

www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#116 MARCH 2000

PROGRAMMABLE DEVICES

Firmware Upgrades
Using Flash Memory

State Machine Debugging
on Your PalmOS

FPGA System-On-A-Chip

Applied USB



CIRCUIT CELLAR ONLINE

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

— FEATURES —

Designing a DSP-Based RAS Server

Part 1: RAS Server Background

Shawn Arnold

Learn all you need to know about Remote Access System servers in Shawn's introduction to RAS servers. If you've ever had a need for a DSP-based RAS server, you'll find yourself referring to this article again and again.

Making the SmartPIC Serial Programmer

Duane M. Perkins

When picking a PIC programmer, you may find that there are as many choices as there are PICs. The more versatile it is, the more money you have to lay out, right? Not if you follow Duane's plan. He gives you more bang for your buck with his new serial programmer.

Defects For Sale Revisited

George Novacek

George expounds on Software Reliability. Go with him as he kicks the tires and puts some software through the paces. He'll show you how you can quantify software reliability.

INTERNET PIC[®] 2000 CONTEST

www.circuitcellar.com/pic2000

Deadline is May 1, 2000

WWW.CIRCUITCELLAR.COM/ONLINE
Table of Contents for February 2000

Resource Links

- LVDS (Low Voltage Differential Signaling)
- XML (Extensible Markup Language)

Bob Paddock

Test Your EQ

8 Additional Questions

— COLUMNS —

Considering the Details

Motors: A Lost Art?

Bob Perrin

How would you get identical performance out of 1-hp motors operating at different line frequencies? The answer led Bob to satisfy a driving curiosity of the inner details of motors. If you ever thought you needed to know more about motors, then Bob's got what you need.

Lessons from the Trenches

Embed This PC

Part 2: Emulator and EPROM Basics

George Martin

Continuing with his latest project, George shows us how to use emulators in an embedded '486 project to monitor the design. There are a few options to choose from, he'll show you how to make the best choice.

Silicon Update Online

Twenty Years Ago, Today

Tom Cantrell




Tom takes us back to the dawning of the DSP. Follow him as he marks important moments in DSP development leading up to today's modern DSP chips.

THE ENGINEERS TECH-HELP RESOURCE



Let us help keep your project on track or simplify your design decision. Put your tough technical questions in front of 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&A's to broaden their own intelligence base.

- 10 **Applied USB: A Cookbook Approach**
Robert R. Severson
- 20 **Flash Forward**
Implementing Downloadable Firmware via Flash Memory
Bob Brown and Michal Tamborski
- 26 **Building a RISC System in an FPGA**
Part 1: Tools, Instruction Set, and Datapath
Jan Gray
- 36 **Killing Bugs in Your PalmOS**
Debugging with State Machines
Jeff Stefan
- 56 **Throw Away the Key!**
An iButton Lock System
Mike Baptiste
- 62  **MicroSeries**
Rapid Gratification with FPGA Design
Part 2: Quicker and Better Design
Tom Bishop
- 72  **From the Bench**
In Theory and in Practice
Part 2: Clock Adjustment and Digital Filters
Jeff Bachiochi
- 76  **Silicon Update**
SoC it to Me
Tom Cantrell

Task Manager Rob Walker Hitting the Green	6
New Product News edited by Harv Weiner	8
Test Your EQ	83
Advertiser's Index April Preview	95
Priority Interrupt Steve Ciarcia Y2K Phooey	96

INSIDE ISSUE 116

- 44 **Nouveau PC**
edited by Harv Weiner
- 45 RPC **Real-Time PC**
A Matter of Time
Part 3: Synchronizing a PC to a Time Signal
Ingo Cyliax
- 51 APC **Applied PCs**
Getting the Databoot
Fred Eady

Hitting the Green



Well, it must be March. Just the other day while getting something from the closet, my golf bag managed to fall over and land at my feet. Could it be a sign? If you believe that the shadow of a groundhog can predict when winter will end, then winter should be on its way out. Once winter heads north for the summer, New England enjoys a few weeks of mud and slush, and then May arrives and it's time to hit the links.

The first few rounds of golf each season are always the most enjoyable for me. For one, all those annoying people who spent all winter referring to 15-degree weather as "refreshing" are too busy working in their gardens to carry on about the "refreshing" smell of their shovelfuls of manure. The second reason is because "Boy, am I rusty" is an honest observation (and valid explanation for repeatedly requiring assistance as you search for your Titleist 2 in the underbrush alongside the fairways).

Try implying that you are "rusty" in July or August and, regardless of what they say to your face, your partners will be laughing at you, not with you. Although comments like, "Hey guys, the wild raspberries over in the far left rough on that dogleg to the right should be ready by next week!" will surely impress them with your keen observation of the growth patterns of the local flora and make them forget about scheduling next week's tee time while you're at work.

However, there's a lot to be done before May arrives. As I write this, I have my tickets and schedule set for ESC-Spring in Chicago. I don't recall ever seeing golf highlights from Chicago at the beginning of March, so I guess I can leave the clubs at home for this trip. This will be my first "major" event since getting on the *Circuit Cellar* scorecard. I'm looking forward to meeting those of you I've worked with over the last year or so, and I'm also looking forward to bringing back some quality editorial leads.

I'm not the only one who has a lot to accomplish before May. If you'll recall, there's over \$26,000 in cash and prizes up for grabs in the Internet PIC2000 contest that's sponsored by Microchip and *Circuit Cellar Online*. The deadline is May 1, so it's almost time to put the finishing touches on your PIC Internet connectivity project.

For those of you who have your sights set on the \$5000 grand prize or one of the Sony VAIO laptops that Philips is giving away in the *Circuit Cellar* Design2K contest, you have until June 30 (note the extension) to finish your designs.

And, for those of you who know your PICs and 8051s but live in climates that allow you to play golf all winter, which is why you haven't had time to work on a design contest entry, there's still time to drop me a note about becoming a judge for one of the current design contests. After all, it's probably going to be too warm for golf pretty soon, so why not spend time indoors judging project entries in air-conditioned comfort?

I don't know how reliable the groundhog-shadow theory is, but I'm going to set up my indoor putting cup and start polishing up my stroke. Be sure to check out the latest rules updates on our homepage and start polishing up your project designs because before this summer's over, you might be hitting some "greens" in the form of design contest prize money. It could be the beginning of a Cinderella story....

rob.walker@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Sue Skolnick

MANAGING EDITOR

Rob Walker

CIRCULATION MANAGER

Rose Mansella

TECHNICAL EDITORS

Jennifer Belmonte
Michael Palumbo

CHIEF FINANCIAL OFFICER

Jeannette Ciarcia

WEST COAST EDITOR

Tom Cantrell

CUSTOMER SERVICE

Elaine Johnston

CONTRIBUTING EDITORS

Mike Baptiste Ingo Cyliax
Fred Eady George Martin
Bob Perrin

ART DIRECTOR

KC Zienka

GRAPHIC DESIGNER

Mary Turek

NEW PRODUCTS EDITOR

Harv Weiner

STAFF ENGINEERS

Jeff Bachiochi John Gorsky

PROJECT EDITORS

Steve Bedford
Janice Hughes
Elizabeth Laurençot
James Soussounis
David Tweed

QUIZ MASTERS

Tak Auyeung Benjamin Day
Bob Perrin

EDITORIAL ADVISORY BOARD

Ingo Cyliax Norman Jackson
David Prutchi

Cover photograph Ron Meadows—Meadows Marketing

PRINTED IN THE UNITED STATES

ADVERTISING

ADVERTISING SALES MANAGER

Bobbi Yush
(860) 872-3064

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

ADVERTISING COORDINATOR

Valerie Luster
(860) 875-2199

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

ADVERTISING CLERK

Sally Collins

CONTACTING CIRCUIT CELLAR

SUBSCRIPTIONS:

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

GENERAL INFORMATION:

TELEPHONE: (860) 875-2199 FAX: (860) 871-0411
INTERNET: info@circuitcellar.com, editor@circuitcellar.com, or www.circuitcellar.com
EDITORIAL OFFICES: Editor, Circuit Cellar, 4 Park St., Vernon, CT 06066

AUTHOR CONTACT:

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

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, Canada/Mexico \$55, all other countries \$85. All subscription orders payable in U.S. funds only via VISA, MasterCard, international postal money order, or check drawn on U.S. bank.

Direct subscription orders and subscription-related questions to Circuit Cellar Subscriptions, P.O. Box 698, Holmes, PA 19043-9613 or call (800) 269-6301.

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

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

V.23 MODEM EVALUATION KIT

Scenix has introduced a reference design that provides a complete system solution for a V.23 modem in originate mode. It has been tested in accordance with the FCC standards and is ideal for low-speed data transmission applications such as point-of-sale terminals, automatic teller machines, remote monitoring equipment, alarm systems, and the back-channel function of set-top boxes.

The V.23 modem is compliant with the CCITT V.23 standard and includes DTMF generation and detection, caller-ID and call-progress functions. It replaces external hardware components with virtual peripheral software modules that are loaded into the on-chip flash/EEPROM program memory of a 50-MHz Scenix SX28AC MCU.



In addition to the processor, an RS-232 jack and line interface circuit, line driver (DAA) and RJ-11 jack, crystal oscillator, operational amplifier, resistor-based adjustable hybrid circuit, and a few decoupling capacitors and resistors are all that is required for implementation. A modem offering complete V.23-compliant capabilities can be configured to occupy an area 2" x 3" or smaller on a printed circuit board.

The SX28AC MCU is priced at less than \$3 in volume. A modem evaluation kit that includes additional components is available for

\$89 from the Scenix web site.

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

NEW PRODUCT NEWS

LOW COST DATA ACQUISITION SYSTEM

The **DAS 100** was developed to answer the need for a system that does not require a PC, costs less than \$800, and is easy to use. It offers a simple menu-driven setup user interface, Internet access, and is suitable for a wide range of applications.

Housed in a weatherproof enclosure, the DAS 100 holds up to six option cards that measure DC voltage, DC current, AC voltage, AC current, frequency, and temperature. Three relays on the motherboard can be activated by input channels or by user-defined parameters such as time of day. The system features up to 48 input channels and nonvolatile flash memory for data logging. The flash memory system stores up to 3900 samples and data can be downloaded by modem, direct serial connection, or via removable memory modules to spreadsheet programs.

A PC is required to program parameters and values, but is not required for operation. For remotely monitoring the data in real time, an optional 16 × 4 LCD with menu buttons displays the data.

A 24-MHz 8-bit single-board computer with FIREDOS provides power. The system has 1 MB of fixed flash memory for data and program storage, 64 KB of SRAM, and 8 KB of EEPROM, plus a real-time clock, two RS-232 serial ports, and a 2-bit ADC.

The price for the DAS 100 begins at \$599.95 for the basic system with an optional LCD screen available for \$99.95.

Fire Wind & Rain Technologies LLC
(520) 526-1133
Fax: (520) 527-4664
www.firewindandrain.com



FEATURE ARTICLE

Robert R. Severson

Applied USB: A Cookbook Approach

Developing a project that interfaces to a USB port isn't as straightforward as parallel or serial-port interfacing, but according to Rob, the opportunities are worth the effort. Read on to learn why USB will be the port of the next generation of PCs.



I'm an engineer whose imagination has been fueled by science fiction. When I think of what future technology might be like, I have some pretty grand dreams. I can think up a lot of big-ticket items like faster-than-the-speed-of-light travel, anti-gravity, and androids. All of these represent significant engineering advances.

I expect my personal engineering efforts will be viewed as simple in comparison. If I fell asleep and woke up 100 years from now, I'd expect a laugh from the android listening to the details of my profession. Integrated circuits? Ha!

However, if I woke up in three years, I wouldn't expect that technology would have changed enough to have a great impact on my work. But, if my work were to involve designing peripherals that interface to the PC, this may not be the case. With a current trend to remove so-called legacy interfaces from the PC, three years may very well be the distant future.

OBSOLESCENCE

Some of the computer equipment that I have littered about in my basement predates the programmers and engineers I work with. I always have an old 386/486 DOS PC setup so I can

interface circuits to the serial and parallel ports. I use a cheap I/O card and often use protection circuitry to avoid damaging the PC ports if the circuit that I'm working on goes awry.

I write a C program that runs in plain old DOS to interface to these devices. DOS still gets the job done nicely, but the user interface for these devices is rather archaic. I've used DOS GUI packages to spiff things up a bit, however, they certainly don't allow for rapid development.

So, when I move to a PC that has a graphical OS and a non-'80s development environment, I gain access to tools that help make a spiffy user interface. What I lose, is the easy access to the parallel and serial ports I had with my old machine.

Sure, there are ways around this. If you program in Visual Basic, you can use one of several publicly available dynamic link libraries to directly access the serial and parallel port hardware. This is a great solution. A simple write to the parallel port data lines, as I've done on the DOS machine, translates cleanly to a similar function in Visual Basic. The serial and parallel ports are accessible again!

The trouble is, the brave new world of the PC will not support these legacy devices. If you follow PC-design trends, you know that the ISA expansion slots in the PC have been



Photo 1—The USBSIMM is a USB-enabled controller card that is placed in a 30-pin SIMM socket, or mounted vertically or horizontally on a board with 0.1" pin-center spacing.

designated as obsolete. The parallel port and serial port will soon follow.

The deficiencies of the legacy ports are too expansive to describe here. Suffice it to say, PC performance has evolved, but the serial and parallel ports still have the same limitations imposed on them as in the mid '80s. The next generation of computers will do away with these two ports.

The UART in the serial port was designed to be polled or to interrupt the CPU for each byte processed. Even with hardware FIFOs, the result was the same. The operating system and the programs that ran under it were generally oblivious to what was connected to the serial ports.

The situation was even worse with the parallel port. Not only did a lot of devices use the port lines in unique ways for their data transfers, but they could also be daisy chained. It was often a shooting match to determine if chained devices would cooperate with each other. Again, the operating system could only control the parallel

port as best it knew how. Peripherals remained external to the operating system, rather than an integrated and functioning part of the OS.

The next generation of computers will replace the serial and parallel ports (along with the keyboard, mouse, and joystick ports) with the Universal Serial Bus. The key advantages of the USB are physical port connectors, a data rate that's fast enough to support a wide number of peripheral devices, the ability to chain many (theoretically, 127) devices from the host port, and protocol robustness.

The disadvantage? Hanging a relay off of your new PC's USB port to control your lights is not an afternoon project. Even the simplest PC interface idea will involve a microcontroller and a USB serial interface engine (SIE), as well as OS drivers.

LONG, WINDING ROAD

Developing a project that interfaces to a USB port is definitely not as straightforward as parallel- or serial-

port interfacing. Making interface circuits for the DOS machine in my basement involved two basic steps—making circuits and writing programs.

A parallel port can provide the TTL-levels needed to adequately control a relay driver. There were eight data lines on the port that could be written to with a simple `out` instruction. Programming and testing this hardware and software combination was not a difficult development effort.

Developing for the USB takes a sharp turn toward the complex. On the peripheral side of the development, a microcontroller is needed to support the complex communication required to be a USB-compliant device. The USB UART is the device I mentioned earlier called the serial interface engine (SIE).

The SIE is implemented differently by different manufacturers. The SIE can be a chunk of silicon in a package meant to be interfaced to the microcontroller of your choice, or it can be a integral part of the microcontroller

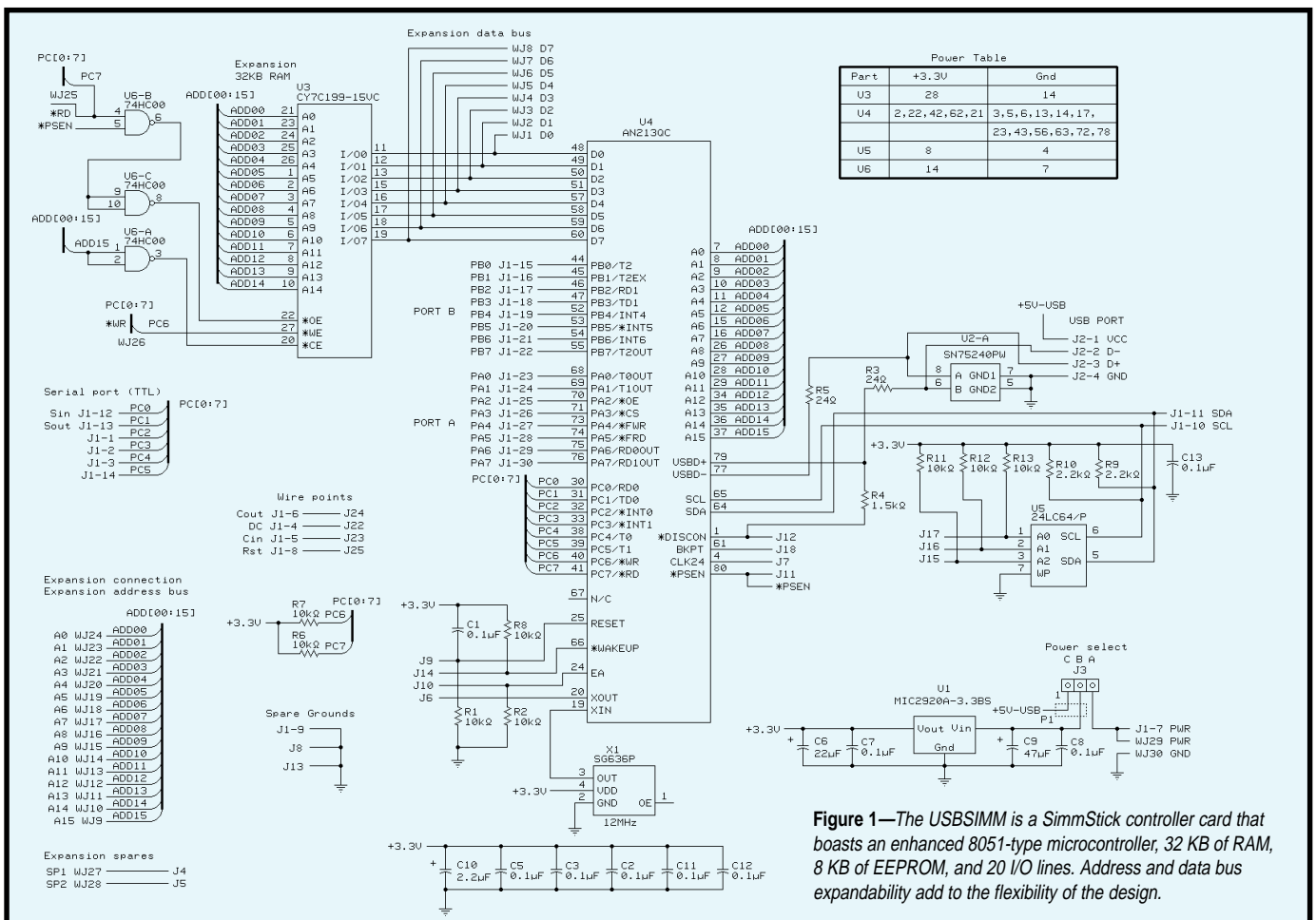


Figure 1—The USBSIMM is a SimmStick controller card that boasts an enhanced 8051-type microcontroller, 32 KB of RAM, 8 KB of EEPROM, and 20 I/O lines. Address and data bus expandability add to the flexibility of the design.

itself. One implementation of the SIE can decode the serial USB datastream and provide the microcontroller with rather raw information, while another more advanced SIE may do more of the communication work. The protocol requires much more than just transferring data bytes.

