing designs and give customers a better sense of your product's value.■

Mike Baptiste earned a B.S. in Computer Systems Engineering from Rensselaer in 1992. After a seven-year "hiatus" working for a large telecommunications company, he returned to his roots working with embedded processors in home automation. He can be reached at baptiste@cc-concepts.com.

RESOURCES

Migrating from Z80180 to Z8S180, Zilog, Inc., www.zilog.com/pdfs/z180/migrate.pdf.

Z8S180/Z8L180 Product Specification, Zilog, Inc., www.zilog.com/pdfs/z180/.

SOURCE

Z8S180 CPU

Zilog, Inc
(408) 558-8500
Fax: (408) 558-8300
www.zilog.com

SILICON UPDATE

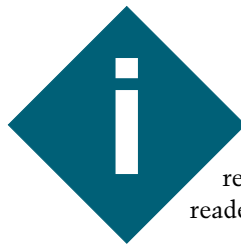
Tom Cantrell

eZ Does It



Years ago, Zilog went toe to toe

with the heavyweights, taking its knocks, but still swingin'. Result: the single-chip successor to the esteemed Z80, proof that underdogs can rise in Rocky style.



sometimes receive letters from readers:

I too have watched, waited, and wondered what Zilog will do next. I have been in this industry for more than 20 years and seen my share of smoke and mirrors. The eZ80 and the TCP/IP stack appear to be just so much smoke and mirrors. The Zilog reps and factory people have told me that they have absolutely no idea what the eZ80 is, when it will be available, or what it does.

One of the Zilog Application Engineers told me marketing puts that "stuff" on the web. Just thought you might let me know if Zilog gave you the impression that this product will ever be released and that the TCP/IP stack won't be some brain-dead stack that we see on other micro-controller products.

I think if the product is real, the stack is free, and the parts work, Zilog could sell a ton of them.

Keep up the great columns.

*Regards,
Jeff*

It's true that I have a soft spot in my heart for Zilog. It goes all the way back 25 years to two guys (Faggin and Ungermann), hunched over a living room table, who had the audacity to take on mighty Intel and Motorola. They didn't know it couldn't be done, so they just did it! I like that.

Of course, as the underdog, Zilog got knocked around and, yes, managed to trip on their shoelaces more than a few times. Though bloodied, they never went down for the count or threw in the towel. I like that too.

Nevertheless, I try not to let nostalgia and emotion cloud my judgment. The fact is, given their tumultuous history, Jeff's skepticism is astute and more than justified.

My reply to him was that he is right. The ball is in Zilog's court to live up to the promises. The jury's still out, but there's no doubt that Franklin is a heavy hitter and Zilog has executed some interesting moves (like buying Seattle Silicon and Production Languages). As for eZ80, I've met living, breathing engineers who are working on it, and I've seen internal specs that are arguably beyond those of a mere marketing exercise.

My response went on to say that if and when Zilog delivers a chip that works and a free (or low-cost) high-quality TCP/IP stack, it will definitely be worth a look. Until then, the saga continues.

It's been almost a year since the new Zilog started talking about a successor to the venerable Z80. It's

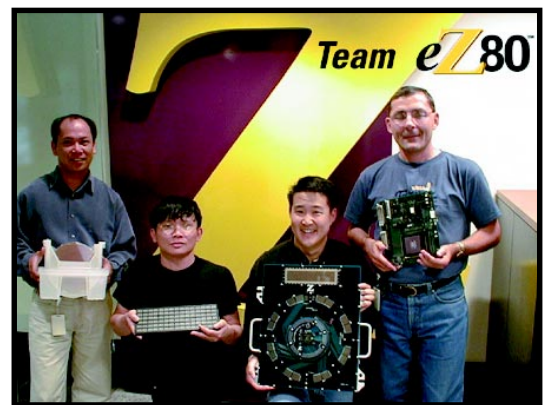


Photo 1—Real engineers with real eZ80s (left to right): Mario Visperas, Test Development Engineer, with eZ80 8" wafers; Albert Le, Test Development Engineer, with a tray of eZ80 chips; Danny Chi, eZ80 Business Line Manager, with an eZ80 ATE load board; Adam Tucholski, Applications Engineer, with a prototype eZ80 EV board.

	A	F
BCU	B	C
DEU	D	E
HLU	H	L
Main register set		
	A'	F'
BCU'	B'	C'
DEU'	D'	E'
HLU'	H'	L'
Alternate register set		
I	R	Mbase
IX		
IY		
SPS		
SPL		
PC		

Table 1—A 24-bit Z80? Why ask why? Just do it!

time for Zilog to give the PR department a rest and let their silicon do the talking (see Photo 1).

ONCE MORE, WITH FEELING

The challenge for the eZ80 is to add enough whizzy stuff to excite customers and keep up with the Joneses without losing the essence of the predecessor's popularity.

Consider the new Volkswagen Beetle, which represents a successful upgrade strategy. VW retained the unique, quirky, fun feeling of the original Bug which still makes me smile when I see one, new or old.

My first car, a hand-me-down '63 Beetle, was unique, quirky, and fun too. Unfortunately, it also had a motor more befitting a lawnmower, a rubber-band shifter, a heater that wouldn't do these days, and the list goes on. This just won't do today. That's why you can get a new Beetle with 150 HP!

But don't overdo it because it's easy to go one feature too far and lose touch with whatever made the original product successful. The line (beyond which, an upgrade path becomes an upgrade cliff in the customers' eyes) isn't always obvious.

The New Coke fiasco is a classic (pardon the pun) example of an upgrade gone awry. Apparently, the major feature of Coke isn't just what it tastes like (New Coke was proven preferable by scientific taste tests), but also the fact that consumers seem to like knowing that it's exactly the same as the Coke they had yesterday.

OLD AND NEW

Zilog did a good job of balancing the past and future with the eZ80, but judge for yourself (see Figure 1).

From a historical perspective, the eZ80 retains critical links with the Z80, most notably binary object code compatibility. Not that you can plug in your old ROMs (initialization and timing details differ), but this definitely lends a warm and fuzzy feeling for designers who are already familiar with the Z80. This is a wise move because upgrade strategies that require the use of translators just never seem to cut it.