In any case, there are a lot of steps to USB device development to get the device to function the way a USB peripheral should. The peripheral USB device must follow a predefined protocol to communicate with the low-level OS functions, enabling it to be recognized as a USB device.

The OS no longer blindly deals with a USB device as it may have with a serial or parallel peripheral. With an OS like Windows 98, there is a sublimely orchestrated interplay of USB host hardware and several layers of drivers. To achieve a tight coupling with the OS, direct access to the hardware has been cloaked behind a curtain of OS calls. Access to this functionality is limited to drivers and API calls. This is definitely not a place for a BASIC POKE.

Development has now evolved from the two simple steps that I mentioned earlier to:

Pin	Signal Name	Function
1	A1 (PC2)	Port C bit 2. General purpose I/O or interrupt 0.
2	A2 (PC3)	Port C bit 3. General purpose I/O or interrupt 1.
3	A3 (PC4)	Port C bit 4. General purpose I/O or timer 0.
4	Power	PWR connects to a wire point
5	Clock in	CI connects to a wire point
6	Clock out	CO connects to a wire point
7	VDD	+5V in/out
8	Reset	RES connects to a wire point
9	Ground	Ground
10	SCL (I2C)	I2C clock pulled high
11	SDA (I2C)	I2C data pulled high
12	Serial in	SI is connected to serial Port 0.
13	Serial out	SO is connected to serial Port 0.
14	IO (PC5)	Port C bit 5. General purpose I/O or timer 1.
15	D0 (PB0)	Port B bit 0
16	D1 (PB1)	Port B bit 1
17	D2 (PB2)	Port B bit 2
18	D3 (PB3)	Port B bit 3
19	D4 (PB4)	Port B bit 4
20	D5 (PB5)	Port B bit 5
21	D6 (PB6)	Port B bit 6
22	D7 (PB7)	Port B bit 7
23	D8 (PA0)	Port A bit 0
24	D9 (PA1)	Port A bit 1
25	D10 (PA2)	Port A bit 2
26	D11 (PA3)	Port A bit 3
27	D12 (PA4)	Port A bit 4
28	D13 (PA5)	Port A bit 5
29	D14 (PA6)	Port A bit 6
30	D15 (PA7)	Port A bit 7

Table 1—The pinout of the USBSIMM follows that of the SimmBus standard.

- design hardware—a microcontroller is mandatory with either an external or internal SIE
- create firmware to handle USB communication for device recognition
- supplement the basic firmware with application-specific routines
- create a Windows device driver that will establish a communication pathway to your device

development is simplified. As a matter of fact, if the USB device can be designed to support a handful of general-purpose I/O lines, the same core design could be a building block for a large number of projects.

However, driver development is no walk in the park. Besides the complexity, there is a serious investment in development tools. If your goal is

- develop an application that uses the driver to talk to the USB device

USB FOR DUMMIES

If developing a USB peripheral is such a long and laborious task, what can be done to make it easier? What part of the process can be eliminated or reduced in complexity? If you have a circuit you wish to control from the USB port on a PC, how can the design process be made easier?

Most of the devices I interfaced with the DOS PC are designs that are not excessively complex. In fact, most outputs are rather simple, such as controlling a relay. The inputs I've done with the monitor have often been the states of switches or other two-state devices.

If the USB device only needs to interface with basic circuitry, then the firmware

SimmStick

The SimmStick was a concept originally developed by Antti Lukats from Estonia. The PC industry used 30-pin SIMM printed circuit boards with dynamic RAM as modules for motherboard memory expansion. The modules snapped easily into the SIMM sockets. Lukats surmised that, because 30-pin SIMM sockets were inexpensive and widely available, using SIMM sockets for something other than memory was a resourceful idea.

The idea was to place a small microcontroller on a 3.5" wide SIMM-like board, the SimmStick. Port lines from the microcontroller would extend to the card edge fingers. A SimmBus pinout evolved to include 16 or more port lines, serial lines, and I²C. The intent of the signals present on the SimmBus was to provide control lines to operate peripherals.

A typical SimmStick controller may be a PIC processor. The PIC on the SimmStick can be programmed through the SimmBus or through a dedicated ISP port connection. A crystal and a voltage regulator may also be present on the SimmStick.

SimmStick peripherals, such as LED modules and relay modules, followed this small form factor. To make a system, a motherboard with several SIMM sockets provides a carrier for the SimmSticks. The motherboards are often designed as a source of power for the SimmSticks, as well as providing serial and other off-board signal connections.

Dontronics, an Australian electronics company, has taken the lead in designing and manufacturing SimmSticks. Dontronics offers PIC and Atmel-based controllers, motherboards, and peripheral boards.

Command	Byte	Value	Function
Set Port A direction	1	73	Set port A direction 1 = output, 0 = input
	2	dd	
Set Port B direction	1	74	Set port B direction 1 = output, 0 = input
	2	dd	
Read Port A data	1	69	Read port A. Data returned in byte 2.
Read Port B data	1	70	Read port B. Data returned in byte 2.
Write Port A data	1	65	Read port A command dd = value to write
	2	dd	
Write Port B data	1	66	Write port B command dd = value to write
	2	dd	

Table 2—The firmware of the USBSIMM provides port control functionality that can be accessed from Visual Basic.

to create a quick Visual Basic application to interface to some simple hardware, it's a burden to use Visual C to develop a driver for your "quick" app.

USBSIMM

If USBSIMM had only one "m", then the "sim" might stand for simple (see the SimmStick sidebar). The

controller (EZ-USB AN2131Q) on the USBSIMM board is manufactured by AnchorChips, a division of Cypress Semiconductor. AnchorChips started the design with a souped-up 8051 that offers a 3× performance boost over a standard 8051. They then added two serial ports, three timers, an advanced SIE, and lastly, 8 KB of internal RAM.

USBSIMM module is a straightforward hardware design. Although an effort was made to minimize components, the module was designed to offer a fair amount of flexibility. In addition to the microcontroller, the USBSIMM has an external 32-KB RAM chip and an 8-KB serial EEPROM (see Figure 1).

The internal RAM is a key feature. The AN2131Q can run firmware from this memory. This design is referred to as soft. Because the firmware is not placed into nonvolatile memory, it can easily be upgraded. This way, if a product is made with an AnchorChip microcontroller, the firmware can ship through a CD or floppy to the product driver.

To achieve soft architecture, some form of firmware download must take place. The advanced SIE takes care of all of the housekeeping needed to support the firmware download. As a matter of fact, the SIE holds the 8051 in reset and takes care of all of the USB communication needed to enumerate to the PC as a peripheral USB device. This is no small feat.

Once the PC is capable of recognizing the SIE as a USB device, the communication channel between the PC and the SIE is used as the path to transfer firmware code to the internal RAM. The PC can then command the SIE to take the 8051 core out of reset to start executing the firmware from

RAM. If designed correctly, the firmware-loading device disappears from the bus, and the new device appears. AnchorChips calls this process reenumeration.

The USBSIMM, either used as a daughterboard in custom hardware or with other SimmStick modules, forms the core of a reasonably potent embedded USB device. The USBSIMM routes all of its port pins to the SimmBus edge connection. The pinout is shown in Table 1.

SHORTCUTS

The USBSIMM, shown in Photo 1, simplifies the development process by creating a general-purpose hardware platform for easy development. But, the 8051 doesn't run on good intentions. Despite the advanced SIE's power to conveniently download firmware to the USB device, code must still be developed for the USB peripheral to function correctly after the SIE allows the 8051 to execute.

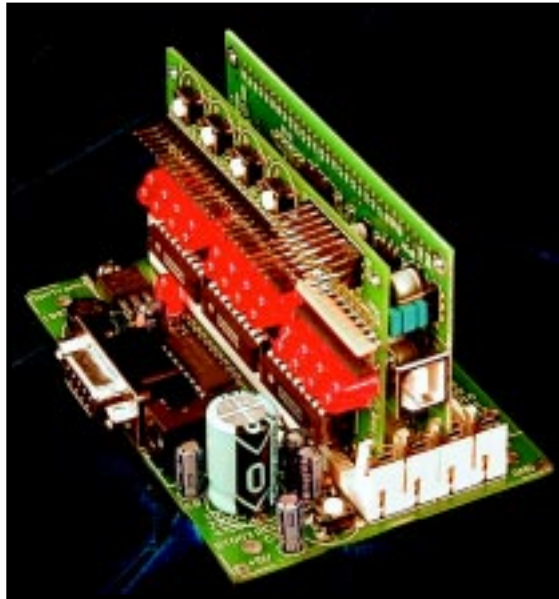


Photo 2—The flexibility of the SimmStick products allows for a quick prototype of a USB design. Here, a USBSIMM board is mounted on a carrier motherboard, along with a support board that has pushbuttons and status LEDs.

Because the intent of this design is to provide a platform for a USB device to control relay circuits and read switch settings, the firmware need only provide two general functions.

The first function is for the firmware to behave as a standard USB device. Rather than creating a USB device that relies on a custom driver for Windows communication, this firmware will appear to Windows as a human interface device (HID). Examples of HIDs are the mouse and keyboard, standard devices for PCs. [1, 2, 3]

If the firmware loaded in the USBSIMM follows the correct configuration as a HID, Windows will load a standard driver to communicate with the device using HID reports, thus eliminating the large step of developing a custom Windows driver.

The section function of the firmware provides routines that will read and write to the port lines on the card. These routines

should allow any of the 16 lines to be set as inputs or outputs, allow the outputs to be set to a logic high or low, and they should also read the state of the input lines.

I developed HID firmware that handles this functionality. If a USBSIMM is loaded with this firmware, Windows sees the USB device as a HID. After Windows detects the USBSIMM, it loads the firmware into the internal RAM.

You may be wondering just how Windows knows to load this firmware into the USBSIMM. Well, it uses a driver. When any new USB device is connected to the PC, the operating system queries the device for vendor and product information. Windows uses this information to determine which driver needs to be associated with the device.

In our case, when the USB-SIMM is first attached, the information Windows receives indicates that the device is an AN2131Q without firmware. Windows searches for the right driver for this hardware. If it doesn't find it already in residence on the hard drive, a request to the user appears in the form of a prompt. For example, the driver may be located on a CD or floppy disk.

The function of this initial driver is strictly to load the firmware to the USBSIMM. The driver files are provided on the *Circuit Cellar* website, or at usbsimm.home.att.net.

After the firmware is loaded, Windows associates the standard HID driver with the new device. Whenever the USBSIMM is attached, Windows uses the first driver to load the firmware to the internal RAM, reenumerates, and connects the HID driver with the firmware. Once Windows sees the USBSIMM as a HID, then an application can communicate with the HID using HID reports [1, 2].

HID reports are the standard format for communication between Windows and a HID. The OS host and the USB device must follow the right format to communicate using these reports. The formats of these reports are too extensive to cover here. They are covered in detail in, *USB Design by Example* and *USB Complete* [1, 2].

The HID reports for the firmware are described in Table 2. These reports allow the host program to set port

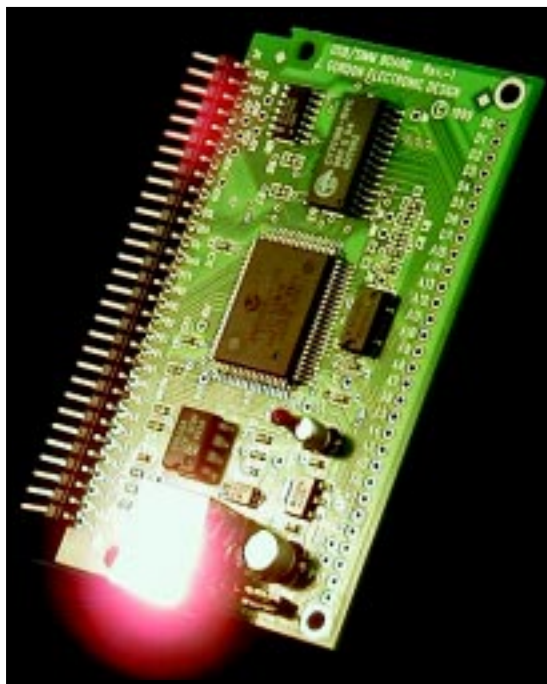


Photo 3—The USBSIMM can be easily used as a prototype system by attaching a row of right-angle pins to the circuit board and inserting it into a solderless proto board.

direction, as well as read from and write to the USBSIMM port lines. Support for accessing the serial EEPROM is described on the web site.

HOST DEVELOPMENT

As described earlier, the host application software interfaces to the device through a standard Windows driver because the USB device has been recognized as a HID. Development of an application in Visual Basic then involves communication with this HID through Windows API calls.

John Hyde, author of *USB Design by Example*, developed the Visual Basic (VB) routines that I use to access Windows API. These routines specifically address communication with a USB HID. Three main functions allow access from VB to the firmware routines in the USBSIMM.

The `OpenUSBdevice` call finds the correct USB device to talk to. The firmware loaded into the USBSIMM identifies itself by firmware revision. In this case, the firmware is called `USBSIMM1v1.0`. Once found and opened, the application can send HID reports to the firmware with the `WriteUSBdevice` function and read HID reports from the firmware using the `ReadUSBdevice` function.

The host must transfer data in the format defined by the HID report. The firmware written for the USBSIMM expects all reports to fit one format. This requirement simplified the development of firmware. The only odd feature about having a single report format is that the report is a fixed size. Six data bytes are always transferred between the host and the USB device. The unused bytes can be set to zero or ignored.

The USBSIMM firmware supports six commands. The sixteen port lines on the USBSIMM originate from port A and B of the AN2131Q. The commands are split into two sets, with each command appropriate for a particular port.

The eight lines of each port can be individually selected to be inputs or outputs. The port

direction command can change the port direction from the default of input to output by setting the appropriate bit of the data byte in the command to a one. See Listing 1 for some simple example VB functions.

A port read will read the port lines regardless of their direction. If only part of the port is set to input, the other lines can be masked to preserve the important information. Port lines that are set as inputs will remain unaffected by a port write.

IS IT COLD OUTSIDE?

I currently use the USBSIMM to interface with home-automation functions. Every door and window in my house is monitored by my security system. Hardwired loops have reed contacts that detect when any door or window is opened. This is great for intrusion detection, but so far it has gone unused for any home-automation purpose.

So, other than the detection of intruders, why would a home-automation system care about open windows, especially when it can't close them? In the spring and fall, the outside air temperature falls into my comfort range for a few short days. During that time, the windows are opened for

Listing 1—An example of Visual Basic access to the USBSIMM firmware routines.

```
' Set all of the lines of port A to output
Private Sub PortAin()
    Dim OutBuffer(10) As Byte
    OutBuffer(0) = 73 ' Port A direction command
    OutBuffer(1) = &H0 ' 0 is input
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
End Sub

' Set all of the lines of port A to output
Private Sub PortAout()
    Dim OutBuffer(10) As Byte
    OutBuffer(0) = 73 ' Port A direction command
    OutBuffer(1) = &HFF ' 1 is output
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
End Sub

' Set all of the lines of port b to input
Private Sub PortBin()
    Dim OutBuffer(10) As Byte
    OutBuffer(0) = 74 ' Port B direction command
    OutBuffer(1) = &H0 ' 0 is input
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
End Sub

' Set all of the lines of port B to output
Private Sub PortBout()
    Dim OutBuffer(10) As Byte
    OutBuffer(0) = 74 ' Port B direction command
    OutBuffer(1) = &HFF ' 1 is output
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
End Sub

' Read port A
Private Sub ReadA()
    Dim OutBuffer(10) As Byte
    Dim InBuffer(10) As Byte
    OutBuffer(0) = 69 ' Read port A command
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
    Call ReadUSBdevice(AddressFor(InBuffer(0)), 6)
    ReadAWindow.Text = TwoHexCharacters(InBuffer(1))
End Sub

' Read port B
Private Sub ReadB()
    Dim OutBuffer(10) As Byte
    Dim InBuffer(10) As Byte
    OutBuffer(0) = 70 ' Read port B command
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
    Call ReadUSBdevice(AddressFor(InBuffer(0)), 6)
    ReadBWindow.Text = TwoHexCharacters(InBuffer(1))
End Sub

' Write port A
Private Sub WriteA()
    Dim OutBuffer(10) As Byte
    OutBuffer(0) = 65 ' Write port A command
    OutBuffer(1) = ReturnHexByte(WriteAWindow.Text)
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
End Sub

' Write port B
Private Sub WriteB()
    Dim OutBuffer(10) As Byte
    OutBuffer(0) = 66 ' Write port B command
    OutBuffer(1) = ReturnHexByte(WriteBWindow.Text)
    Call WriteUSBdevice(AddressFor(OutBuffer(0)), 6)
End Sub
```


fresh air, closed when it gets cool or warm, and opened again.

Unfortunately, the furnace or air conditioner may not be disabled when the windows are open. I'm sure my air conditioner won't cool the great outdoors and I wouldn't put any money on the ability of my furnace to heat the entire city, either.

A simple kill-switch type of relay contact can be inserted in the electric thermostat loop to prevent the furnace or air conditioner from running while any windows or doors are open. My home-wiring configuration provides an active signal if one or more windows are open. Another signal indicates door status.

The USBSIMM is mounted in a carrier SIMM socket on a board that interfaces to these signals, as well as to the kill relay. For the purpose of simplified illustration, and to make bench-top debugging manageable, I used a DT003 SimmStick motherboard to hold the USBSIMM, along with a DT203 SimmStick I/O module for status lights and pushbuttons (see Photo 2). The USBSIMM can also be used in solderless proto boards for rapid prototyping, as seen in Photo 3.

The DT203 SimmStick I/O module has an LED status light for each of the SimmBus's 16 I/O lines. This allows for display of several status signals, one of which is the kill-relay state. Four buttons on the top of the DT203 acted as door and window inputs.

The USBSIMM communicates the status of the inputs to a VB application that decides whether to open the kill relay based on the status of the doors and windows. A USB development that interfaces with a Pentium III to make a decision a couple of resistors and a broken transistor could handle is definitely overkill, but this is just step one. The exciting part comes with more development.

FUTURE DIRECTION

At the time this article was penned, the USBSIMM firmware supported basic port I/O. USBSIMM firmware development is continuing and will include several practical functions such as serial port communication, A/D converter interfacing, and

X-10 control. Once developed, these features will be made freely available.

By eliminating difficult or expensive development steps, a USB device can interface to the same type of custom hardware that is often hung off of a parallel port. By using the USB-SIMM and the firmware developed for it, developing custom circuits and the programs that control them is simple.

Using USB today as the PC interface method for your next project assures that you won't be fighting a battle to find an appropriate port on the next generation's old PCs. ☑

Rob Severson is a senior design engineer and creator of all things USB at J. Gordon Electronic Design, Inc. Although Rob is not a deity (by any stretch of the imagination), his coworkers would be amused if you reached him at usbgod@jged.com.

REFERENCES

- [1] J. Hyde, *USB Design by Example: A Practical Guide to Building I/O Devices*, Wiley, NY, www.usb-by-example.com, 1999.
- [2] J. Axelson, *USB Complete: Everything You Need to Develop Custom USB Peripherals*, Lakeview Research, Madison, WI, 2000.
- [3] J. Lyle, "USB Primer: Classes and Drivers", *Circuit Cellar* 107, June 1999.

SOURCES

AN2131Q
AnchorChips
(858) 613-7900
Fax: (619) 676-6896
www.anchorchips.com

USBSIMM
Control Solutions, Inc.
(612) 784-3309
Fax: (612) 786-5778
usbsimm.home.att.net

J. Gordon Electronic Design
(612) 786-2405
www.jged.com

DT003 SimmStick
Dontronics
+613 9338-6286
Fax: +613 9338-2935
www.dontronics.com

Flash Forward

FEATURE ARTICLE

Bob Brown & Michal Tamborski

Implementing Downloadable Firmware via Flash Memory

If you're going to use flash memory to simplify firmware upgrades, you'll need to design your system from the beginning with this in mind. Listen in as Bob and Michal explain some of the problems and solutions that come with flash memory.



Just a few years ago, the standard method of doing a firmware upgrade was to send an EPROM to each customer.

The customer then had to open up the system, pull the old EPROM out of its socket, and plug the EPROM with the new firmware into the socket. If no leads were bent, the customer could run the new version of the firmware.

With the success of flash memory, you can eliminate the need to perform EPROM swaps for firmware upgrades. However, using flash memory is a little more involved than simply substituting a flash-memory device for an EPROM. You must design your system from the start to make firmware upgrades as easy as possible. In this article we present some of the problems we encountered and methods we used to perform firmware upgrades.

WHAT IS FLASH MEMORY?

Flash memory can be considered part of the line of nonvolatile electrically programmable memory devices. The EPROM is the oldest relative of flash memory. EPROMs are typically programmed on a device programmer external to the target system.

During programming, EPROM require a super voltage (a voltage that is above the normal operating voltage

range). A typical EPROM that operates on a +5-V power supply may require a super voltage of from +10 to +25 V during programming. EPROMs are erased by exposure to UV light. This erasure method requires that EPROMs be placed in ceramic packages with clear windows to allow the UV light rays to reach the die.

Because the windowed ceramic package is more expensive to produce than standard plastic packages, one-time-programmable (OTP) EPROMs are produced in plastic packages. The OTP EPROMs are less expensive, but they cannot be erased.

The EEPROM is another device in the line of nonvolatile electrically programmable memory devices. EEPROMs do not require a device programmer and are in-system programmable. Typically, they are programmed on a byte-by-byte basis with a write to a memory address initiating an erasure, and then programming of that byte. This operation typically takes a few milliseconds per byte.

During the '80s, it was thought that because the EEPROM didn't require a windowed ceramic package, it would become less expensive than EPROMs and might make the EPROM obsolete. Of course, this didn't happen.

The memory cell of the EEPROM is more complex than that of an EPROM. As memory densities increased, the EEPROM, with its more complex memory cell, couldn't keep pace with the increasing density of EPROMs. The EEPROM found its niche as a part used when small amounts of nonvolatile rewritable memory are needed. EEPROMs with serial (rather than parallel) interfaces are quite commonly used when a few bytes of permanent storage are needed and access time is not critical.

Flash memory combines many of the best attributes of EPROMs and EEPROMs. Like EEPROMs, flash memory can be erased electrically, eliminating the need for windowed ceramic packages and allowing flash-memory devices to use less expensive plastic packages. Like EPROM, flash memory has a memory cell that's less complex than that of the EEPROM, which allows flash memories to have

densities that are competitive with that of EPROMs. Given these factors, flash memory has the potential to do what the EEPROM has failed to do—make the EPROM obsolete.

Early flash memories required an external super voltage for programming. This meant that to program a flash memory in-system, you had to provide and control the super voltage.

Many of the analog IC companies produced devices to create the super voltage for flash memories from +5 V. Of course, the need for the super voltage increased system cost and complexity. The newer flash memories can be erased and programmed using only the system logic voltage, typically +5 or +3.3 V, which makes designing in flash memory as simple as designing in an EPROM.

Flash memory differs from EPROM and EEPROM in the style of erasure and programming. Flash memories are block devices so they are erased and programmed at the block level.

The block (or sector) size can be uniform within a given flash-memory device. For example, we've used a 512-KB device that has eight 64-KB blocks. However, the block size does not have to be uniform. Another device we've used has 256 KB with sectors that consist of a 16-KB, two 8-KB, one 32-KB, and three 64-KB blocks.

Block sizes vary from fine-grain devices, with sector sizes in the 128-byte range to course-grained devices with sectors of 64 KB or larger. Regardless of the block size or arrangement, in order to write a byte to a flash memory, you must erase and then reprogram an entire block. This is a more complex operation than simply writing to the device, as is required by an EEPROM.

Some flash memories, often called boot block, have special sectors that reside at the top or bottom of the device's address space. These special boot block sectors can be locked to prevent reprogramming in system. The boot block sectors are designed to prevent the system's boot-up code from being accidentally erased.

You choose a boot block flash memory with the boot block at the bottom of the address space if your

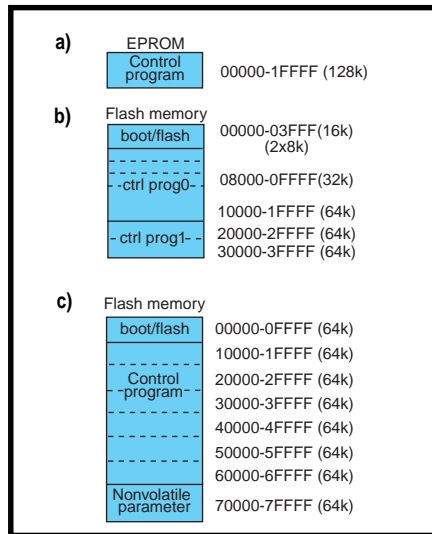


Figure 1—The first generation (a) system contained EPROM and EEPROM, whereas our second-generation system (b) combined flash memory and EEPROM. The third-generation (c) used just flash memory.

microprocessor runs from address 0 at reset (e.g., 68k family) or a boot block at the top of the address space for a micro that runs from the top of memory at reset (e.g., 80X86 family).

FLASH PROGRAMMING

Programming typical flash-memory devices is usually accomplished by the specific command sequence, which initiates the Embedded Erasure or Embedded Program algorithms. Let's look at the three procedures. The details are for one of the AMD flash-memory devices we used. Most flash-memory devices are similar.

Device identification is the first step before any erasure or programming takes place. The host system can run an autoselect sequence, allowing you to get manufacturer and device codes. All command sequences start with a pair of unlock writes to the flash-memory device (a write of AAH to flash address 555H followed by a write of 55H to flash address 2AAH). Then the autoselect command 90H is written to address 555H.

At this point, the device is in a mode that allows the host system to execute two consecutive reads at address XX00h and XX01h (manufacturer and device IDs, respectively). The system has to exit this mode by executing RESET command (writing F0H to flash address 2AAH) to return to normal reading array data mode.

Erasure is a six-bus-cycle operation and can be applied to the entire chip or to just the selected sector. We used the sector-erase algorithm only.

Two unlock write cycles start the sequence, followed by a setup write cycle command (80H to 555H). Then, an additional two unlock write cycles are followed by writing the erase command (30H) to the address of the sector to be erased. Those six write cycles initiate Embedded Erase algorithm.

The host system should monitor the status of the erase operation by checking the Toggle and Timeout bits. The Toggle bit (bit 7 for our device) changes state on every read from the device until the erase or program operation is complete.

Before running the programming sequence, a copy of the data block to be written should be stored safely in the RAM. Byte programming is implemented in a four-bus-cycles sequence. The first two cycles are the unlock write cycles, followed by the program setup command (A0H to 555H). Then the actual data is written to the desired address. The host system can determine the status of the programming operation by checking the Toggle and Timeout bits. The typical byte programming time is 2–20 μs.

SYSTEM ARCHITECTURE

As you can see in Figure 1a, our first-generation system did not use flash memory, it used a 68EC000

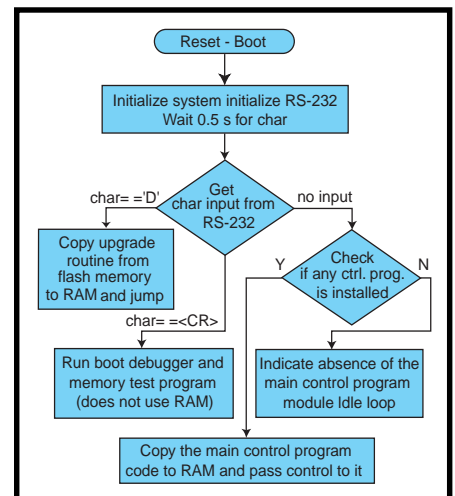


Figure 2—The boot program algorithm executes out of the boot sector of the flash memory on a system reset. It monitors the serial port and then executes the proper procedure, based on the serial port activity.

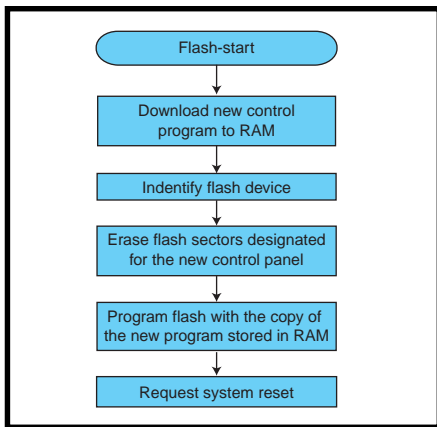


Figure 3—This program algorithm is loaded into RAM from flash memory to upgrade a control program.

processor that was configured for an eight-bit wide data bus. The program code was stored in a 128K × 8 EPROM. Data storage consisted of some SRAM and a small EEPROM for nonvolatile parameter storage.

The system had a serial port for board testing and system debugging. Field program upgrades were done by shipping EPROMs to customers, who then opened the system, removed the old EPROM and replaced it with the new EPROM. This is an expensive and error-prone procedure. It was decided that the next generation system must be upgradable without chip swapping. The obvious choice was to use flash memory and not the EPROM.

However, creating a field-upgradable system that replaces EPROM with flash memory requires the system to be designed with field upgradability in mind. You can't execute code that resides in the flash memory while the flash device is being erased or programmed. That meant that our flash-memory programming code must execute out of RAM during actual flash programming.

We also wanted to be protected in case of a power outage during a program upgrade. It would be OK for the user to have to restart the upgrade, but we had to be sure that the code to perform the upgrade was always protected from the loss of power.

Given those major considerations and other system factors, the second-generation system in Figure 1b used a 68EC020 processor. Taking advantage of the dynamic bus sizing capability of the 68EC020, we used a 256K × 8

flash memory, a 128K × 32 SRAM bank, and a 2K × 8 EEPROM.

Like the older system, there was also a serial port that would be used for program upgrades, in addition to its testing and debugging use. To allow our code to run at maximum speed, it would execute in the 32-bit wide SRAM, which meant that there would be no performance penalty for using only an 8-bit wide flash memory (with a few wait states).

Following a reset, our code is moved from flash memory to SRAM and executed in SRAM. Because executing code out of SRAM would be a requirement for programming the flash memory during a field upgrade, we needed this capability anyway. After weighing all the issues, we came up with the organization for the firmware.

Instead of the single program that was used in the EPROM-based system, the firmware would consist of three programs. We came to refer to these three programs as the boot, flash, and control programs. All of these programs would be stored in the flash memory, but the flash and control programs execute out of SRAM.

The control program is the main operating program for the system. This, of course, is the program we want to be able to upgrade in the field. Its actual operation isn't relevant to this article, however the operation of the boot and flash programs are, so let's look at them further.

BOOT PROGRAM

The boot program is the code that executes on a system reset (see Figure 2). It performs basic system initialization and then waits in a loop for 0.5 s, checking for a character from the serial port. The boot program resides and executes out of the boot block section in the flash memory.

If no character was received, the boot program would copy the control program from flash memory into SRAM. Then it would execute a jump to the start of the control program and the system would run normally.

If an ASCII "D" character is retrieved from the serial port during the loop, it indicates that a field upgrade is to be started. The flash program is

copied from flash memory to SRAM, then it jumps to the start of the flash program. If a <CR> is retrieved from the serial port, the boot program brings up a low-level debugger that could be used for board troubleshooting.

FLASH PROGRAM

Like the boot program, the flash program (see Figure 3) is stored in the boot block sector of the flash-memory device. As mentioned, the flash program is copied from flash memory into SRAM and executed from SRAM when a control program upgrade is requested. Once started, the flash program downloads the new control program over the serial port to where it is temporarily stored in SRAM.

After the download is complete, the appropriate flash blocks are erased and the new control program is written into those blocks. When that's done, the upgrade is complete and the user is prompted to reset the system.

On this system, the flash memory has a 16-KB boot block. By placing the boot and flash programs in the boot block sector, the flash program was protected. Having the flash program in the boot block means the flash program code could never be erased.

If the flash update code was part of the control program, it would be erased during a field upgrade. If a power outage occurred before the new program was written into flash memory, the system would be left without a control program and without the ability to download a new program. By having the flash program in the boot block, the system always has the ability to download a new control program, even if the last update attempt is interrupted by a power outage.

Of course, storing both the boot and flash programs in the boot block has its downsides. It means the code for both programs must be small enough to fit in the boot block and the programs can never be updated. Due to these constraints, we kept the boot and flash programs quite simple.

Along with the system code, we needed to create a few utility programs to convert the output of our development system's linker into the formats needed for our system and a

program to perform the field download of new control programs.

The flash and control programs had to be linked to run at their final address in RAM. Because the programs would be stored in a different address in the flash memory, we created a program to create an S-record file whose address fields were for the appropriate address in the flash memory. Another utility merged the S-record files for the boot, flash, and control programs into one S-record file for initial flash-memory programming.

Another utility converted the linker output for the control program into our download file format. Because this format was the exact image of the control program as stored in the flash memory, it is called the image file.

The image file has a 512-byte header. The first 64 bytes are ASCII text that includes the program revision, creation date, and a comment field. The comments are mainly used for in-house identification of special program versions. The remainder of the header is binary and contains the length and start address of the text, data, and bss program segments. Following the header is the exact binary image of the program, as it will reside in RAM during program execution.

These utilities were used in-house and we created quick no-frills command-line programs in a couple of days. We also created a quickie version of the download program, however we had to create more polished versions of the download program for both Windows and Apple systems.

The download program accepts a file name to be downloaded, then displays the first 64 bytes of the file header, which shows the user the revision and creation date of the file. The program then prompts the user to reset our system. Our system responds to the download request and sends the first 64 bytes from the program header(s) (the second-generation system allowed two control programs to be stored in flash memory, the third generation, only one) that it has stored in flash memory. The download program displays these headers so the user can see the currently stored program versions.

The user can then abort the download process or (in the second-generation system) select the program to overwrite. The file is downloaded in 512-byte blocks, each block is followed by a checksum byte and each block is acknowledged by the system. When the download is complete, the system writes the program into flash memory and sends an acknowledge to complete the transaction. The upgrade is complete in about a minute.

Another option for programming flash memory is the micro's JTAG or BDM ports. Our third-generation system (see Figure 2c) uses a ColdFire processor that has both ports available. We used the JTAG port to initially program the flash-memory device, so the flash memory could be soldered to the board without having to be programmed. In our case, JTAG wasn't an option for end-user upgrades.

IT'S ALL YOURS

The method we've discussed was developed for downloadable program upgrades using flash memory and has

made field program upgrades almost painless. With a tweak here and there, this method could be modified for use on many different systems. 📄

Bob Brown is a consultant specializing in embedded systems. His company, Alta Engineering, also sells electronic kits that he has designed. Bob has published more than a dozen articles on computers and electronics. You may reach him at alta@ieee.org.