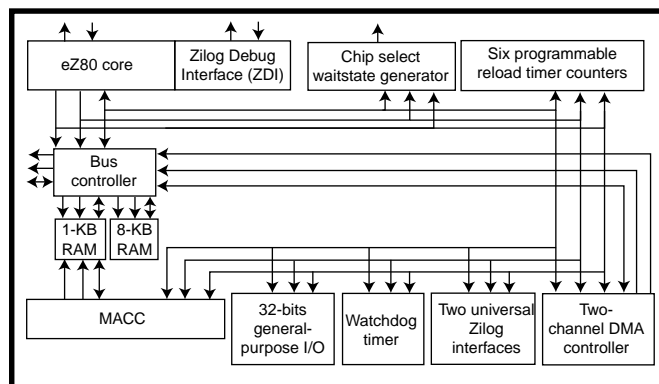
But like that 150-horsepower Beetle, the eZ80 has a turbo motor. With a three-stage pipeline design targeting 50 MHz and beyond, Zilog claims four times the throughput of a Z80 running at the same clock rate. That puts the eZ80 in rather exclusive company with high-performance 8- and 16-bit chips, presuming the external program memory can deliver the goods (i.e., fast access time). Do note that the internal RAM offers single-cycle access and can be used for both code and data.

Because external program memory is required, the eZ80 has four chip select outputs, each independently programmable as memory (on 64-KB boundaries) or I/O (on 16-byte boundaries), with 0 to 7 wait states.

Sure, the architecture is long in the tooth, but isn't that the case with other popular 8-bit chips like the '51, 68xx, and PIC? Actually, a Z80 refresher highlights the fact that it has aged gracefully and still stacks up well against old and new competitors.

CISCy? Yes, indeed, but not any stranger than most of the other popular 8-bit chips. I'd say it's one of the easier chips to program in assembler and also reasonably well-suited for C. For instance, there's a real stack and ways to get to it, unlike lesser chips that cramp your style with a tiny fixed stack just for return addresses.

Figure 1—At first glance, the eZ80 looks like a typical high-integration 8-bit micro, but there are plenty of surprises under the hood.



Similarly, the Z80's relatively sophisticated multimode interrupt scheme is carried forward. Look close enough and you can discern the foreshadowing of your highfalutin PC's interrupt scheme, which goes all the way back to the 8080 and Z80 days. In addition to the simple-minded fixed vectoring of most MCUs, the eZ80 also supports external vector or instruction fetch, enabling dynamic and adaptive interrupt service.

PECK-O-PERIPHS

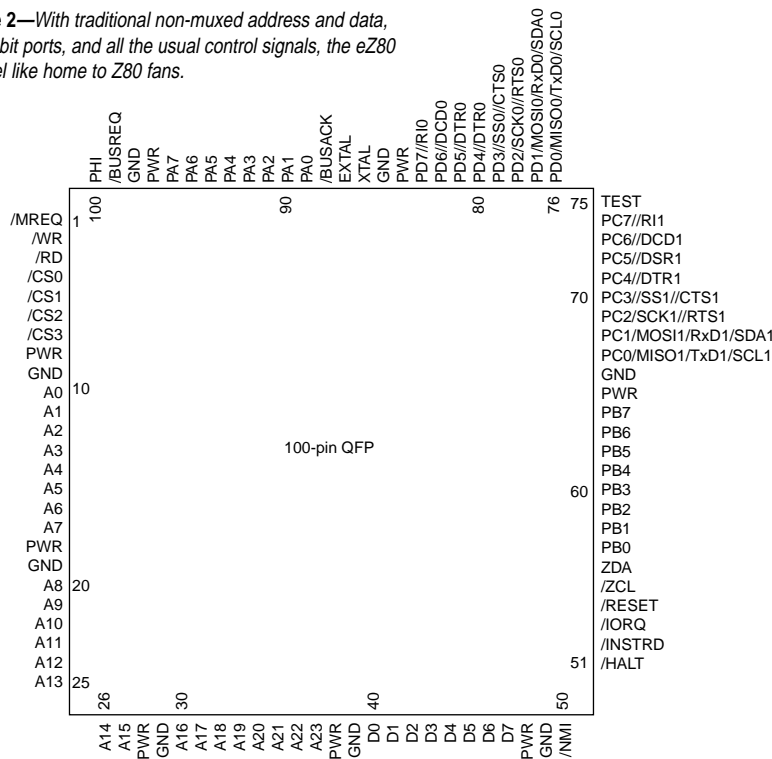
Part of the Z80's popularity had less to do with the CPU and more to do with the other peripheral chips you could buy to work with it, such as the SIO (serial I/O), CTC (counter/timer), PIO (parallel I/O), and so forth. The eZ80, like the '180 before it, packs the equivalent of yesterday's multi-chip Z80 board onto a single chip.

The eZ80 has 32 pins available for I/O. These are shown in Figure 2 organized as four 8-bit ports (Ports A–D). Notice that while there is an *NMI (Non-Maskable Interrupt) pin, there aren't any regular *INT requests. This seems odd.

It turns out that taking advantage of the aforementioned vectored interrupt scheme, every one of the 32 I/O pins can serve as an interrupt request, with programmable edge or level sense and polarity. Each pin has its own vector for instant response, far faster than figuring it out in software.

Two of the ports (C and D) serve optional double-duty as serial I/O for the rapid-fire UZI (Universal Zilog Interface). This is an improvement over the simpler UARTs of the SIO and '180. You can check out this improvement in Figure 4.

Figure 2—With traditional non-muxed address and data, four 8-bit ports, and all the usual control signals, the eZ80 will feel like home to Z80 fans.



The UART, a clone of the NS16550 found in PCs, is high-end compared to those found on most micros, incorporating 16-byte transmit and receive FIFOs, break generation and detection, complete error detection (parity, overrun, and framing), and a full complement of modem control lines.

These days, clocked serial ports are all the rage, extremely handy for small carry-on items such as an EEPROM. The UZI supports both popular standards, I²C and SPI, with a programmable selection of phase, polarity, and pin assignment.

There's also a dedicated clocked serial interface for debug known as ZDI (Zilog Debug Interface). With debug features such as memory and register modify, single-step, breakpoints, and more, this is one more example of the trend towards built-in minimal pin count debugging.

On the timer/counter front, the eZ80 incorporates simple 16-bit units, making up for a lack of sophistication (i.e., no dedicated pins, PWM modes, etc.) with quantity (six channels) and speed (up to half the system clock). Each timer has its own prescaler (clock divided by 2, 4, 8, or 16) and interrupt vector.