Michal Tamborski is a software engineer working for Ulte division of Heidelberg Publishing Services. He is involved in embedded programming for systems controlling high-resolution laser image setters. You may reach him at mtamborski@ultra.com.

SOURCE

Flash devices

Advanced Micro Devices, Inc.
(408) 732-2400
Fax: (408) 732-7216
www.amd.com

Building a RISC System in an FPGA

FEATURE ARTICLE

Jan Gray

Part 1: Tools, Instruction Set, and Datapath

To kick off this three-part article, Jan's going to port a C compiler, design an instruction set, write an assembler and simulator, and design the CPU datapath. Get reading, you've only got a month before your connecting article arrives!



used to envy CPU designers—the lucky engineers with access to expensive tools and fabs. But, field-programmable gate arrays (FPGAs) have made custom-processor and integrated-system design much more accessible.

20–50-MHz FPGA CPUs are perfect for many embedded applications. They can support custom instructions and function units, and can be reconfigured to enhance system-on-chip (SoC) development, testing, debugging, and tuning. Of course, FPGA systems offer high integration, short time-to-market, low NRE costs, and easy field updates of entire systems.

FPGA CPUs may also provide new answers to old problems. Consider one system designed by Philip Freidin. During self-test, its FPGA is configured as a CPU and it runs the tests. Later the FPGA is reconfigured for normal operation as a hardwired signal processing datapath. The ephemeral CPU is free and saves money by eliminating test interfaces.

THE PROJECT

Several companies sell FPGA CPU cores, but most are synthesized implementations of existing instruction sets, filling huge, expensive FPGAs, and are too slow and too costly for

production use. These cores are marketed as ASIC prototyping platforms.

In contrast, this article shows how a streamlined and thrifty CPU design, optimized for FPGAs, can achieve a cost-effective integrated computer system, even for low-volume products that can't justify an ASIC run.

I'll build an SoC, including a 16-bit RISC CPU, memory controller, video display controller, and peripherals, in a small Xilinx 4005XL. I'll apply free software tools including a C compiler and assembler, and design the chip using Xilinx Student Edition.

If you're new to Xilinx FPGAs, you can get started with the Student Edition 1.5. This package includes the development tools and a textbook with many lab exercises.[3]

The Xilinx university-program folks confirm that Student Edition is not just for students, but also for professionals continuing their education. Because it is discounted with respect to their commercial products, you do not receive telephone support, although there is web and fax-back support. You also do not receive maintenance updates—if you need the next version of the software, you have to buy it all over again. Nevertheless, Student Edition is a good deal and a great way to learn about FPGA design.

My goal is to put together a simple, fast 16-bit processor that runs C code. Rather than implement a complex legacy instruction set, I'll design a new one streamlined for FPGA implementation: a classic pipelined RISC with 16-bit instructions and sixteen 16-bit registers. To get things started, let's get a C compiler.

Register	Use
r0	always zero
r1	reserved for assembler
r2	function return value
r3–r5	function arguments
r6–r9	temporaries
r10–r12	register variables
r13	stack pointer (sp)
r14	interrupt return address
r15	return address

Table 1—The *xr16* C language calling conventions assign a fixed role to each register. To minimize the cost of function calls, up to three arguments, the return address, and the return value are passed in registers.

Listing 1—This sample C code declares a binary search tree data structure and defines a binary search function. Search returns a pointer to the tree node whose key compares equal to the argument key, or NULL if not found.

```
typedef struct TreeNode {
    int key;
    struct TreeNode *left, *right;
} *Tree;

Tree search(int key, Tree p) {
    while (p && p->key != key)
        if (p->key < key)
            p = p->right;
        else
            p = p->left;
    return p;
}
```

C COMPILER

Fraser and Hanson's book is the literate source code of their lcc retargetable C compiler.[1] I downloaded the V.4.1 distribution and modified it to target the nascent RISC, xr16.

Most of lcc is machine independent; targets are defined using machine description (md) files. Lcc ships with 'x86, MIPS, and SPARC md files, and my job was to write xr16.md.

I copied xr16.md from mips.md, added it to the makefile, and added an xr16 target option. I designed xr16 register conventions (see Table 1) and changed my md to target them.

At this point, I had a C compiler for a 32-bit 16-register RISC, but needed to target a 16-bit machine with `sizeof(int)=sizeof(void*)=2`. lcc obtains target operand sizes from md tables, so I just changed some entries from 4 to 2:

```
Interface xr16IR = {
    1, 1, 0, /* char */
    2, 2, 0, /* short */
    2, 2, 0, /* int */
    2, 2, 0, /* T* */
}
```

Next, lcc needs operators that load a 2-byte int into a register, add 2-byte int registers, dereference a 2-byte pointer, and so on. The lcc ops utility prints the required operator set. I modified my tables and instruction templates accordingly. For example:

```
reg: CVUI2(INDIRU1(addr)) \
    "lb r%c,%0\n" 1
```

uses `lb rd, addr` to load an unsigned char at `addr` and zero-extend it into a 16-bit int register.

```
stmt: EQI2(reg,con) \
    "cmpi r%0,%1\nbeq %a\n" 2
```

uses a `cmpi, beq` sequence to compare a register to a constant and branch to this label if equal.

I removed any remaining 32-bit assumptions inherited from `mips.md`, and arranged to store long ints in register pairs, and call helper routines for `mul, div, rem`, and some shifts.

My port was up and running in just one day, but I had already read the lcc book. Let's see what she can do. Listing 1 is the source for a binary tree search routine, and Listing 2 is the assembly code lcc-xr16 emits.

INSTRUCTION SET

Now, let's refine the instruction set and choose an instruction encoding. My goals and constraints include: cover C (integer) operator set, fixed-size 16-bit instructions, easily decoded, easily pipelined, with three-operand instructions (`dest = src1 op src2/imm`), as encoding space allows. I also want it to be byte addressable (load and store bytes and words), and provide one addressing mode—`disp(reg)`. To support long ints we need add/subtract carry and shift left/right extended.

Which instructions merit the most bits? Reviewing early compiler output from test applications shows that the most common instructions (static frequency) are `lw` (load word), 24%; `sw` (store word), 13%; `mov` (reg-reg move), 12%; `lea` (load effective address), 8%; `call`, 8%; `br`, 6%; and `cmp`, 6%. `Mov, lea, and cmp` can be synthesized from `add` or `sub` with `r0`.

69% of loads/stores use `disp(reg)` addressing, 21% are absolute, and 10% are register indirect.

Therefore we make these choices:

- `add, sub, addi` are 3-operand
- less common operations (logical ops, `add/sub` with carry, and shifts) are 2-operand to conserve opcode space
- `r0` always reads as 0
- 4-bit immediate fields
- for 16-bit constants, an optional immediate prefix `imm` establishes the most significant 12-bits of the instruction that immediately follows
- no condition codes, rather use an interlocked compare and conditional branch sequence
- `jal` (jump-and-link) jumps to an effective address, saving the return address in a register
- `call func` encodes `jal r15, func` in one 16-bit instruction (provided the function is 16-byte aligned)
- perform `mul, div, rem`, and variable and multiple bit shifts in software

The six instruction formats are shown in Table 2 and the 43 distinct instructions are shown in Table 3. `adds, subs, shifts, and imm` are uninterruptible prefixes. Loads/stores take two cycles, jump and branch-taken take three cycles (no branch delay slots). The four-bit `imm` field encodes either an `int` (-8-7): `add/sub`, logic, shifts; unsigned (0-15): `lb, sb`; or unsigned word displacement (0, 2-30): `lw, sw, jal, call`.

Some assembly instructions are formed from other machine instructions, as you can see in Table 4. Note that only signed char data use `lbs`.

ASSEMBLER

I wrote a little multipass assembler to translate the lcc assembly output into an executable image.

Format	15-12	11-8	7-4	3-0
rrr	op	rd	ra	rb
rri	op	rd	ra	imm
rr	op	rd	fn	rb
ri	op	rd	fn	imm
i12	op	imm12
br	op	cond	disp8	...

Table 2—The xr16 has six instruction formats, each with 4-bit opcode and register fields.

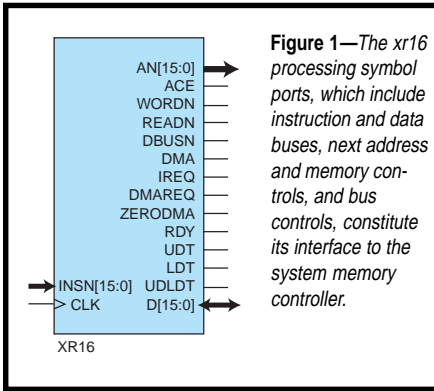


Figure 1—The *xr16* processing symbol ports, which include instruction and data buses, next address and memory controls, and bus controls, constitute its interface to the system memory controller.

The *xr16* assembler reads one or more assembly files and emits both image and listing files. The lexical analyzer reads the source characters and recognizes tokens like the identifier `_main`. The parser scans tokens on each line and recognizes instructions and operands, such as register names and effective address expressions. The symbol table remembers labels and their addresses, and a fixup table remembers symbolic references.

In pass one, the assembler parses each line. Labels are added to the symbol table. Each instruction ex-

pands into one or more machine instructions. If an operand refers to a label, we record a fixup to it.

In pass two, we check all branch fixups. If a branch displacement exceeds 128 words, we rewrite it using a jump. Because inserting a jump may make other branches far, we repeat until no far branches remain.

Next, we evaluate fixups. For each one, we look up the target address and apply that to the fixup subject word. Lastly, we emit the output files.

I also wrote a simple instruction set simulator. It is useful for exercising both the compiler and the embedded application in a friendly environment.

Well, by now you are probably wondering if there is any hardware to this project. Indeed there is! First, let's consider our target FPGA device.

THE FPGA

The Xilinx XC4005XL-PC84C-3 is a 3.3-V FPGA in an 84-pin J-lead PLCC package. This SRAM-based device must be configured by external ROM or host at power-up. It has a

14 × 14 array of configurable logic blocks (CLBs) and 61 bonded-out I/O blocks (IOBs) in a sea of programmable interconnect.

Every CLB has two 4-input look-up tables (4-LUTs) and two flip-flops. Each 4-LUT can implement any logic function of 4 inputs, or a 16 × 1-bit synchronous static RAM, or ROM. Each CLB also has "carry logic" to build fast, compact ripple-carry adders.

Each IOB offers input and output buffers and flip-flops. The output buffer can be 3-stated for bidirectional I/O. The programmable interconnect routes CLB/IOB output signals to other CLB/IOB inputs. It also provides wide-fanout low-skew clock lines, and horizontal long lines, which can be driven by 3-state buffers at each CLB.[2]

The XC4000XL architecture would appear to have been designed with CPUs in mind. Just eight CLBs can build a single-port 16 × 16-bit register file (using LUTs as SRAM), a 16-bit adder/subtractor (using carry logic), or a four-function 16-bit logic unit. Because each LUT has a flip-flop, the

device is register rich, enabling a pipelined implementation style; and as each flip-flop has a dedicated clock enable input, it's easy to stall the pipeline when necessary. Long line buses and 3-state drivers form an efficient word-wide multiplexer of the many function unit results, and even an on-chip 3-state peripheral bus.

THE PROCESSOR INTERFACE

Figure 1 gives you a good look at the xr16 processor macro symbol. The interface was designed to be easy to use with an on-chip bus. The key signals are the system clock (CLK), next memory address (AN_{15:0}), next access is a read (READN), next access is 16-bit data (WORDN), address clock enable: above

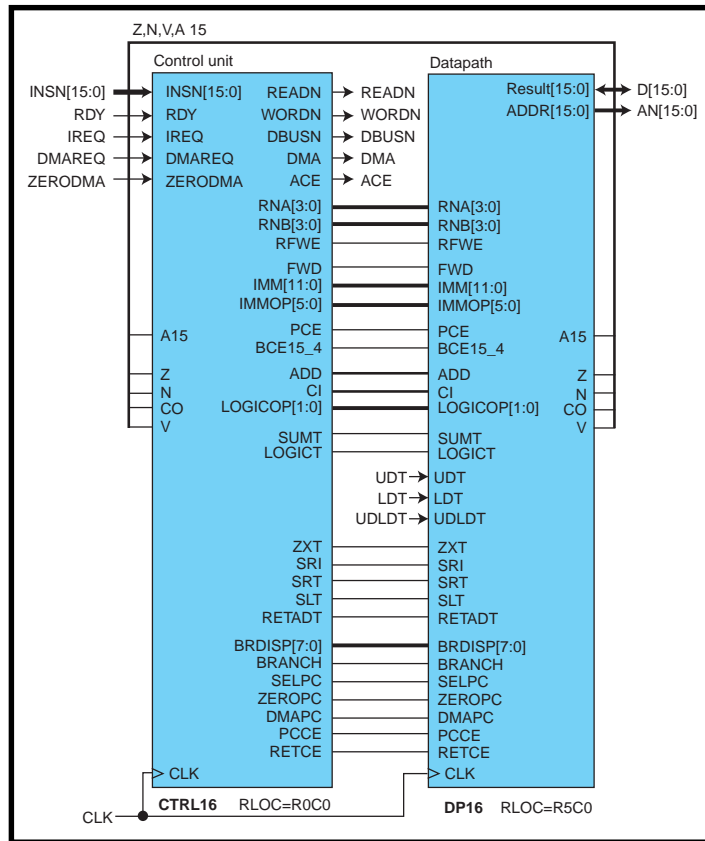


Figure 2—The control unit receives instructions, decodes them, and drives both the memory control outputs and the datapath control signals.

signals are valid, start next access (ACE), memory ready input: the current access completes this cycle (RDY), instruction word input (INSN_{15:0}), on-chip bidirectional data bus to load/store data (D_{15:0}).

The memory/bus controller (which I'll explain further in Part 3) decodes the address and activates the selected memory or peripheral. Later it asserts RDY to signal that the memory access is done.

As Figure 2 shows, the CPU is simply a datapath that is steered by a control unit. Next month, I'll examine the control unit in greater detail. The rest of this article explores the design and implementation of the datapath.

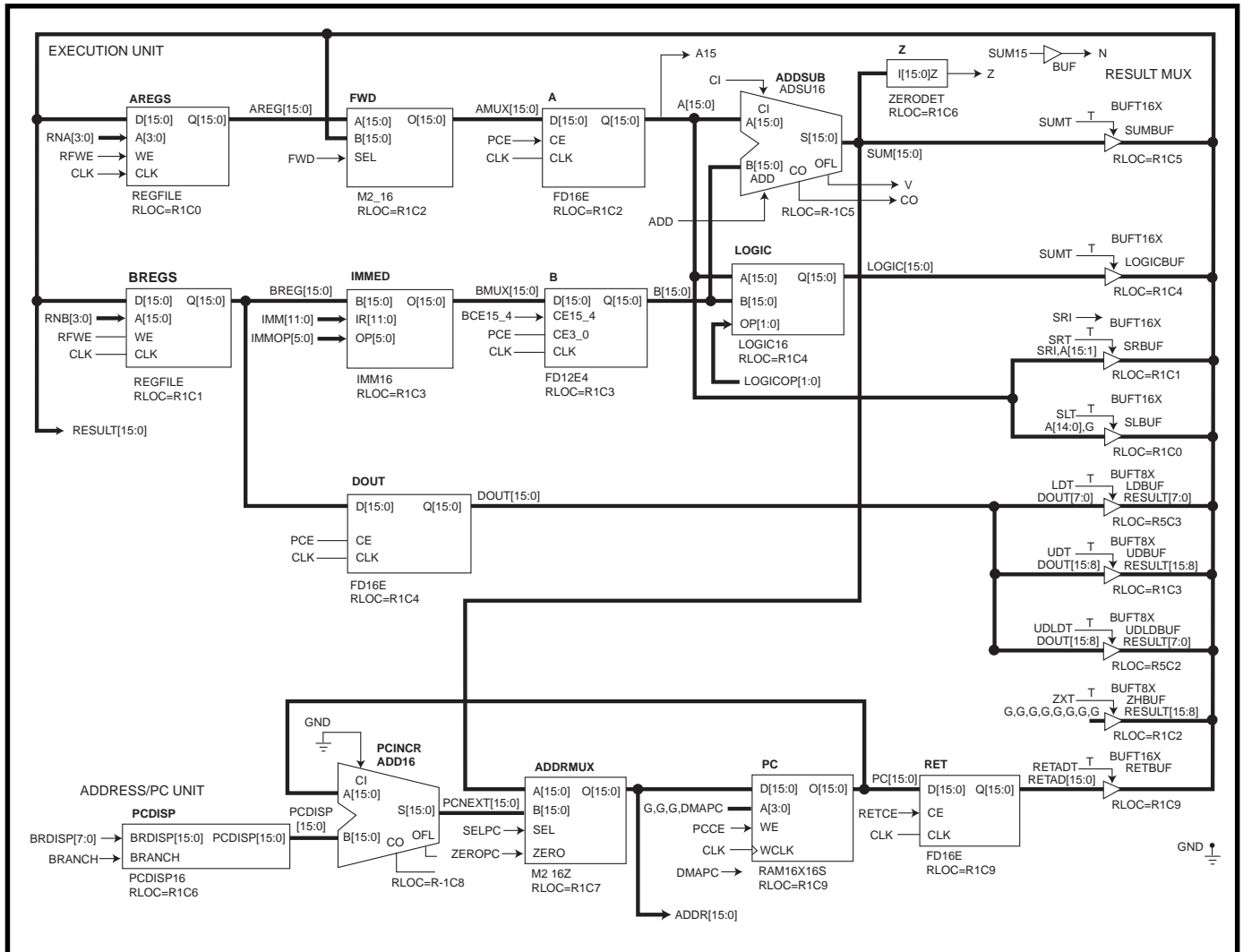


Figure 3—The pipelined datapath has an execution unit, a result multiplexer, and an address/PC unit. Operands from the register file or immediate field are selected and latched into the A and B operand registers. Then the function units, including ADDSUB, operate upon A and B, and one of the results is driven onto RESULT_{15:0} and written back into the register file. Meanwhile, the address/PC unit increments the PC to help fetch the next instruction.