There's also a dedicated watchdog timer with four programmable timeout periods— 2^{18} , 2^{22} , 2^{25} , or 2^{27} system clock cycles. Timeout can be programmed to generate either a RESET or *NMI. At RESET, the watchdog comes up disabled and, after embedded, stays that way until the next RESET. Note that writing A5h followed by 5Ah to a register keeps the watchdog at bay.

Finally, there's a two-channel DMA controller, another big-ticket item you won't find on many MCUs. It offers two modes of operation, burst (full block transferred after bus is acquired) and cycle-steal (control relinquished back to the CPU between bytes).

I could tell from the documentation that the DMAC does not have any external requests and only supports memory-to-memory transfers, not memory-to-I/O. This rules out use with internal I/O functions, most notably the UZI, and calls for a bit of cleverness when designing in an external I/O chip. For example, a DMA-targeted I/O chip must be memory-mapped, not I/O-mapped. You can fake an external DMA request line for burst transfers using an interrupt on one of the PIO pins.

BEYOND 64 KB

So far, I've described a kind of Z80 on steroids, which is pretty cool but doesn't really stand out from the crowd. To up the ante and excitement level, the eZ80 goes a bit (make that 16 MB) further.

Traditionally, 8-bit chips have been held within a 64-KB barrier. Yes, there are all manner of hack-arounds, from a simple page register to the fancy MMU on the Z180, but all require juggling 64-KB chunks within the larger address space. It can work if your program and data are easily partitioned, but if not, it can get pretty ugly. Conventional wisdom says you should probably go with a chip that has a true 32-bit programming model.

For those of you who can get by with the 64-KB chunk approach, the eZ80 includes a simple 8-bit page register (MBASE) for the high-order address bits (A16–A23). At reset, MBASE is zeroed so the eZ80 comes up like the good-old 64-KB Z80.

I originally reported that the eZ80 would include the '180 MMU, but that's not the way it turned out. On reflection, it's no great loss because, for all its 1-MB bluster, the '180 was still a 64-KB chip under the hood.

Instead, Zilog came up with a clever scheme for going beyond 64 KB, and the short and sweet of it is that the eZ80 is really a 24-bit processor!

It's quite simple. There's an ADL (Address and Data Long) bit, and when it's set, anything a regular Z80 does with 16 bits, the eZ80 does with 24 (see Table 1). The 16-bit registers of the Z80 (BC, DE, HL, IX, IY, and the PC) become 24-bit registers, and there's an extra 24-bit stack pointer (SPL) in addition to the historical 16-bit one (was called SP, now SPS).

The ADL bit can't be toggled by software because it instantly changes the interpretation of the PC (i.e., MBASE plus 16-bit PC versus true 24-bit PC). Instead, ADL is controlled by appending a suffix to all instructions that change the PC, including JP, CALL, RET, and RST.

Similarly, instruction suffixes define whether a memory address or immediate data is 16 or 24 bits, and whether the operation is performed on

16- or 24-bit data, or both. For example, the Z80 instruction LD rr, (nnnn) loads 16 bits into register pair rr from the 16-bit address nnnn. On the eZ80, there are four permutations:

- LD.SIS rr, (nnnn)—16-bit data at 16-bit address (same as Z80)
- LD.SIL rr, (nnnnnn)—16-bit data at 24-bit address
- LD.LIS rr, (nnnn)—24-bit data at 16-bit address
- LD.LIL rr, (nnnnnn)—24-bit data at 24-bit address

The instruction suffixes generate prefix bytes. Fortunately, the Z80 has four defacto spare op-codes in the form of instructions like LD B,B (i.e., load a register with itself). In the unlikely case that the eZ80 assembler encounters one of these instructions, it generates a warning and replaces it with a regular NOP.

The prefix bytes work at all times, independent of the ADL setting. An assembler pseudo-op, .assume adl = 1 ; or 0, tells the assembler which kind of code to generate. The documentation observes that, “The programmer is, of course, responsible for ensuring that this source-file setting matches the state of the hardware ADL mode bit when the code is executed.” Read it and heed it.

It’s straightforward if your software is either 64-KB and Z80 mode or 24-bit mode in its entirety (i.e., ADL is either 0 or 1, but never changes). It gets more complicated if the software includes both types (i.e., ADL changes state during operation), as would be the case when combining legacy code with new code. There is a mixed ADL bit, instructions to tweak it, and a set of rules to follow, mainly related to

which stack pointer (SPL or SPS) to use and how much (two or three bytes) to stack and unstack. If you just want to drop in a self-contained legacy routine, it may involve little more than tweaking a single RET instruction to restore the caller’s ADL state. I advise against going overboard with mixed mode because it’s easy to get tripped up by spaghetti code (e.g., multiple exits) and dueling stacks.

MAC ACK

At the beginning of this article, the eZ80 was an 8-bit chip. Now you’ve discovered that it’s really 24 bits. Hmm....

Well, hold on to your hats. Thanks to a built-in Multiply and Accumulate unit (MAC), the eZ80 is really 32 bits, or 40 bits for that matter. In fact, it’s not just a processor, it’s also a DSP.

The MAC, as shown in Figure 3, consists of twin 256 × 16 X and Y RAMs feeding a multiplier with a 32-bit result added to a 40-bit accumulator, all in a single clock cycle. This new Beetle burns rubber!

The X and Y RAMs are dual-ported for connection to the eZ80 core as a block of memory that is 1 KB.

Also note that control of the MAC is in the hands of a set of 16 I/O registers that define the calculation to be performed (i.e., X and Y addresses and length) and hold the accumulator initial value and result.

Actually, there are two complete sets of MAC registers that operate in a ping-pong fashion, allowing software to set up the next calculation while the current one is in progress, hiding the overhead. Making things even faster and easier, there’s a new variant (OTI2R) of the traditional block I/O instruction that increments both the memory and I/O address. Just set up pointers to a 16-byte calculation descriptor in memory and the MAC I/O base address and blast away.

All things considered, the eZ80 is quite interesting, don’t you think? Of course, this first chip is presumably only the start, but it’s fun to speculate. A bunch of on-chip, high-speed flash memory is the most obvious

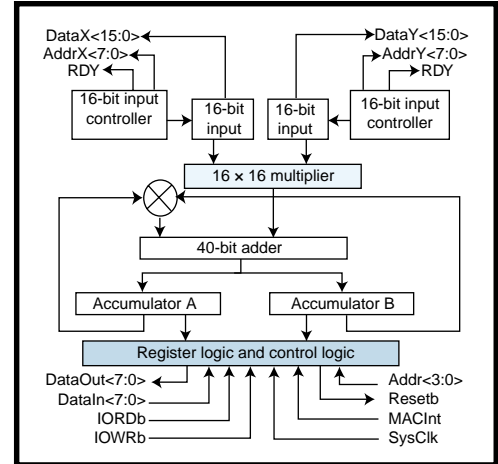


Figure 3—Soft-modem anyone? With a single-cycle MAC unit, that’s one of the obvious DSP-like eZ80 applications.

upgrade. Toss in a TCP/IP stack and soft modem or Ethernet and you’d have a nifty embedded web gadget. The true “beyond 64-KB” capabilities offered by the 24-bit ADL scheme make for some intriguing possibilities on the software front as well.

CAPTAIN COMEBACK?

Even though eZ80 silicon actually exists, I’d say it’s too soon to pop the bubbly. It could be a long road getting from the lab to the warehouse.

However it turns out, I think the eZ80 folks should be congratulated. The design has style, flair, and dare I say, soul that harkens back to the damn-the-torpedo days of glory. It’s also got all the horsepower and features of the latest and greatest.

Not to get too overwrought about it, but I’d say the eZ80 could mark a renewal of leadership by Zilog. If all goes well, at the least, the eZ80 will let Zilog take some skeletons out of the closet and bury them for good!

Captain Zilog may be an old-timer in silicon years, but he isn’t giving up or holding back. I like that. 📧

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

SOURCE

eZ80 8-bit microprocessor
Zilog, Inc.
(408) 558-8500
Fax: (408) 558-8300

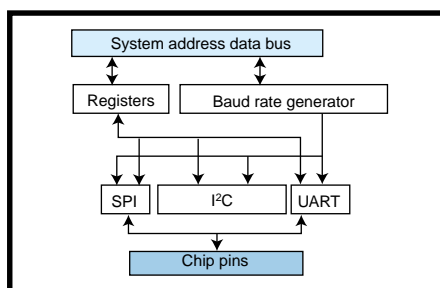


Figure 4—The eZ80 UZI (Universal Zilog Interface) shoots bits in three popular serial formats.

Megawatt Castles Made of Sand

FROM THE BENCH

Jeff Bachiochi

Exploring the Solar Cell

As a child, I had no idea that my castles were made out of the second most abundant element on earth. Nor could I have envisioned that the technologies of today would be based on the same substance and could potentially be the answer to our reliance on fossil fuels.

PHOTOELECTRIC EFFECT

Isaac Newton's Law of Conservation of Energy states that "energy can neither be created nor destroyed." Solar power can't be described as the creation of electrical energy from the sun. Solar power uses a photoelectric cell's ability to change the sun's energetic photons into electron-hole pairs that can be drained as electrical energy. The sun's energy supplies virtually all of the energy that powers the earth's ecology. The majority of this energy is in the visible portion of the electromagnetic spectrum. Photons are particles of the sun's energy. Photons traveling at different wavelengths have different amounts of energy, which is measured in electron volts (eV). The energy level of photons in the entire spectrum of sunlight spans from infrared (energy level of -0.5 eV) to ultraviolet (energy level of -2.9 eV). Is it coincidence that the binding energy on an electron to its nucleus in an atom is on the order of 1 eV?

ATOMS

We know that all matter on earth is made of atoms. For the basis of this discussion I will only talk about three

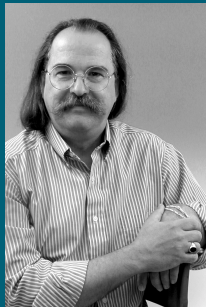


wouldn't mind living by the ocean.

I have fond memories of vacations at the shore.

I still enjoy reading a good book and sitting for hours in the sunshine while listening to the seagulls. The endless rush of saltiness foaming along the shore is soothing to the soul.

I can walk at water's edge for miles, especially after a storm (see Photo 1). That's when the ocean throws back the treasures it has collected. My children have inherited my fascination with creating sand sculptures, fully realizing that when they return the next day, nature will have wiped the canvas clean.



Nature brings out the best in Jeff. It also

brings out the best in energy production. Could the key ingredient for boundless amounts of energy be in the sandcastles of children?

Photovoltaic timeline

1839	Discovery of PV effect by Edmond Becquerel
1873	Selenium as a PV ~ 1% efficient
1918	First single silicon crystal grown
1954	Cadmium as a PV, efficiency of PV silicon up to 6%
1955	First commercial PV product (cost of energy \$1500/W)
1958	First PV-powered satellite (Vanguard I, 8-year life)
1960	Laboratory efficiency up to 14%
1972	Nigerian village receives PV system for educational TV
1979	Arizona's Papago Indian Reservation receives 3.5-kW village system
1980	Utah's Natural Bridges National Monument receives a 105.6-kW system
1982	Worldwide production exceeds 9.3 MW
1983	Laboratory efficiency up to 18%
1991	Solar Energy Research Institute becomes U.S. Department of Energy's National Renewable Energy Laboratory (NREL)
1993	NREL opens Solar Energy Research Facility
1995	Laboratory efficiency up to ~21%
1996	DoE announces National Center for Photovoltaics

Table 1—Man's study of the PV effect dates back more than 150 years!

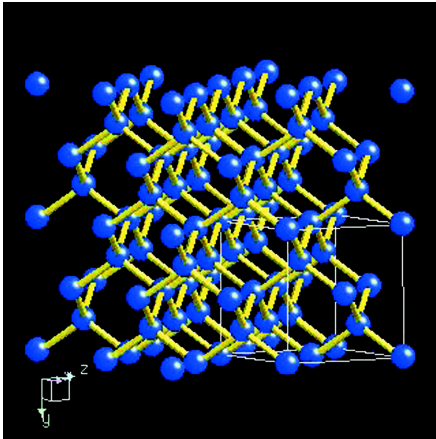


Figure 1—Each silicon atom has a link (shown as a strut between balls) with four additional atoms. Notice the nonrandom pattern of the silicon crystal-line structure.

of the basic particles that make up an atom. Protons (positively charged particles) and neutrons (uncharged particles) form the nucleus and the majority of the mass of all atoms. Orbiting around the nucleus are electrons (negatively charged particles). Although the particles of atoms can have charges, they are neutral in charge as a unit because the number of electrons equals the number of protons, and the equal and opposite charges cancel each other out. Because there is a difference in charge between the protons (in the nucleus) and the electrons flying free, they are captured (attracted) in an orbit around the nucleus. An orbiting electron's distance from the nucleus is dependent on the energy of the electron. As the energy level of the electron goes up, its orbit is further from the nucleus. However, electrons are fixed into orbits (energy level zones) forming shells. Each orbit can only hold a maximum number of electrons in each shell (i.e., 2-8-18-32). Additionally, each shell can have groups of electrons in sub-orbits, but that won't be part of this discussion.