DATAPATH RESOURCES

The instruction set evolved with the datapath implementation. Each new idea was first evaluated in terms of the additional logic required and its impact on the processor cycle time.

To execute one instruction per cycle you need a 16-entry 16-bit register file with two read ports (add r3, r1, r2) and one write port (add r3, r1, r2); an immediate operand multiplexer (mux) to select the immediate field as an operand (addi r3, r1, 2); an arithmetic/logic unit (ALU) (sub r3, r1, r2; xor r3, r1); a shifter (srai r3, 1), and an effective address adder to compute reg+offset (lw r3, 2(r1)).

You'll also need a mux to select a result from the adder, logic unit, left or right shifter, return address, or load data; logic to check a result for,

negative, carry-out, or overflow; a program counter (PC), PC incrementer, branch displacement adder (br L), and a mux to load the PC with a jump target address (call _foo); and a mux to share the memory port for instruction fetch (addr ← PC) and load/store (addr ← effective address).

Careful design and reuse will let you minimize the datapath area because the adder, with the immediate mux, can do the effective address add, and the PC incrementer can also add branch displacements. The memory address mux can help load the PC with the jump target.

DATAPATH SCHEMATIC

Figure 3 is the culmination of these ideas. There are three groups of resources. The execution unit is the

heart of the processor. It fetches operands from the register file and the immediate fields of the instruction register, presents them to the add/sub, logic, and (trivial) shift units, and writes back the result to the register file. The result multiplexer selects one result from the various function units. The address/PC unit drives the next memory address, and includes the PC, PC adder, and address mux. Now, let's see how each resource is implemented in our FPGA.

REGISTER FILE

During each cycle, we must read two register operands and write back one result. You get two read ports (AREG and BREG) by keeping two copies of the 16 × 16-bit register file REGFILE, and reading one operand

from each. On each cycle you must write the same result value into both copies.

So, for each REGFILE and each clock cycle you must do one read access and one write access. Each REGFILE is a 16 × 16 RAM. Recall that each CLB has two 4-LUTs, each of which can be a 16 × 1-bit RAM. Thus, a REGFILE is a column of eight CLBs. Each REGFILE also has an internal 16-bit output register that captures the RAM output on the CLK falling edge.

To read and write the REGFILE each clock, you double-cycle it. In the first half of each clock cycle, the control unit presents a read-port source operand register number to the RAM address inputs. The selected register is read out and captured in the REGFILE output register as CLK falls.

In the second half cycle, the control unit drives the write-port register number. As CLK rises, the RESULT_{15:0} is written to the destination register.

OPERAND SELECTION

With the two source registers AREG and BREG in hand, you now select the A and B operands, and latch them in the A and B registers. Some examples are shown in Table 5.

The A operand is AREG unless (as with add₂) the instruction depends on the result of the previous instruction. Next month, you'll see why this pipeline data hazard is avoided by forwarding the add₁ result directly into the A register, just in time for add₂.

FWD, a 16-bit mux of AREG or RESULT, does this result forwarding. It consists of 16 1-bit muxes, each a 3-input function implemented in a single 4-LUT, and arranged in a column of eight CLBs. The FWD output is captured in the A operand register, made from the 16 flip-flops in the same CLBs. As for the B operand, select either the BREG register file output port or an immediate constant.

For rri and ri format instructions, B is the zero- or sign-extended 4-bit imm field of the instruction reg-

Hex	Fmt	Assembler	Semantics
0dab	rrr	add rd,ra,rb	rd = ra + rb;
1dab	rrr	sub rd,ra,rb	rd = ra - rb;
2dai	rri	addi rd,ra,imm	rd = ra + imm;
3d*b	rr	{and or xor andn adc sbc} rd,rb	rd = rd op rb;
4d*i	ri	{andi ori xori andni adci sbci slli slxi srai srli srxi} rd,imm	rd = rd op imm;
5dai	rri	lw rd,imm(ra)	rd = *(int*)(ra+imm);
6dai	rri	lb rd,imm(ra)	rd = *(byte*)(ra+imm);
8dai	rri	sw rd,imm(ra)	*(int*)(ra+imm) = rd;
9dai	rri	sb rd,imm(ra)	*(byte*)(ra+imm) = rd;
Adai	rri	jal rd,imm(ra)	rd = pc, pc = ra + imm;
B*dd	br	{br brn beq bne bc bnc bv bnv blt bge ble bgt bltu bgeu bleu bgtu} label	if (cond) pc += 2*disp8; r15 = pc, pc = imm12<<4; imm'next _{15:4} = imm12;
Ciii	i12	call func	
Diii	i12	imm imm12	
7xxx	-	reserved	
Exxx	-	reserved	
Fxxx	-	reserved	

Table 3—The xr16 needs only 43 different instructions to efficiently implement an integer-only subset of the C programming language.

ister. But, if there's an imm prefix, load B_{15:4} with its 12-bit imm12 field, then load B_{3:0} while decoding the rri or ri format instruction which follows.

So, the B operand mux IMMED is a 16-bit-wide selection of either BREG, 0_{15:4} || IR_{3:0}, sign_{15:4} || IR_{3:0}, or IR_{11:0} || 0_{3:0} ("||" means bit concatenation).

I used an unusual 2-1 mux with a fourth "force constant" input for this zero/sign extension function, primarily because it fits in a single 4-LUT. So, as with FWD, IMMED is an 8-CLB column of muxes.

The B operand register uses IMMED's CLBs 16 flip-flops. The register has separate clock enables for B_{15:4} and B_{3:0} to permit separate loading of

the upper and lower bits for an imm prefix.

For sw or sb, read the register to be stored, via BREG, into DOUT_{15:0}, another column of eight CLBs flip-flops.

ALU

The arithmetic/logic-unit consists of a 16-bit adder/subtractor and a 16-bit logic unit, which concurrently operate on the A and B registers.

LOGIC computes the 16-bit result of A and B, A or B, A xor B, or A andnot B, as selected by LOGICOP_{1:0}. Each logic unit output bit is a function of the four inputs A_i, B_i, and LOGICOP_{1:0} and fits in a single 4-LUT. Thus, the 16-bit logic unit is a column of eight CLBs.

ADDSUB adds B to A, or subtracts B from A, according to its ADD input. It reads carry-in (CI) and drives carry-out (CO), and overflow (V). ADDSUB is an instance of the ADSU16 library symbol, and is 10 CLBs high—one to anchor the ripple-carry adder, eight to add/sub 16 bits, and one to compute carry-out and overflow.

Z, the zero detector, is a 2.5-CLB NOR-tree of the SUM_{15:0} output.

The shifter produces either A>>1 or A<<1. This requires no logic, so mux simply selects either SRI || A_{15:1} or A_{14:0} || 0. SRI determines whether the shift is logical or arithmetic.

Listing 2—Here's the xr16 assembly code (with comments added) that gcc generates from Listing 1. gcc has done a good job, although a few register-to-register moves are unnecessary.

```

_search: br L3          ; r3=k r4=p
L2:     lw r9,(r4)
        cmp r9,r3      ; p->k < k?
        bge L5
        lw r4,4(r4)    ; p = p->right
        br L6
L5:     lw r4,2(r4)    ; p = p->left
L6:L3:  mov r9,r4
        cmp r9,r0      ; p==0?
        beq L7
        lw r9,(r4)
        cmp r9,r3      ; p->k != k?
        bne L2
L7:     mov r2,r4      ; retval = p
L1:     ret

```

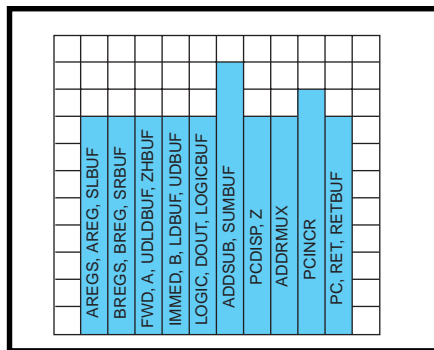


Figure 4—In the datapath floorplan, RLOC attributes applied to the datapath schematic pin down the datapath elements to specific CLB locations. The $RESULT_{15:0}$ bus runs horizontally across the bottom eight rows of CLBs.

RESULT MULTIPLEXER

The result mux selects the instruction result from the adder, logic unit, $A \gg 1$, $A \ll 1$, load data, or return address. You build this 16-bit 7-1 mux from lots of 3-state buffers (TBUFs). In every cycle, the control unit asserts some resource's output enable, driving its output onto the $RESULT_{15:0}$ long line bus that spans the FPGA.

In the third article of this series, I'll share the CPU result bus as the 16-bit on-chip data bus for load/store data. During sw or sb , the CPU drives $DOUT_{7:0}$ and/or $DOUT_{15:8}$ onto $RESULT_{15:0}$. During lw or lb , the selected memory or peripheral drives the load data on $RESULT_{15:0}$ or $RESULT_{7:0}$.

ADDRESS/PC UNIT

This unit generates memory addresses for instruction fetch, load/store, and DMA memory accesses. For each cycle, we add $PC += 2$ to fetch the next instruction. For a taken branch, we add $PC += 2 \times disp8$. For jal and $call$, we load PC with the effective address SUM from ADDSUB.

Refer to Figure 3 to see how this arrangement works. PCINCR adds PC and the PCDISP mux output (either +2 or the branch displacement) giving PCNEXT. ADDRMUX then selects PCNEXT or SUM as the next memory address.

If the next memory access is an instruction fetch, $ADDR \leftarrow PCNEXT$, and PCCE (PC clock enable) is asserted to update PC with PCNEXT. When the next access is a load/store,

SELPC and PCCE are false, and $ADDR \leftarrow SUM$, without updating PC.

PCDISP is a 16-bit mux of $+2_{15:0}$ and $2 \times disp8$, 5 CLBs tall. PCINCR is an instance of the ADD16 library symbol, 9 CLBs tall. ADDRMUX is a 16-bit 2-1 mux with a fourth input, ZERO, to set PC to 0 on reset. It's 16 LUTs, 8 CLBs tall.

PC is not a simple register, but rather it is a 16-entry register file. PC_0 is the CPU PC, and PC_1 is the DMA address. PC is a 16×16 RAM, eight CLBs tall.

I used RLOC attributes to place the datapath elements. Figure 4 is the resulting floorplan on the 14×14 CLB FPGA. Each column of CLBs provides logic, flip-flops, and TBUF resources.

THE DATAPATH IN ACTION

Next, let's see what happens when we run `0008: addi r3,r1,2`. Assuming that $PC=6$ and $r1=10$, PCINCR adds $PCDISP=2$ to $PC=6$, giving $PCNEXT=8$. Because SELPC is true, $ADDR \leftarrow PCNEXT=8$, and the next memory cycle reads the word at 0008. Because PCCE is true, PC is updated to 8.

Some time later, RDY is asserted and the control unit latches `0x2312` (`addi r3,r1,2`) into its instruction register. The control unit sets $RNA=1$, so $AREG=r1$. BREG is not used. FWD is false so $A=AREG=r1=10$. IMMOP is set to sign-extend the 4-bit imm field, and so $B=2$.

We add $A+B=10+2$ and as SUMT is asserted (low), we drive $SUM=12$ onto the RESULT bus. The control unit asserts RFWE (register file write en-

Assembly	Maps to
<code>nop</code>	<code>and r0,r0</code>
<code>mov rd,ra</code>	<code>add rd,ra,r0</code>
<code>cmp ra,rb</code>	<code>sub r0,ra,rb</code>
<code>subi rd,ra,imm</code>	<code>addi rd,ra,-imm</code>
<code>cmpi ra,imm</code>	<code>addi r0,ra,-imm</code>
<code>com rd</code>	<code>xori rd,-1</code>
<code>lea rd,imm(ra)</code>	<code>addi rd,ra,imm</code>
<code>lbs rd,imm(ra)</code>	<code>lb rd,imm(ra)</code>
(load-byte, sign-extending)	<code>xori rd,0x80</code> <code>subi rd,0x80</code>
<code>j addr</code>	<code>jal r0,addr</code>
<code>ret</code>	<code>jal r0,0(r15)</code>

Table 4—Many assembly pseudo-instructions are composed from the native instructions. Only rare signed char data use the rather expensive `lbs`.

Instruction(s)	A	B
add rd,ra,rb	AREG	BREG
addi rd,ra,i4	AREG	sign-extimm
sb rd,i4(ra)	AREG	zero-extimm
imm 0x123	ignored	imm12 0 _{3,0}
addi rd,ra,4	AREG	B _{15,4} imm
add ₁ r3,r1,r2	AREG	BREG
add ₂ r5,r3,r4	RESULT	BREG

Table 5—Depending on the instruction or instruction sequence, A is either AREG or the forwarded result, and B is either BREG or an immediate field of the instruction register.

able), and sets RNA=RNB=3 to write the result into both REGFILES' r3.

DEVELOPMENT TOOLS

This hardware was designed, simulated, and compiled on a PC using the Foundation tools in Xilinx Student Edition 1.5. I used schematics for this project because their 2-D layout makes it easier to understand the data flow because they offer explicit control and because they support the RLOC (relative location) placement attributes that are essential to floorplanning (to achieve the smallest, fastest, cheapest design).

To compile my schematics into a configuration bitstream, Foundation runs these tools:

- map: technology mapping—map schematic's arbitrary logic structures into the device's LUTs and flip-flops
- par: place and route—place the logic and flip-flops in specific CLBs and then route signals through the programmable interconnect
- trce: static timing analysis—enumerate all possible signal paths in the design and report the slowest ones
- bitgen: generate a bit stream configuration file for the design

HIGH-PERFORMANCE DESIGN

The datapath implementation showcases some good practices, such as exploiting FPGA features (using embedded SRAM, four input logic structures, TBUFs, and flip-flop clock enables), floorplanning (placing functions in columns, ordering columns to

reduce interconnect requirements, and running the 3-state bus horizontally over the columns), iterative design (measuring the area and delay effects of each potential feature), and using timing-driven place-and-route and iterative timing improvement.

I apply timing constraints, such as net CLK period=28;, which causes par to find critical paths in the design and prioritize their placement and routing to best meet the constraints. Next, I run trce to find critical paths. Then I fix them, rebuild, and repeat until performance is satisfactory.

I've built some tools, settled on an instruction set, built a datapath to execute it, and learned how to implement it efficiently in an FPGA. Next month, I'll design the control unit. 📧

Jan Gray is a software developer whose products include a leading C++ compiler. He has been building FPGA processors and systems since 1994, and now he designs for Gray Research LLC. You may reach him at jan@fpgacpu.org.

SOFTWARE

Visit the *Circuit Cellar* web site for more information, including specifications, source code, schematics, and links to related sites.

REFERENCES

- [1] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings, Redwood City, CA, 1995.
- [2] T. Cantrell, "VolksArray," *Circuit Cellar*, April 1998, pp. 82-86.
- [3] D. Van den Bout, *The Practical Xilinx Designer Lab Book*, Prentice Hall, 1998. (Available separately and included with Xilinx Student Edition.)

SOURCE

Xilinx Student Edition 1.5

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

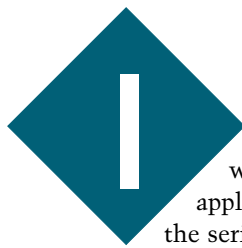
Killing Bugs in Your PalmOS

FEATURE ARTICLE

Jeff Stefan

Debugging with State Machines

You've written an application that uses the serial port on your Palm Pilot, but when you try to launch the app, nothing happens. If this situation sounds familiar, pay close attention as Jeff reveals an old embedded-systems programmer's trick.



Let's say you want to write an application that uses the serial port on your PalmPilot. Maybe you want to collect data or connect to a GPS receiver.

You've written the program using the Palm OS Serial Manager function calls, and now you're ready to test your new application. You plug in a serial adapter cable, launch your serial application, and nothing happens. Worse yet, your PalmPilot locks up so badly you have to pull the batteries and wait for the memory to discharge.

You can't step through your code with a debugger because the debugger uses the serial port to communicate with the Pilot. What do you do? It's time to employ a trick that embedded-systems programmers use—implicit debugging using state machines.

Embedded-system programmers have blinked LEDs and placed custom `printf()` statements in strategic places throughout their embedded code for years. We won't be blinking any LEDs from our Pilot (although it's possible). But, the example program writes status information to a debug display form in an orderly fashion.

The design also adds the structure of a state machine to the example program. After all the states are developed and debugged, the Palm OS Se-

rial Manager function calls are added, completing the program. If the program hangs up or faults in one of the states, the problem can be easily isolated and fixed. The line of code that caused the problem isn't explicitly displayed, but is implicitly detected if program loops in a state, skips states, or exits a state unexpectedly.

STATE MACHINES 101

It's worth taking a short side trip into theory and history to understand some of the fundamental theorems behind the machinery of computation. It's also useful to know what a computing machine can and can't do.

State machines aren't new, and their origin isn't in the domain of practical programming. State machines have their roots in automata theory, with the simplest machine being a deterministic finite acceptor.

A deterministic finite acceptor (DFA) is an abstract machine containing a set of finite states (functions enabling transitions from one state to another) and an associated set of input symbols taken from a finite alphabet. Deterministic finite acceptors have a neat and concise mathematical notation. A DFA (M) is expressed as:

$$M = (Q, \Sigma, \delta, q(0), F)$$

where Q is the finite set of states, Σ is an alphabet consisting of a finite set of symbols, δ is a transition function from $Q \times S$ to Q , $q(0)$ is a distinguished start state, and F is a final state.

A language, denoted $L(M)$, is the set of strings from the alphabet that cause the DFA to halt in the final state. Any other string isn't in the language.

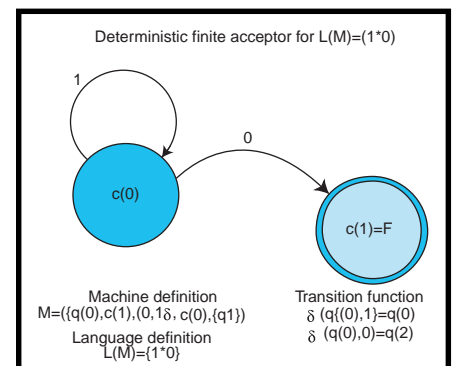


Figure 1—This DFA accepts a string consisting of any number of ones followed by a zero.