The atomic number is the quantity of protons in each atom of that particular element. Atoms have no charge, therefore there must be an electron orbiting for every proton in the nucleus. When there are eight electrons orbiting in the outermost shell (or two if we're talking about the elements with a single shell), the element is happy because this is a stable condition. When this condition

doesn't exist, atoms lose, gain, or share electrons trying to satisfy the stable condition.

SILICON

The silicon atom has an atomic number of 14. The 14 electrons orbit the nucleus in three levels, or shells. The first shell is complete with only two electrons. The second shell is complete with eight electrons. The third and outer shell has four electrons. It would like to have eight electrons (or none). If each silicon atom shares electrons with four other atoms, its outer shell becomes content and a pure silicon crystalline structure is formed. (I wanted to make a 3-D model of a silicon crystal to help me visualize the structure, but the graphic shown in Figure 1 [1] was much better than any of my gumdrop and toothpick sculptures.)

The sharing of electrons is called covalent bonding. If an atom is content, then it's a bad conductor because there are no free electrons to travel through the element. The traveling of electrons through the element is called the current flow.

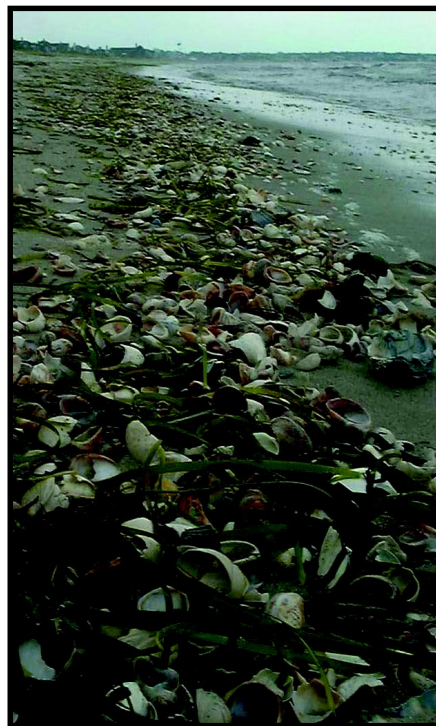


Photo 1—Recreation for the masses. Both the raw and refined states of silicon give us pleasure. Will silicon also provide the key ingredient for unlimited energy production?

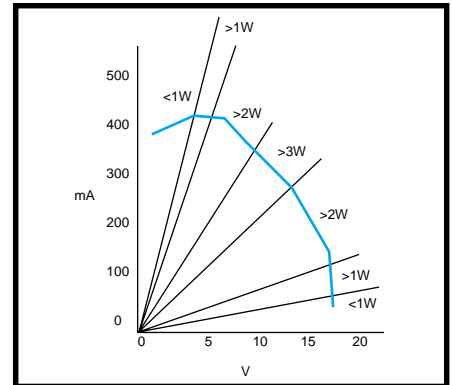


Figure 2—I plotted the output voltage (along the X axis) versus the output current (along the Y axis) to get an idea of where the sweet spot of the solar array is. At either ends of the graph, available power drops off quickly. The highest power output is at the center of the graph.

If silicon is a bad conductor, how can it be made to support current flow? This can be accomplished by adding impurities (either atoms with fewer or more electrons in their outer shell) to the silicon without affecting the crystalline structure.

Boron has only three electrons in its outer shell. When Boron is added (doped) into a silicon crystal, there are areas where the covalent bonding is incomplete. Because the Boron atom has only three electrons, there remains an empty slot where an electron bond would normally be in the silicon crystal. This slightly unhappy state creates free holes, and thus, is a better conductor. This lack of electrons (one for every Boron atom) designates the substance as a p-type material.

Phosphorus, on the other hand, has five electrons in its outer shell. When silicon is doped with phosphorus, there are areas where an extra electron is hanging out that's not needed for the covalent bonding of the silicon crystal. Again, this unhappy state is caused by free electrons, creating a better conductor. These extra electrons (one for every Phosphorus atom) designate the substance as an n-type material. The n-type and p-type materials carry no negative or positive charge at this time because all the atoms have an electron for every proton.

When the two materials are joined, the free electrons from the n-type material are drawn across the junction into the material with a lack of electrons (or an abundance of holes), and

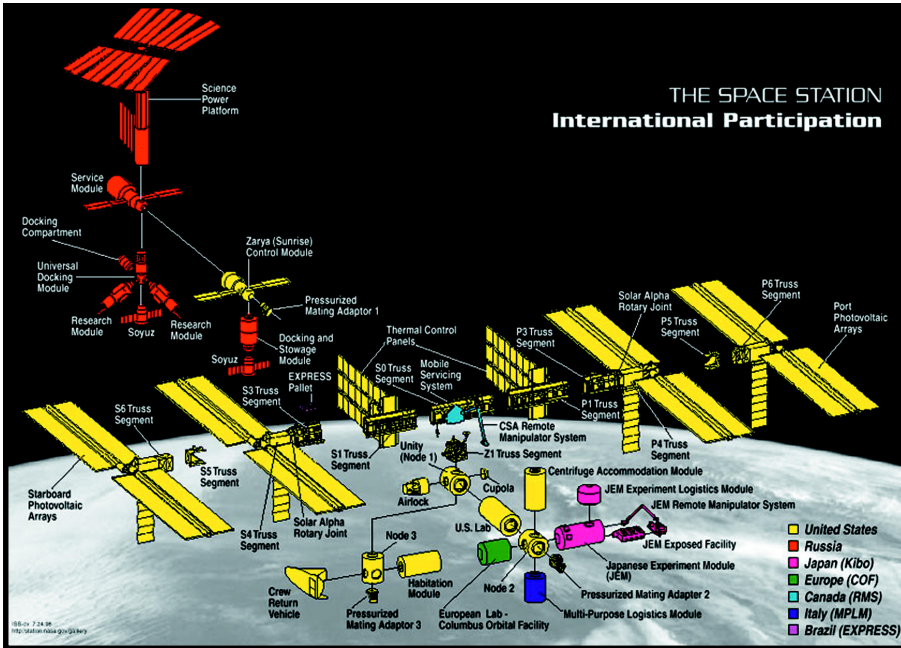


Figure 3—Unlike the space shuttle, which uses fuel cells for energy, the International Space Station gets its power from PV modules. Space Station Freedom has the advantage of converting photons from the sun without the absorption losses of our atmosphere.