Consider the DFA in Figure 1. The language that this DFA accepts can be expressed as $L = \{1^*0\}$ where the star superscript represents one or more 1s. If this looks suspiciously like a regular expression, you're right. This is precisely classified as a regular expression: any string accepted by a DFA is a regular expression by definition. The simple DFA has no output or memory; either it accepts a string in the language, or it doesn't.

The DFA has two states, $q(0)$ and $q(1)$, an input alphabet of two elements 1 and 0, a transition function, and a final state of $q(1)$. A final state in an automaton is a double circle. This innocent two-state machine will never halt if presented with a singular 1 or an infinite stream of 1s. This arrangement has deep implications in the theory of computability, especially when machines are more complex.

To see why this simple loop has a profound impact on computing machinery, we need to step back to the dawn of computer science and visit one of the founders, Alan Turing.

The world changed profoundly in 1936, although few knew it. A 24-year-old Englishman named Alan Turing completed a paper called "On Computable Numbers with an Application to the Entscheidungsproblem" (Entscheidungsproblem is the German expression for decidability).

Decidability asks, "Can an algorithm be developed to solve a particular problem that yields a yes or no answer?" In his small, somewhat obscure paper, Turing described an abstract universal computing machine, now recognized as a Turing Machine.

The Turing Machine is at the foundation of modern computer science, and defines the notion of a computation. That notion is that the sequence of configurations in a Turing Machine leading to a halt state is called a computation. This means that if your algorithm runs and halts successfully on a Turing Machine, it's a computation.

A Turing Machine can represent any computational device, and Turing Machines can take other Turing Machines as in-

put. This is called a Universal Turing Machine, and here's where the deep problem lies. The question is, does an algorithm exist (given any Turing Machine and any input to the Turing Machine) that determines whether the Turing Machine finishes in a halt state? The answer, is no.

Looking back at the simple DFA that accepts an arbitrary number of 1s followed by a 0, if our input is unseen, we have no way of predicting whether the DFA will accept a string and wind up in a halt state. If it looks like the machine is in an infinite loop, it may be that it's processing an enormously long string of ones before it encounters a zero. There's simply no way of knowing, and that's the limit of computation as we know it.

USEFUL TOOLS

So much for theory! The mathematical model of a machine eventually emerged from theory to practice, and evolved into transducers. Transducers have output functions along with transition functions. These models were further refined into two types of basic machines—Mealy machines and Moore machines.

In a Mealy machine, all actions are performed during a transition. In a Moore machine, all actions occur inside a state. Mealy and Moore machines can be used together, there's no rule that the two machines need to be mutually exclusive.

The state machine shell function in this article follows the Moore Machine model because all the processing is done within a state. This machine shell is manifested as a simple `void` function, as shown in Listing 1. The

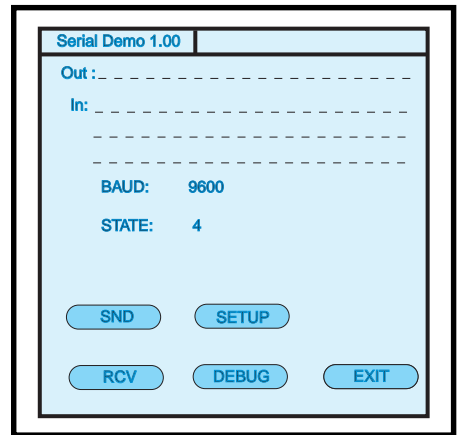


Figure 2—Here's what you'll see when the PalmOS is running the serial demo main form.

function contains two control variables, `Done` and `State`, which are both initialized to zero. If the global `DebugActive` variable is true, the code enunciates the current state.

The example shows specific PalmOS code that writes the current state to a form, but this code can be altered to blink an LED, or pulse an output line that can be captured on a logic analyzer or storage scope in any embedded application. When the last state is entered and the work performed, `Done` is set to 1 and `State` is set to -1, an unknown state, cleanly exiting the state machine.

When building parts of an application that can fit a state machine model (e.g., a communications protocol), first rename the function and determine the number of states it takes to complete the end-to-end processing. Then add the states, incrementing the `State` variable each step of the way.

Usually the number of states is small. If a state machine becomes too large, it means that more than one machine is required and the design should be re-examined.

What are the advantages of using this state machine form? First, the program contains a common framework and avoids a tangled mess of if-then-else statements. Second, using a switch statement is efficient because the only code that executes is the code that needs to. Third, a common debug output is constructed that displays the current state of the machine.

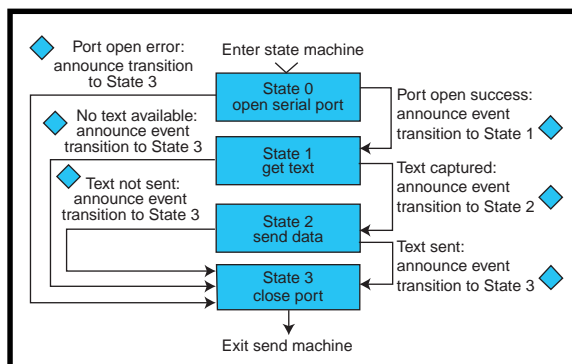


Figure 3—All that's happening here is the state writing a number to the main form STATE text field.

FILLING IN THE STATES

In the PalmOS serial application presented in this article, the end-to-end processing consists of opening and initializing the serial port, sending and receiving data, then closing the serial port. A screenshot of the application is shown in Figure 2.

There are two machines, the Send Machine and the Receive Machine. The Send Machine has four states and the Receive Machine has five. If you enter a string in the OUT text box and tap the SND button, the data is sent out the serial port at the current transfer rate. If you tap the RCV button, the application waits 5 s for data to accumulate in the port's buffer then outputs the data to the IN text box.

The DEBUG button toggles the DebugActive variable. When DebugActive is a 1, then the state numbers are displayed in the STATE field. The SETUP button toggles the transfer rate from 9600 to 4800 bps. If you want to receive data from a GPS receiver, 4800 bps comes in handy. Most GPS receivers output data at 4800 bps at 1-s intervals.

SEND MACHINE

The Send Machine state diagram is shown in Figure 3. There are four states inside the Send Machine—Open Serial Port, Get Text, Send Data, and Close Port. All of the processing, such as opening and initializing the serial port, is done within the states. The

Listing 1—The state machine shell is manifested as a simple void function.

```
void StateMachine(void)
{
    int State, Done;
        //init control variables
    State = Done = 0;
        //do any other preliminary initialization or other work here

    while(!Done)
    {
        switch(State)
        {
            //State 0
            case 0:
                //Do work that belongs to this state here. If debug
                //display flag is on, then display current state
                if(DebugActive)
                {
                    //Display current state. The following is specific to the
                    //PalmOS app, but any output mechanism can be substituted.
                    Frm = FrmGetActiveForm( );
                    FrmCopyLabel(frm, 1013,"0");
                }
                State = 1;
                break;

                //State 1: Exit State
            case 1:
                //This is the exit state in this two-state machine.
                //Do cleanup work here, such as closing any open
                //ports or resetting any flags.
                if(DebugActive)
                {
                    //Display current state. The following is specific to the
                    //PalmOS app, but any output mechanism can be substituted.
                    frm = FrmGetActiveForm( )
                    FrmCopyLabel(frm,1013,"1");
                }
                State = -1;
                Done = 1;
            }
            //switch
        }
        //!Done
    }
    //StateMachine
}
```

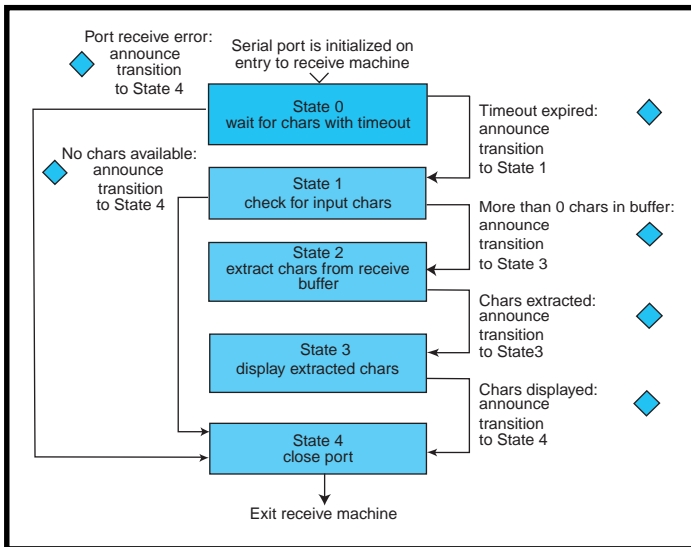


Figure 4—The Receive Machine is a bit more complicated than the Send Machine and requires one more state. The serial port is initialized before entry into the state machine simply to reduce the number of states.

The PalmOS Serial Manager functions are fairly simple and easy to use, with a few caveats. The PalmOS serial communications architecture is built on multiple layers of increasing functionality. The layer this article's example application uses is the bottom byte-level serial manager. All upper layer protocols go through the serial manager, so it's a good place to start.

There are a few things to be aware of when writing PalmOS serial applications. The first major issue is leaving the serial port open for extended periods of time.

The PalmPilot is based on the Motorola Dragonball MC68328 processor (which consumes little power), but leaving the serial port open rapidly drains the batteries. Leaving the port open for a minute or less at a time seems to work reasonably well, if the port remains closed for a minute or two. The basic rule is, keep the port open only when you need to, and close it when the application is terminated. Another application will not be able to enter the serial port if your application keeps it open.

The next item to pay close attention to is extracting received data from the receive buffer. By default,

the Serial Manager receive buffer is 512 bytes, so if an application needs a larger input buffer, it can call `SerSetReceiveBuffer()` and set a pointer to a new buffer.

The key is to restore the original buffer pointer when the application terminates because the PalmOS won't automatically return the buffer to the operating system. All that's required is to call `SetSerReceiveBuffer()` with the `bufsize` parameter set to 0, but it's an easy thing to forget.

The last item not to ignore is user input. It's easy to tie up the application while waiting forever for a character

edges are labeled with Event Enunciators, indicated by the diamonds.

Before inserting the actual PalmOS serial port function calls in the states, the function is filled in with the Event Enunciators, compiled, then executed. This process enables the program to enter, execute, and exit the state machine safely and reliably.

The next step is to open and close the port without errors. This completes the core end-point processing. The last steps consist of capturing the text from the OUT label and sending it out the port.

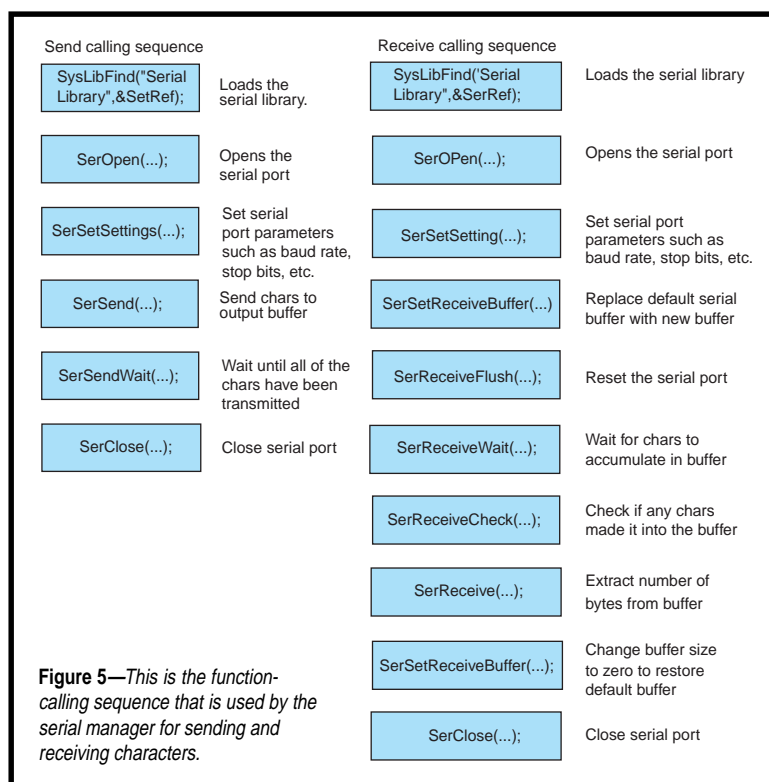
HARDWARE AND FUNCTIONS

The PalmPilot UART is somewhat scaled down from the typical PC fare. Instead of a 16-byte buffer, the Palm UART has an 8-byte buffer, and all of the usual serial control lines aren't present. The Palm UART signals are TxD (Transmit Data), RxD (Receive Data), RTS (Request to Send), CTS (Clear to Send), and Gnd (Signal Ground). The UART signals that are not supported by the PalmPilot include RT (Rise Indicator) and DTR (Data Terminal Ready).

RECEIVE MACHINE

On entry, the Receive Machine (shown in Figure 4) waits for incoming characters for a certain period of time. The next state checks to see if any characters entered the receive buffer. If no characters were received, the machine short-circuits and jumps to the exit state, which closes the port and exits the machine.

If characters are received, the next state extracts the characters for further processing. The next state displays the characters in the IN field, and the last state closes the port.



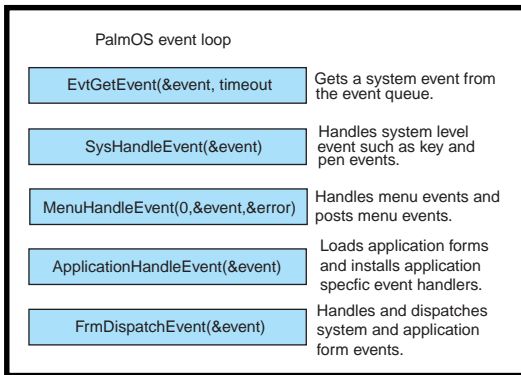


Figure 6—All PalmOSs process events in this fashion.

to enter the receive buffer. The only way out of this situation is to reset the PalmPilot.

There are generally two methods to handle user input while using the serial port. The first method, which is the one the example program uses, opens the serial port, waits a short period of time for characters to accumulate in the buffer, processes them, then closes the port. The port is only open when the user taps the RCV button.

The second method is to add a timeout parameter to main EventLoop function. Instead of using the default timeout parameter of evtWaitForever, a function pointer, symbolic value, or hardcoded value can be substituted instead. An event, called a nilEvent, is generated when the timeout expires. An application can utilize a nilEvent just like any other event.

nilEvent is commonly used in serial applications to periodically poll for serial data. In between the polling for serial data, the main event loop can still receive and process events in the normal fashion. This method is a bit more risky because it may leave the serial port open for longer than expected and therefore deplete the batteries at a faster rate.

The serial manager functions are called in a straightforward sequence and lend themselves to a state machine model and implementation.

The serial manager has recently been enhanced in latest version of the PalmOS to the new serial manager. The new serial manager is the third iteration of the PalmOS serial API which allows more than one serial

connection to be maintained at the same time, including the IR port. To stay compatible for versions from PalmOS 2.0 on up, it's safe to stay with the serial manager calls listed in Figure 5.

The PalmOS offers an event-driven API and a rich but simple set of GUI resources to work with. The events are processed in a specific order by a series of nested function calls, as you can see in Figure 6.

SERIAL PROGRAM SHELL

The application program, PalmSer.c, consists of seven PalmOS functions and three application-specific functions. These functions are:

- StartApplication()
- RomVersionCompatable()
- StopApplication()
- ApplicationHandleEvent()
- MainFormHandleEvent()
- EventLoop()
- PilotMain().

A PalmOS application is initialized by calling StartApplication(). StartApplication() is where you initialize global variables, check for operating version compatibility, open databases, and possibly establish any communications.

The PalmOS has evolved from V.1.0 to V.3.x. Most newer applications are not compatible with V.1.0 devices (e.g., early US Robotics Palm devices). Most PalmOS programs call RomVersionCompatable() and abort if the device is running a version of the PalmOS below V.2.0. A standard notification (an alert) is displayed, showing you that the application won't run on the current device. The alert is almost identical to a message box in a Windows application.

StopApplication(), the counterpart to StartApplication(), is the place to close an open serial port and close any open databases.

ApplicationHandleEvent() loads the application's forms and event handlers associated with the

forms. This function is a programmer-defined event handler and is application specific.

MainFormHandleEvent() is the event handler installed when ApplicationHandleEvent() is called. EventLoop() is the main event loop described in Figure 7.

And, last of all, PilotMain() is where the application is first entered, and is similar to main() in a ANSI C program. Like main(), parameters can be passed to the application. These parameters, called launch codes, can control the startup behavior of an application.

BUG FREE

Programming a PalmOS application takes patience and practice. Debugging PalmOS serial applications can be frustrating and time consuming. As forward as the function calling sequence is, there's still plenty of room for error and unexpected program behavior. Adding a straightforward, deterministic state machine shell and filling in the functions one at a time makes writing PalmOS serial-communication programs a pleasure. ☐

Jeff Stefan is a software engineer at Visteon. He has worked in embedded-systems software design for many years and is the author of 15 technical articles. Jeff is currently working on his first book, "Embedding Artificial Intelligence." You may reach him at jstefan1@visteon.com.

RESOURCES

- P. Linz, *An Introduction to Formal Languages and Automata*, D.C. Heath and Company, Lexington, MA, 1990.
- T.L. Booth, *Sequential Machines and Automata Theory*, John Wiley & Sons, New York, 1968.
- US Robotics, *Developing PalmOS 2.0 Applications*, 1997.

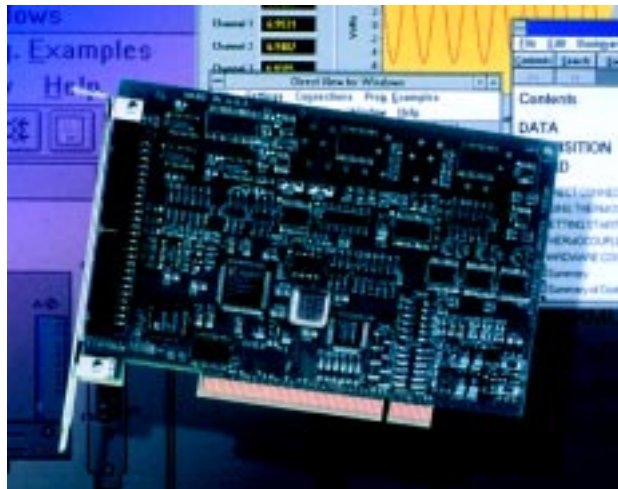
SOURCE

Dragonball MC68328
 Motorola
 (512) 328-2268
 Fax: (512) 891-4465
www.mot-sps.com

PCI DATA ACQUISITION BOARD

An ultra low-cost line of data acquisition boards for the PCI Bus has been announced by ADAC. The **PCI-5500** family consists of 12- and 16-bit resolution analog input boards that include performance features such as FIFOs, DMA, channel gain RAM, autocalibration, clocked analog output options, and a custom PCI bus interface for high-speed data transfers.

The PCI-5500 line consists of three models. The PCI-5500 features a 12-bit ADC with a 100-kHz sampling rate, eight analog input channels, 1024 word A/D FIFO, DMA, and interrupts. A programmable A/D pacer clock, multiple triggering modes, and 16 digital I/O lines are also included. The PCI-5501MF is a high-performance board that features a 12-bit ADC with a 100-kHz sampling rate, eight differential or 16 single-ended analog input channels, a 1024 word A/D FIFO, DMA, and interrupts. In



addition, the board offers programmable gain (high and low gain options), channel/gain RAM, programmable A/D pacer clock, multiple triggering modes, a dual 12-bit clocked DAC option, and 16 digital I/O lines. The PCI-5503HR is a high-resolution version of the PCI-5501MF featuring a 16-bit ADC with a 200-kHz sampling rate.

The PCI-5500 line is fully compatible with most popular data acquisition software packages, including LabVIEW, TestPoint, LabTech Notebook, and LabTech Control. In addition, the PCI-5500 line includes ADAC's ADLIB series of drivers for custom programming under WindowsNT, Win98, and Win 3.1. All PCI-5500 software drivers are free, and can be downloaded from ADAC's web site.

ADAC
(800) 648-6589
Fax: (781) 938-6553
www.adac.com

SINGLE-BOARD COMPUTER

The **VNS-786** is a single-board computer that is based on the Intel TX chipset. Designed for embedded applications, it supports the Pentium, Pentium MMX, and Tillamook processors from Intel and the K6-II and K6-III processors from AMD. Expansion is available through a PC/104-Plus embedded bus.

The VNS-786 supports up to 128-MB of 3.3-V SDRAM memory and has a 512-K synchronous burst level-2 cache. Major onboard peripherals include PCI SVGA, 10/100 Mbit Ethernet, and audio subsystems. The onboard SVGA comes with 2-MB video memory standard (optional 4 MB). It features both 3-V and 5-V flat-panel display support, full-motion video input, and simultaneous CRT/flat-panel operation. The onboard audio is SoundBlaster Pro and Windows sound system compatible, and



includes bridge-mode power amplifier. Disk interfaces include two Ultra DMA EIDE and one floppy disk. Also onboard are four deep-FIFO serial ports (with RS-232 and RS-485 support), two enhanced parallel ports, PS/2 keyboard, and mouse, and two USB ports (with overcurrent protection). A 32-pin JEDEC socket supports DiskOnChip and other solid-state disk solutions. Other embedded features include no-battery operation and a watchdog timer. The VNS-786 supports 5-V only operation and control of ATX power supplies.

The VNS-786 with 233-MHz Pentium MMX CPU is available for **\$550**, in quantities of 100.

Adastra
(510) 732-6900
Fax: (510) 732-7655
www.adastra.com

Nouveau PC

edited by Harv Weiner

A Matter of Time

Part 3: Synchronizing a PC to a Time Signal

There are ready-made solutions as well as totally home-grown approaches available for most engineering problems. As Ingo concludes this series, he shows us that sometimes it makes sense to adopt a mixture of both.

As you know, there are a lot of things one can do with time, including wasting it, so let me finish this series by explaining how you can synchronize an embedded PC to a time signal using a WWVB time receiver from Ultralink. Ultralink makes several versions of time-code receivers. I'm looking at the 320BS, which is distributed by Parallax specifically for attaching to a Basic Stamp.

LAST TIME...

To recap, WWVB is a time station in Colorado operated by the NIST. WWVB broadcasts on a carrier frequency of 60 kHz which, unlike its sister stations in the shortwave band, is not as affected by atmospheric conditions.

At this frequency, signals tend to travel near the surface of the Earth and

follow its contours, like a big waveguide. Because it's not scattered and reflected like shortwave signals, the propagation time is more predictable.

Also, because the wavelength is very long, the signal penetrates most buildings and structures. With a GPS microwave signal, by contrast, I need to have a line of sight from the antenna to the satellite.

Most shortwave radios don't tune below 500 or 150 kHz, and many serious shortwave radios don't tune below 60 kHz. That's probably not a big loss because there are no voice transmissions in the very low frequency (VLF) band.

If you listened to WWVB, you would hear a faint 100-Hz hum. [1] Well, I don't have a VLF-capable receiver, but I was

curious, so I built the converter for a shortwave receiver shown in Figure 1. This converter (or mixer) mixes the frequency of a local oscillator—in this case, a 4-MHz TTL oscillator with the input signals from the antenna.

The converter translates the signals from the VLF band to 4 MHz and above. Its image is at 4 MHz and below. Figure 2 shows the spectrum conversion.

Command	Response	Function
0x01	RX YY DD HH MM SS mm	Read current time
0x02	S	Diagnostic receive
0x03	—	Force update
0x04	tc	Read UTC time correction
0x05	rv	Return version/revision
"A"	R X Y Y L T D D H H M M S S m m	Read current time
"B"	S	Diagnostic receive
"C"	—	Force update
"E"	t c	Read UTC time correction
"F"	r v	Return version/revision

Table 1—These are the commands and responses for the Ultralink atomic clock receiver module. If you send the detector a BCD number (0x01–0x05) or an ASCII character (A–F), the response will be either a BCD or ASCII string, respectively.

Shortwave receivers are usually quite sensitive around 4 MHz, but you can translate the signal to any frequency you want by changing the local-oscillator frequency. The real purpose of the low-pass filter is to prevent signals near 4, 8, 12 MHz, and so on, from interfering with the VLF signal you want to hear. A properly balanced mixer can't put a signal "back" out the antenna.

If you do it right, you should get the same kind of beating hum as in the sound file in [1], assuming your receiver's audio section can reproduce the 100 Hz. Once you're this far, you can use an LM567-based tone detector to detect the 100-Hz tone and generate a digital facsimile of it.

CHECK THE CLOCK

In Part 1, I mentioned that WWVB is just a form of IRIG-H signaling and last month I covered the general IRIG protocol.

WWVB transmits its time codes at 1 bps with a 60-kHz carrier being directly modulated with the IRIG time signal, using 10-dB power reduction. The pulse-width relationship is 20/50/80%, which indicates 0, 1, and reference mark, respectively. Actually, it's almost slow enough that you can decode the signal by hand.

I did just that, and it worked OK. The decoded signal was pretty noisy. If you're interested in just a radio clock and don't really care how it all works, you're probably better off getting a radio-clock module like the 320BS. It's quite sophisticated, works when you plug it in, and doesn't tie up your shortwave receiver.

The Ultralink clock comes in two pieces—a receiver/antenna unit and a host-interface module (see Photo 1).

A three-wire cable connects the receiver and host-interface modules to provide power to the circuitry on the receiver module, as well as the response to the received signal from the receiver module to the host-interface module.

Having a separate receiver/antenna module means you can move the modules away from electrical noise generators, such as motors or poorly shielded monitors. In fact, the cable can be 200' long.

Besides the host-interface connection to the antenna/receiver module, the cable makes the host connection via an eight-

Listing 1—The PPS low-level driver responds to PPS signals on parallel port interrupt lines and measures the period of the system clock.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <rtl_fifo.h>
#include <rtl_sched.h>
#include <rtl_sync.h>
#include <asm/rt_irq.h>
#include <asm/rt_time.h>
#include <rtl_sched.h>
#include <asm/io.h>
#include <linux/cons.h>

#undef LPT_PORT
#undef LPT_IRQ

#ifdef xxx
#define LPT_PORT 0x378          /* standard */
#else
#define LPT_PORT 0x278          /* lpt1 on VSBC-1 */
#endif
#define LPT_IRQ 7

#define PERIOD (RT_TICKS_PER_SEC)
RTIME clkdifftot;          /* keep a running total of clock diffs */
unsigned clkdiffn;          /* and how many we have sampled */
RTIME ck_before,ck_now;

/* Interrupt handler—synchronize period timer with start of first word */
void irq_handler(void)
{
    ck_now = rt_get_time();
    if(ck_before){
        clkdifftot += ck_now - ck_before;
        clkdiffn++;
    }
    ck_before = ck_now;
}

/* User I/O handler—user sends int, compute average clock ticks/s and return */
int user_io_handler(unsigned int fifo)
{
    int cmd;
    unsigned avg;
    rtf_get(2,&cmd,4);
    if(cmd == 0){
        clkdifftot = 0;
        clkdiffn = 0;
        ck_before = 0;
    }
    if(clkdiffn)
        avg = clkdifftot/clkdiffn;
    else
        avg = 0;
    rtf_put(1,&avg,4);
    return 0;
}

/* Initialize module—set up FIFOs and interrupt/user-I/O handlers */
int init_module(void)
{
    int old_irq_state;

    printk("Starting PPS Module\n");

    /* create some FIFOs */
    rtf_create(1,4);          /* from drv to user */
    rtf_create(2,4);          /* from user to drv */
    rtf_create_handler(2,&user_io_handler);

    /* init the interrupt handler */
    outb(0x00,LPT_PORT+2);    /* disable */
    rtl_no_interrupts(old_irq_state);
    request_RTirq(LPT_IRQ, irq_handler);
    rtl_restore_interrupts(old_irq_state);
}
```

(continued)

Listing 1—continued

```
/* init the variables */
clkdiffTot = 0;
clkdiffn = 0;
ck_before = 0;

/* OK, start it up. */
outb(0x10,LPT_PORT+2);          /* enable */
return 0;
}

/* Cleanup module—remove all of the stuff we allocated */
void cleanup_module(void)
{
int old_irq_state;
printk("Stopping PPS Module\n");

/* shut off the interrupt */
rtl_no_interrupts(old_irq_state);

/* remove FIFOs */
rtf_destroy(1);
rtf_destroy(2);

/* free up the interrupt handler */
outb(0x00,LPT_PORT+2);          /* disable IRQ */
free_RTirq(LPT_IRQ);

/* we're done */
rtl_restore_interrupts(old_irq_state);
}
```

pin single-row header connector that carries 5–15 VDC power/ground. The interface module takes the demodulated signal from the receiver/antenna module and synchronizes its internal clock to the time signal. It also decodes the data transmitted to maintain a real-time clock.

After the module locks on to the time station, it generates a PPS signal that is synchronized so the rising edge happens at the beginning of each second. The host computer then polls the module via a TTL-level asynchronous serial interface for the

time of day and other information. The result comes in either BCD or ASCII format, depending on the format of the command (see Table 1).

MOMENT OF TRUTH

To test the module, I connected the modules with 6' of shielded cable and powered it up. I didn't think I would be able to pick up anything; my lab is mostly at basement level. I also have several monitors and usually at least one or two running computers without skins on.

Listing 2—This short program reads the clock period from the PPS low-level driver.