vice versa. This creates a barrier at the junction of the two materials, a transition region if you will, where there are no extra holes or electrons because

they have filled in each other's deficiencies. In this area, the material is less conductive because there are no mobile charge carriers (holes or elec-

trons). At equilibrium, the transition region, which spans the junction, has a charge or potential across it. The n-type material has lost electrons to the p-type material and now has an imbalance of protons to electrons (fewer electrons), so it becomes positively charged. At the same time, the p-type material has taken on some electrons from the n-type material and now has an imbalance of protons to electrons (fewer protons than electrons), so it becomes negatively charged.

If you take a battery or current source and apply its potential across a p-type material, current (holes) will flow from the battery's positive side through the p-type material and back into the battery's negative side (holes being the majority carrier in the p-type material). The same will happen across n-type material (in this case, the electrons are the majority carrier). The polarity doesn't matter with either of these materials on their own.

When the pn junction is established, things change. The pn junction is considered forward biased if the

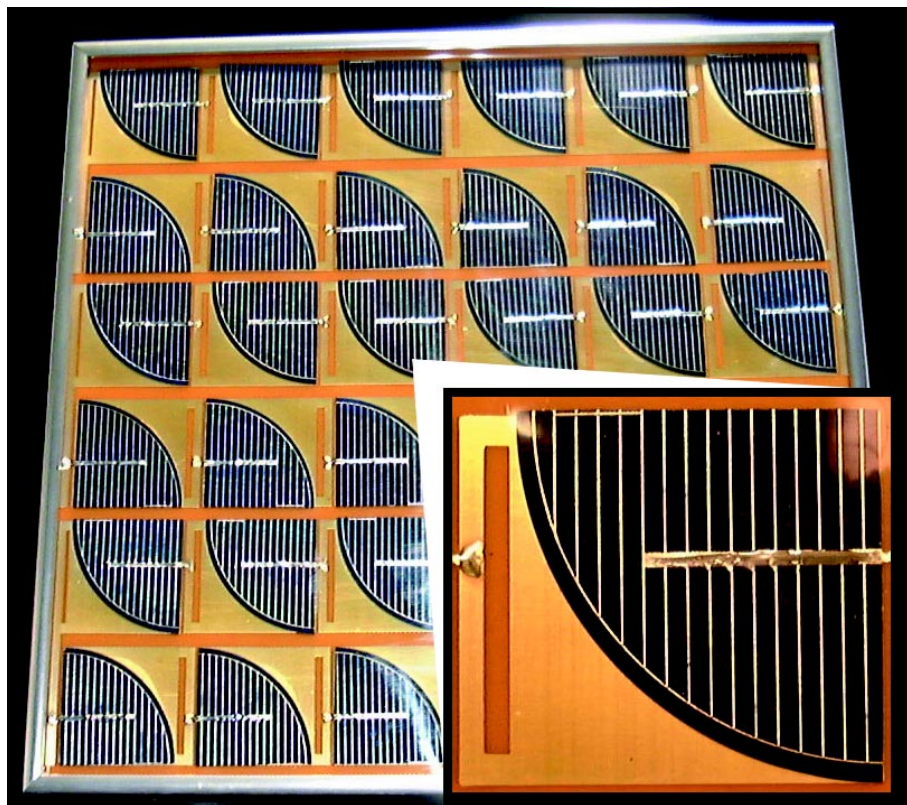


Photo 2—This PV array is made up of 35 quarter-round solar cells connected in series. It can produce 300 mA of current to charge a 12-V battery. Over the period of one day, that could add up to a few amp hours.

negative side of the battery is connected to the n-type material and the positive side of the battery is connected to the p-type material. When forward biased, electrons flow into the n-type connection and up to the transition region where there is a deficiency of electrons (positive charge). Meanwhile, holes are flowing into the p-type connection filling in a deficiency of holes (negative charge). This reduces the potential across the barrier to allow electrons and holes across it.

If the battery connections to the junction are reversed, it's considered reversed biased. In this case, electrons flow into the p-type connection and are repelled by an abundance of electrons (negative charge).

At the same time, holes are flowing into the n-type connection and are repelled by an abundance of holes (positive charge). And this increases the potential across the barrier, preventing the flow of electrons and holes across it.

BAND GAP ENERGY

For an electron to break out of its shell and be free of its nucleus bonding, it must gain energy. The amount of energy is measured in electron volts (about 1.1 eV for silicon). An external battery can supply this energy, or it may come from the sun.

Photon particles (photonic energy from the sun) of the visible part of the electromagnetic spectrum are at different energy levels. If a photon entering your pn material has sufficient energy (more than 1.1 eV in silicon), it will jar an electron free of its bond. The free electron, or hole, gets drawn across the junction creating a current flow into an externally-connected device. The eV current product is power. The optimum eV for a cell is 1.4 eV. With a higher eV, fewer photons from the sun's energy have sufficient energy to free electrons; this limits current flow, reducing the maximum potential power. Below 1.4 eV, more free electrons are knocked loose, but the potential across the cell is also reduced along with the maximum potential power. So, how does light get into a diode junction?

SOLAR CELL MAKE UP

Materials are used to maximize absorption of photons and minimize their reflection and recombination in order to maximize the electron conduction through the junction. The grown monocrystalline silicon has the highest energy conversion factor of any solar cell. It's also the most expensive to produce. Grown crystals are sliced into wafers. The wafers are doped with the appropriate substances to create the p-type and n-type areas within the wafer. However, covering the layers with metal to create an electrical connection for current flow also covers the surface where light needs to penetrate. To allow some light to enter, holes must be placed in the metal surface at least on one side. A grid pattern (or screen) of metal is used to minimize the shadowing as much as possible (see Photo 2).

AMORPHOUS SILICON

Amorphous silicon doesn't form a crystalline structure and has dangling bonds where another atom would normally be attached. Hydrogen is used to plug up these bonds by sharing an electron with the silicon. In this material, an intrinsic layer (of un-doped amorphous silicon) is sandwiched between the ultra-thin p-type outer layer and the thin n-type inner layer and the eV field extends through the p-i-n layers to induce electron movement.

POLYCRYSTALLINE THIN FILMS

Thin film technology, a spin-off of the semiconductor industry, is much easier to manufacture. This doesn't require the growing slicing and treating of a crystalline ingot to produce a homojunction (same base material used in both doped layers). Thin film materials are deposited in thin layers on a glass or plastic substrate and produce a heterojunction (different base material used in each layer). These include amorphous silicon, gallium arsenide, copper indium diselenide, or cadmium telluride. Creating solar cells with multiple layers (each uses material with different band gap potential) is an attempt to capture more than the pitiful 15–25% of photon energy in a silicon cell (up to 35%).

Thin films can't be doped to form n-type and p-type layers. Instead it uses the layering of different materials to provide the extra electrons (or holes). The top windowed n-type layer must be thin enough and have a band gap of more than 2.8 eV to let through all of the available light. The lower absorbing p-type layer must have high absorption (for high current) and a suitable band gap (for high voltage) and be a few microns in thickness.

When Gallium arsenide (GaAs) is used for manufacturing solar cells, it has the advantage of having an optimum band gap of 1.43 eV, although the higher power conversion comes with higher costs. GaAs withstands high temperatures, allowing it to work with concentrators.

A concentrator is a system where sunlight is gathered from a large area and focused on the solar cell, increasing light intensity and heat. GaAs cells need only be a few microns thick. They are also resistant to radiation damage.

Tradeoffs for today's solar cells are:

- Monocrystalline (single crystal cell)—excellent conversion (~14%) with high manufacturing costs.
- Polycrystalline—good conversion (~12%) with lower production costs.
- Amorphous—fair conversion (~9%) with the lowest production cost and shorter life span.

SOLAR CELL LOSES

Why are solar cells so inefficient? First of all, ~55% of the light energy is wasted. Photons with less energy than the band gap voltage won't be able to free an electron. Those photons with a greater amount of energy will give up what's needed and carry off the remaining energy. The materials used in manufacturing the cell will waste another large chunk of energy along with the cover glass coating and the contact grid (used in the crystalline silicon cell).

You can expect a maximum intensity of about 1 kW/m² from the sun. Those photons actually hitting a solar cell can be reflected, absorbed, or pass right through the substance. Cells are specified under a standard set of conditions: when the source is at 45° to

the surface and the temperature equals 25° C, the insolation (measure of light energy per area) equals some number of watts per square meter.

PV power is normally reduced with higher temperatures (~0.5% for each degree C), however this doesn't affect the expected lifetime of the solar cell. Increasing the latitude places more of the atmosphere in the solar energy's path, so a PV unit in Arizona will produce ~50% more power than the same one in Massachusetts.

Solar cells are most often used in parallel to increase current and in series to increase voltage. The connection of multiple cells forms an array. Even arrays of cells can be connected both in parallel and in series to form larger PV modules.

Plotting the voltage and current output of a cell demonstrates the relationship between eV and current. Open-circuit measurement of a PV cell has a maximum voltage (~0.5 V) and minimum current (0 V). Short circuit measurement of a PV cell would have minimum voltage and maximum current. This graph shows how the cell will perform under different load conditions. Figure 2 is a graph I plotted from the solar cell array. Photo 2 illustrates how the surface conductors reduce the overall transparent area of a photovoltaic cell.

ORIENTATION AND SYSTEM SIZE

For those of us who are living in the Northern Hemisphere, the best fixed orientation of solar modules for maximum absorption year-round is south facing at an angle equal to the area's latitude. Many solar systems (including those for solar hot water) track the sun's position to absorb maximum radiation throughout the day. As they say in real estate, it's "location, location, location." The same goes for the placement of solar cells because the shading of even one cell in an array or module can cut its output in half.

For those of you who are interested in solar power for your home, it's best to check meteorological data and household demand to determine a worst-case scenario (average intensity in your area and peak usage in your home). Realizing that the sun is not

out 24 hours a day brings up the question of storage. Deep-cycle batteries store energy for when you can't draw directly off of the PV system. Although many appliances run off low voltage DC, they are made to run from an AC source. So, unless you rip out the power supply from these appliances, they will not run on the DC available from the PV system. You have the choice of altering your appliances or altering your power source. DC to AC converters can change your DC battery source into usable AC (just like the battery backup unit on your PC.) Lead acid batteries are most widely used for storage, but can be discharged to only 40–50% and NiCads, although more expensive, can be totally discharged. Today's charge controller provides the best charging characteristics for whatever battery style you might choose to use. However, you must be prepared for some big numbers because today's cost for a PV system without batteries is around \$9 per watt. Of course, the cost shrinks as the production goes up.

Stay tuned next month when I discuss how solar cells helped keep me online while spending the week communing with nature. Also, for a photovoltaic timeline, see Table 1.

THE FINAL FRONTIER

The best possible place for PV cells is not on earth. Because the atmosphere plays a large part in how much solar energy falls on a PV cell, you'd be better off without it (as far as converting photons into free electrons). For this reason, the International Space Station (ISS) shown in Figure 3 will have an advantage in PV conversion over ground-based PV systems. Above our atmosphere, photonic energy is allowed to fall directly on the PV cell without losing energy to the atmosphere. Space Station Freedom's electric power systems (EPS) use PV arrays and NiH₂ (nickel/hydrogen) batteries. Orbiting the earth, the ISS spends about one third of each 97 min. behind earth's shadow. Because PV arrays won't produce energy in the shadows, the onboard batteries store and release

energy to assure station power 24 hours a day, 7 days a week.

Although it only produces about 2 kW of power at this stage of construction, when fully assembled in ~2006, ISS will be capable of producing 110 kW of PV power, using about 250,000 solar cells. To maximize energy conversion, each of the main PV arrays will tilt or rotate to track the sun. This is enough energy for 55 average households. Covering more area than a football field, the ISS will be easily seen from earth as it passes overhead. The majority of this area will be the PV arrays, as shown in Photo 2.

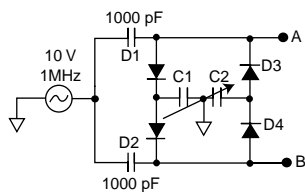
Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on Circuit Cellar's engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.

REFERENCE

[1] M. Winter, *WebElements*, www.webelements.com.

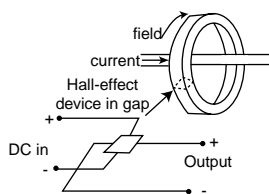
CIRCUIT CELLAR Test Your EQ

Problem 1—The following circuit is used to read a tilt sensor that is implemented as a differential capacitor—when the sensor is tilted, C1 increases while C2 decreases (or vice-versa). Both capacitors vary between 4 and 30 pF.



The voltage differential between points A and B varies with the position of the sensor. How would you analyze the operation of this circuit?

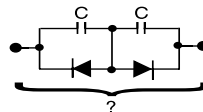
Problem 3—One way to sense DC current is to wrap an iron core around a wire, cut a gap into it and put a Hall-effect sensor in the gap, as shown below. However, Hall-effect sensors are notoriously temperature-sensitive and nonlinear. What's a simple way to deal with these issues?



Problem 2—If you put two identical nonpolarized capacitors back-to-back as shown, the overall capacitance is half of one of the capacitors.



If you put two electrolytic capacitors back-to-back with parallel diodes as shown below, what is the overall capacitance?



Problem 4—OK, so you got a real deal on a used Bridgeport milling machine, and just got it set up in your shop. However, you just noticed that the motor on it requires 3-phase power, and all you have in your neighborhood is single-phase power. Is there anything simple you can do, short of trying to find a new single-phase motor that will fit the Bridgeport?

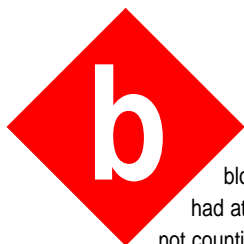
What's your EQ?—The answers and 4 additional questions and answers are posted at www.circuitcellar.com.

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

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

PRIORITY INTERRUPT

Upgrade Math



Buying a PC has always been confusing. I guess we can credit the combination of massive marketing efforts and bloated operating systems for conditioning us to think that we have to upgrade our PC every 12–18 months. We've had at least seven generations from the venerable old PC, to the '286, '386, '486, P1, PII, PIII, and soon the P4. That's not counting competing brands and same-generation clock speed increases.

Don't get me wrong, for CPU-speed fanatics who design video processing software, a 5% increase in processing speed is worth getting excited about. They're also willing to pay virtually any price to get it. If you are a CPU-junkie (I'm a car fanatic so I'm not throwing rocks), forget everything I say here and enjoy the rest of the magazine. For the rest of us, however, it might be worth a little price-performance calculation.

Last week we needed to buy three more desktop systems for the office. They weren't for server applications or anything strenuous, they were simple office desktops tied to the in-house LAN for doing word processing, spreadsheets, and a little web surfing (DSL through the LAN). Unfortunately for the staff, I asked what we needed and got involved rather than just saying go buy them.

There was a time when increased performance was a necessity just to keep up with software feature-creep. I remember my first Windows machine. It was a '486DX-25 running Windows 3.0. I had upgraded from '386SX-16 running DOS. According to the Dhrystone MIPS comparison of these machines, the '486 was 9.3 times faster. I had a few different '486 versions before I got my first Pentium, a Gateway P90. The P90 was benchmarked as 5.6 times faster than the '486DX-25 (52 times faster than the '386).

I can definitely say that the P90 was faster, but certainly not 50 times faster on the latest versions of the same software. Word ran a couple times faster. Of course, it was moving a lot more megabytes of feature-bloat now. And strangely, it also seemed there was now a hard drive access with virtually every keystroke. As with most of that generation, it had to run faster just to stay even.

It took buying my first PII before I started seeing some real horsepower. As for the PCs I've had since, I only remember gradual improvements. The truth of the matter is that CPUs today might contain millions of transistors but, most of the time (and for most of us) they are executing wait loops. The average PC's "useable performance" peaked a while ago. For the mainstream office/small business user of today there is little benefit to endless increases in CPU power. The bottlenecks in PC performance these days are virtually never with CPU speed. Ninety percent of the time we sit in front of our monitor, waiting for something to happen.

Even with high-bandwidth Internet connections, I doubt a 1-GHz PIII loads a webpage any faster than one running at 500 MHz. The time taken for opening and closing documents probably won't change either. It typically isn't the CPU's fault. Disk drive accesses and reads are enormously slow when compared to a CPU's capability for processing that data. Although SDRAM and large caches provide the structure for a fast system, inefficient software with too much downward compatibility and operating system resource hogs (like Windows) can negate it.

Deciding what system to buy for the office involved a little re-education. Basically, if you aren't ready for an overkill PIII box (or soon the P4), the only Intel-specific alternative these days is the Celeron. Back when I bought an early PII machine I remember checking out the Celeron. The media described the Celeron as having zero L2 cache and it being a "brain-dead PII."

If that was the history, why was I now seeing so many high-speed Celeron machines offered next to the PIIIs? Well, apparently Intel got the message and they put the 128-MB L2 cache back in and increased its clock speed too. As for being brain-dead—not anymore.

A quick search on "processor benchmarks" provided a little more education about today's market. I'm always looking for a cost-effective solution. It's even better if it turns out to be a bargain. The Intel CPUmark 99 benchmark data gave a value of 45.1 for a 700-MHz Celeron and 63.0 for a Pentium III. That says the PIII is only 40% faster than a Celeron for the same clock speed.

Before I get tons of mail about my crummy math and that I've overlooked things like branch prediction architectures, RISC versus CISC, and an in-depth discussion about integrated caches, this isn't scientific. I consider this to be a minimum resolution comparison at best. There's also the fact that most Celeron systems are 600–700 MHz and PIIIs are being packaged increasingly at 800 MHz and faster with larger hard drives and lots of extras in the box.

Nonetheless, for a simple Office 97 application where the CPU spends most of its time waiting for the user, the choice seemed clearer than I originally thought. Where I live, an 800-MHz PIII system sells for well over \$1000. At the same place, a 600-MHz Celeron system is less than \$500. Considering the expense of some of the other things I've bought in life, people who know me might laugh that I'm going to such extreme consideration over a few bucks. I simply smile back. It's not the savings. It's the thrill of the hunt!

steve.ciarcia@circuitcellar.com