```
#include <stdio.h>
#define WRFIFO "/dev/fifo1"
#define RDFIFO "/dev/fifo2"
main()
{
FILE *rdfp,*wrfp;
int cmd;
if(!(wrfp = fopen(WRFIFO,"w"))){
perror(WRFIFO);
exit(1);
}
if(!(rdfp = fopen(RDFIFO,"r"))){
perror(RDFIFO);
exit(1);
}
cmd = 1;
fwrite(&cmd,4,1,wrfp);
fread(&cmd,4,1,rdfp);
fclose(rdfp);
fclose(wrfp);
printf("Average Number of Ticks = %d.\n",cmd);
exit(0);
}
```

On powerup, it synchronized itself to the clock. Once synchronized, it woke up once per hour and resynchronized itself to the radio clock.

I live about 1000 miles away from the time station. Yet, once the clock has been turned on and it synchronizes the PPS signal, it is synchronized to every atomic clock in the world.

Being skeptical, I attached one logic probe that generates the obnoxious high/low and pulse tone to the PPS signal of the atomic clock receiver and tuned one of my shortwave receivers to WWV in the shortwave band. Sure enough, I couldn't tell the difference between the second tick of the WWV and the logic probe.

A GPS receiver compensates for the propagation delay so its PPS output is several milliseconds earlier than the other receivers. Not audible, but clearly visible on a scope.

Once the system is connected to my PC, there are two pieces of information I'm interested in. I'd like the current time of day and I also want to calibrate my PC



Photo 1 — The receiver unit contains all the analog circuitry to receive and demodulate the WWVB radio signal. The module also has a loopstick antenna.

clock. To find out the current time of day, I just have to attach the serial module to the PC's serial port. All that is needed is a TTL-to-RS-232 serial converter.

The Ultralink module uses 3-V TTL-level serial I/O and a 3–15-VDC power supply. I needed to convert these signals to RS-232 levels to use them with a PC. I used the self-powered RS-232-to-3-V TTL converter shown in Figure 3. The module only uses 600 μ A, so we can also try stealing power from any unused RS-232 signals.

The rectified signal's 12-V supply is somewhere between 5 and 10 VDC on most PCs. We use this to power the Ultralink

module with a 10- μ F capacitor to buffer it. If the PC can't supply enough current to power the module, you can increase the value of the capacitor to about 100 μ F.

The serial I/O signals are converted using a couple of 2N2222 general-purpose transistors. They provide the necessary level conversion and logic inversion.

To power the 3-V serial receive line on the module, I used a low-dropout 3-V voltage regulator that drops the 5–10 VDC derived from the DTR/RTS pins to the required 3 V. You can change the 2.2-k Ω pullup resistor if the circuit draws too much current from the RS-232 interface.

A 74xx04 inverter inverts the PPS signal from active high to active low, as needed by the parallel port interface to generate a falling-edge interrupt (see Figure 3). You might note that there is a generic 7404 gate shown, but for proper low-power operation, this should be a 74HC04 or equivalent CMOS converter.

Once attached via the serial port, the host sends a read time command (either 0x01 or "A"), and the module responds with a message in BCD or ASCII format, respectively. One of the pins on the host-interface module host connector is a data-rate pin—open for 9600 bps or short it to ground for 2400 bps.

TIME OF ARRIVAL?

The module responds to a command within 5 ms. This is important, because we need to know the latency between the request and when the host receives the data. In other words, we need to know the time it takes to transmit the command, the turn-around latency of 5 ms, and the time taken to send the response:

$$T_{cmd} = 10 \times \frac{1}{9600} = 1.04 \text{ ms}$$

$$T_{lat} = 5 \text{ ms}$$

$$T_{resp} = 70 \times \frac{1}{9600} = 7.29 \text{ ms (in BCD)}$$

To send the command and get a time back, takes:

$$T = T_{cmd} + T_{resp} + T_{lat}$$

$$= 1.04 + 5.00 + 7.29 \text{ ms}$$

$$= 13.33 \text{ ms}$$

I'm trying to avoid the ambiguity that arises when we send the command to read the time right before the second tick. In this case, we can't be sure if the time sent back is for the second that occurs before or after the tick.

But, if we send the command 13.33 ms before the expected second tick, we know that the response is in the current second interval. Of course, we also have to take into account the latency in processing serial data in the host's OS, which adds some time to the estimate.

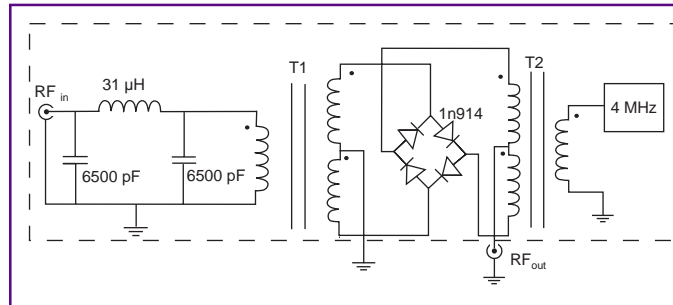


Figure 1—In the VLF converter circuit, there is a low-pass filter, preferably tuned to pass only signals below the AM broadcast band (i.e., 500 kHz), followed by a diode ring doubly balanced mixer (DBM), which is driven by a TTL/CMOS clock oscillator.

SIGNALING THE TIME

To calibrate the PC's clock, I have to measure time more accurately than the second count available via the serial port. I can do this with the PPS signal on the connector of the host-interface module.

The PPS signal is synthesized on the Ultralink clock module by locking an oscillator to the 100-Hz signal of the WWVB transmission. The signal detects

the beginning of the modulated pulse's pulse, which starts the PPS pulse with a rising edge.

The rising edge of the pulse is used as the timing reference mark.

The accuracy of the edge depends on how accurately the module's oscillator can be phase-locked to the carrier and how accurately the start of the timing pulse can be detected.

The module uses several cycles of the pulse to enhance the accuracy and boasts an accuracy of ± 20 ms. The clock may drift up to 0.02 s/h between synchronization updates. This technique is much better than my attempt to use a VLF converter, shortwave receiver, and LM576-based tone detector.

To use this PPS signal to measure the PC clock, I attach it to the ACK line of the parallel port. This line, if you remember from "Parallel Port Interfacing" (*Circuit Cellar* 113), generates an interrupt request on a high-to-low transition. So, I have to attach an inverter to get the correct sense from the PPS signal, because its reference edge is low to high.

Once this is accomplished, we can generate one interrupt per second on the PC. To test this setup, I wrote a program to measure the number of clock ticks between seconds. The source code in Listing 1 is implemented as an RT-Linux module.

I start with `init_module`, which initializes some FIFOs to communicate with a user-level application. FIFOs, if you recall from the article series "Embedded RT-Linux" (*Circuit Cellar* 100–104), are how real-time threads communicate with

non-real-time processes under RT-Linux. Next, I install an interrupt handler and turn on the parallel port interrupts.

The interrupt handler records the exact clock tick it was invoked in. Some latency occurs between when the interrupt really occurred and the time the code is invoked. This interrupt response latency can be measured and is deterministic under Linux.

At worst, it will be about 10 μ s, which is the overhead introduced by the scheduler. However, it's not going to be much of a factor, because the clock module accuracy will be ± 20 ms.

After the first time it's invoked, the interrupt handler computes the time between this invocation and the last time it was invoked (presumably the last time it was called). I accumulate the result because I want to compute the average period. I also record the number of sample periods collected.

I registered a handler that wakes up when there's activity on the receive FIFO, indicating that a user application wants to read the average time. The application sends a zero word to reset the totals. Any other command value just computes the average and returns the result.

The application software is short, as you see in Listing 2. It opens the two FIFOs from the user side—one for read, the other for write. It then sends a nonzero word to the user side and reads the result. The result is printed to the screen.

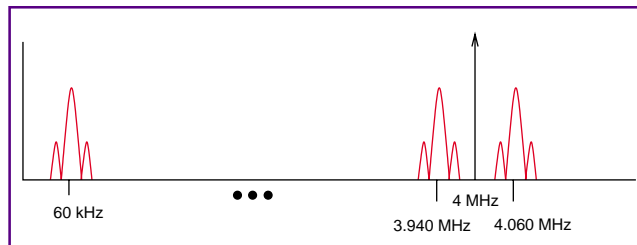


Figure 2—Here, the VLF band (0–500 kHz) is shifted by 4 MHz. The result is a band at 4.0–4.5 MHz and its image band at 4.0–3.5 MHz.

system constant (`RT_TICKS_PER_SECOND`). Replace this constant with our computed constant and the period of the IRIG signal should match the period of the PPS signal.

To align the IRIG signal with the PPS signal (i.e., for both signals to start on a second boundary), use the PPS interrupt service routine to kick off the IRIG task. A more advanced algorithm would check the starting time of the IRIG frame with the PPS signal and minimize the difference.

OUTTA TIME

As with any engineering problem, sometimes it makes sense to adopt a mixture of both ready-made and home-grown solutions. Instead of using my custom time-code receiver, I decided to adapt a receiver module to my system.

Companies like Datum would be glad to sell you a complete turnkey time reference system, which might be the best solution if you require high accuracy and stability, and if you have a budget to match. In the end, of course, only you know your application best. RPC.EPC

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

REFERENCE

[1] www.ezomm.com/~cyliax/wwwb.wave

SOURCES

Basic Stamp, 320 BS
Parallax, Inc.
(888) 512-1024
(916) 624-8333
Fax: (916) 624-8003
www.parallaxinc.com

Time reference systems

Datum, Inc.
(949) 598-7500
Fax: (949) 598-7555
www.datum.com

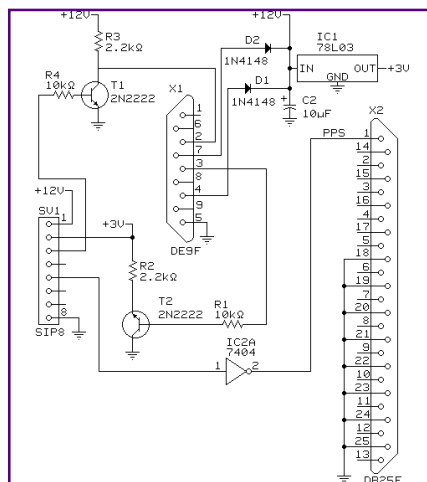


Figure 3—Two diodes steal power from the DTR and RTS signals, one of which needs to be at a 12-V level to power this circuit.

ON THE DOT

Once we figure out the average tick rate the system runs, we check to see if it's accurate to better than 20 ms. There should be 1,193,180 ticks/s.

If we measure 1,374,281 ticks/s, the system clock is running fast. It thinks each real second is about 1.15 s long, which is 150 ms longer than it should be. That's clearly worse than the ± 20 -ms accuracy that we can rely on from the atomic clock.

The simplest fix is to use the figure of 1,374,281 ticks as the prescale for computing time intervals in RT-Linux. Last month, I was able to generate IRIG codes by programming RT-Linux to periodically wake up a task. The calculation used a

Applied PCs

Fred Eady

Getting the Databoot

Not only is DOS still alive, but as Fred shows us, Arcom and Datalight have teamed up to provide a great deal of functionality in a hidden ROM-DOS partition that is Kryptonite-resistant and ready at a moment's notice.

Here we are, well into the year 2000, DOS is supposed to be dead and Windows NT is supposed to relinquish its throne to Windows 2000.

In fact, many Windows fanatics pronounced DOS dead years ago. You know that famous old line, "The rumors of my demise have been greatly exaggerated." Hey, at least you aren't still entering your programs using 80-column punch cards, or worse, cassette tape.

The world is still waiting for the Y2K-bug destruction, and thanks to Datalight, DOS is still very much alive. If you think about it, before FAT32 and the "non-DOS Partition," you had to load some minimal DOS code to get Windows 3.xx or 9x on your hard drive.

Today, we have Windows NT, Windows 2000, 98, and some Windows 95 and Windows 3.11 running out there. There's also Windows CE and Windows NT Embedded. I'm quite sure that none of these hardy

OSs have ever crashed, but what if Windows NT Embedded sucked up a corrupted file and spewed chunks?

And what if the machine that just lost its cookies is in California, and you're not? What if I told you that you could use DOS to recover the application and OS, via the Internet? Read on, Grasshopper.

LOOK, UP IN THE SKY

It's a bird, it's a plane, it's SuperBoot! Hidden within a hard disk or flash memory, SuperBoot could be thought of as invisible. SuperBoot is really a feature and extension of Datalight's ROM-DOS. This "invisible" partition is a hidden ROM-DOS partition, residing on the same physical media as the primary OS.

The idea is to stuff as much functionality into the hidden ROM-DOS partition as possible. When the primary OS fails or the hardware goes, the hidden ROM-DOS partition can be activated to provide a stable diagnostic platform. From this partition, all the emergency tools that were preloaded can be activated.

Although there's good reason to include things like a TCP/IP stack or special diagnostic routines, you can opt to establish a remote connection with the failing machine and manipulate diagnostics or download good files. And, who says the embedded



Photo 1—The blue box may not be as full as before, but the embedded Elan-104NC is carrying quite a powerful load.

device has to be on the blink? You can use this technology to download an upgrade or grant permission to other parts of your code.

For those of you new to SuperBoot, it works like this. There must be some type of monitor or OS (Windows, QNX, Linux, etc.) on spinning electromagnetic media, silicon flash, optical disk, or in read-only memory that establishes the hardware and system environment when an embedded PC hardware is booted.

Using a method called double booting, SuperBoot is activated via a hotkey sequence at boot time. This process allows ROM-DOS to boot and load from its hidden partition. Because this hidden ROM-DOS partition is not recognized, and therefore ignored by the primary OS, it lies dormant until it is activated.

Besides ROM-DOS and the SuperBoot code, Datalight Sockets (a TCP/IP stack designed to run under ROM-DOS) can also reside in the hidden partition. A TCP/IP stack means Internet connectivity.

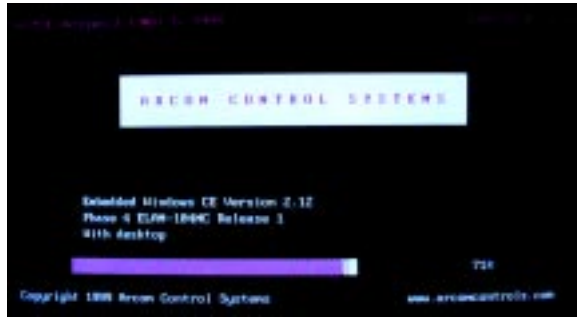


Photo 2—This screen holds bunches of “what am I” info. It’s good reading while Windows CE initializes.

Including Datalight Sockets in the hidden partition allows a TCP/IP portal to be implemented, which then allows remote access to the failing code or hardware. The on-site technician can call home, or the main system (that never crashed on January 1) can take over and remotely stuff files down the failing machine’s throat.

MORE ON ROM-DOS

I like DOS. In your daily routine, notice the guys who open a Windows command prompt to copy files or view directories. Those guys love DOS, too. ROM-DOS is the embedded programmer’s DOS.

Your embedded hound dog doesn’t have to be PC-compatible to use the features of ROM-DOS. Any 80186 or higher CPU with about 10 KB of RAM and around 50 KB of ROM is welcomed by ROM-DOS. If you’re into the NEC V-series CPUs, no problem. All the goodies you know and love in Bill’s 6.22 are included in Datalight’s ROM-DOS.

There’s not much difference in the way ROM-DOS works versus Bill’s 6.22 version, but selectable device drivers are the key to ROM-DOS’s universal nature. Datalight offers a ROM-DOS development package to help you assemble a suitable ROM-DOS image. Using device drivers allows a ROM-DOS platform to be implemented on any system.

Both OSs use a CONFIG.SYS to process startup commands, but unlike DOS 6.22, ROM-DOS doesn’t require COMMAND.COM to boot. Datalight’s ROM-DOS supports both RAM and ROM disks, and even provides remote disk-drive access via the serial port.

There are ways to embed Microsoft’s DOS 6.22, but it’s not as easy as doing it with ROM-DOS. Bill’s 6.22 doesn’t like Windows’ long filenames, either. Datalight’s ROM-DOS kernel supports them with no problems. There are a lot of other features that set ROM-DOS apart from the good-old everyday DOS and, if you’re interested in delving into them, visit the Datalight web site.

For the embedded developer, one of the biggest advantages in using ROM-DOS is that it’s half the size of normal DOS when fully implemented. This small footprint enables ROM-DOS to be squeezed into the tightest embedded design. ROM-DOS 6.22 comes in at 73 KB versus 133 KB for a fully ROMed version of Bill’s 6.22. As well as being half the size, ROM-DOS’s average cost is half as much.



Photo 3—Not many Win CE icons here, but keep your eye on that background bitmap.

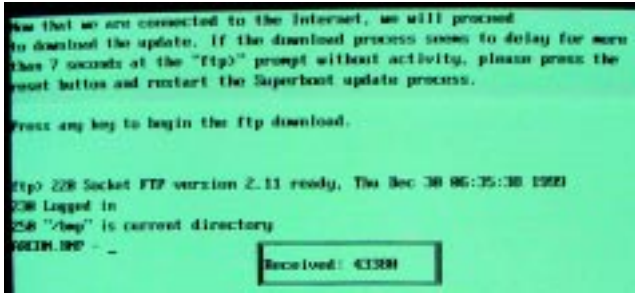


Photo 4—It's no crystal ball, but a hint of the future can be seen in the name of the image being downloaded.

FLASHDATALIGHT

These days, embedded and flash memory are often mentioned in the same breath. There are times when a hard disk or floppy drive just won't do. In those cases, flash memory steps in for the hard disk and sometimes even doubles as ROM on the system board.

The seemingly free flexibility that comes with flash memory does have a cost—additional software overhead. Datalight's engineers developed FlashFX to help integrate flash memory without paying the high cost of writing the overlord code.

Datalight's FlashFX software makes flash memory arrays appear as disk drives. This arrangement allows your application to use the flash memory area just as it would a physical disk drive. Your applications can access FlashFX arrays just as they would if the flash were spinning.

Unfortunately, flash media can wear out. FlashFX uses a process called wear leveling to extend the life of the flash memory array. The wear-leveling algorithm makes sure all the erasable flash memory area is used evenly, so recently written to or recently erased sectors, aren't used excessively.

FlashFX is written in ANSIC and doesn't really have a flash memory hardware preference. Flash memory parts from Intel

or AMD can use the features offered by FlashFX. Datalight offers a FlashFX porting kit that makes FlashFX portable to any microprocessor and embedded PC running Windows CE.

IN A BLUE BOX...

Last time we looked into a case like this, out popped a Datalight Thin Client, and my Florida-room porch extended all the way to Washington state. (Translation: porch is a place outside of the house, usually connected to the house much like a deck. You'll usually find rocking chairs, swings, and Grandma on the porch.)

What's in this box? Looks like an Arcom Elan-104NC to me (I'm not psychic—the model and type are written on the PC/104 connector). As you can see in Photo 1, there's also a 56k external modem. Let's see, there's a 10BaseT Ethernet, SVGA, mouse/keyboard, IDE interface, serial ports, and a flat panel. It looks like a slick Eurocard (100 mm × 160 mm).

After some Internet detective work, I determined that the Elan-104NC sports a 100-MHz AMD Elan 486SX processor with millenium-compliant Datalight BIOS.

The M16-F8 comes with 16 MB of DRAM (M16) and 8 MB of Intel Strata flash memory (F8). ROM-DOS 6.22 comes loaded, along with Datalight's FlashFX, and a 128-KB battery-backed SRAM disk.

If you need RS-485 capability, the Elan-104NC includes three serial ports with one doubling as your RS-485 driver. If you require a no-trouble Ethernet interface, the Elan-104NC employs a RealTek RTL8019AS 10BaseT Ethernet controller.

The label on the blue box says Elan-104 WinCE Development Kit. That must mean, in addition to the Datalight BIOS, Datalight ROM-DOS, and Datalight FlashFX, there is some Windows CE code in the flash memory, too.

PLUGGING ALONG

Taking a look at the Elan-104NC in Photo 1, you'll see that you really have to try if you want to plug something into the



Photo 5—Whether you're downloading bit-maps, files, operating system images, repair operations, or upgrades, using SuperBoot, ROM-DOS, and TCP/IP can make it happen.

wrong header. I don't mind scratching my head from time to time wondering which serial port is COM1, but wondering about power connectors doesn't give me a warm and fuzzy feeling. The Elan-104NC power connector (the green receptacle) is keyed and very much obvious as to what it is.

At powerup, I was greeted by the standard banners telling me who did what and how much memory was available. The Arcom/Windows CE install screen shown in Photo 2 immediately followed the Datalight banner screens. Once I got the Elan-104NC booted, I saw what you see in Photo 3—the Datalight bitmap on a Windows CE desktop.

At this point, I played around a little to see just what was loaded. I didn't find much, which is a good thing. After all, this is supposed to be a tight load containing only the application, ROM-DOS, and Windows CE modules needed to do the job.

I ran into an interesting situation while I was exploring. I tried to attach a standard 1.44-MB floppy drive to see if the Windows CE load would recognize it.

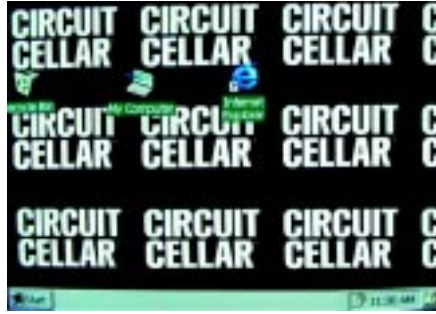


Photo 6—They say the third time's a charm!

Shucks, there's a floppy power connector grafted into the green Elan-104NC power connector. I figured it must be there for me to play with. And, the Elan-104NC supports standard floppy drives. Besides, I couldn't find a way to store and retrieve the screenshots I wanted to show you.

I should have known that what I was about to do was taboo, because I was locked out of the Datalight BIOS setup routines. Talk about foolproof! Users can't foul things up if they can't get to them.

I decided to go ahead and install the floppy drive. Maybe there's something hidden that I can exploit. Flip went the power to the Elan-104NC and blink went

the floppy drive LED. No problems so far—the Datalight banner and Windows CE desktop appeared on the monitor.

I opened a command prompt and attempted to get at the floppy. No luck, so I went to the Windows CE system setup. Still no luck. All right, I'll reboot. Maybe the hardware will be recognized then. No luck, and this time no display! My first word should probably not appear in print, but my first thought was that I'd damaged the load or worse yet, the Eurocard.

For the next few boots, I got nothing. I disconnected the floppy and rebooted. Again, nothing. The engineer (and idiot) in me said, "Put the floppy back on and try it one more time." I did and—wham—the display was back, but I still couldn't get to the floppy drive. Well, at least now I was working again. I could go on and describe the dog and pony show, but it would be great to get screenshots of the mutt and nag performance using the floppy drive interface. No dice. I had to resort to plan nine from outer space.

In the interim, I decided to connect the Elan-104NC to a larger monitor. In doing so, I forgot to reconnect the floppy. I powered everything up and things were back to normal. I think I know what happened, but I'm hesitant to go there with you (cockpit error). So, I'll move on and describe what the Windows CE application does and what Datalight resources it uses to accomplish the task.

TO THE PHONE BOOTH

My superhero used the corner phone booth to change clothes. We're going to do the same thing with the Elan-104NC.

The application that's loaded on the Elan-104NC demonstrates how SuperBoot, in conjunction with the Elan-104NC hardware, can be used to remotely recover or repair an ailing Windows CE-based application. The SuperBoot hidden partition was carved out using FlashFX when the flash memory was formatted. We already know that the Datalight BIOS drives the Elan-104NC hardware, and the Datalight ROM-DOS with the SuperBoot kernel is installed and hidden in flash memory, along with the CE OS.

The SuperBoot BIOS extension lies in wait, looking for a specific key sequence. When that key sequence (Alt-F4) is invoked during the initial boot phase, the SuperBoot partition (a ROM disk design-

nated as drive D) is booted instead of the primary Windows CE OS partition on flash memory drive C.

The SuperBoot boot sequence loads ROM-DOS, Datalight Sockets, and an FTP application. The idea is to show how the Elan-104NC, loaded with ROM-DOS, the Datalight SuperBoot feature, and Windows CE, can be configured to recover or completely rebuild the software image on the Elan-104NC via the Internet. Remember, although we're using Windows CE here, you can use any popular OS.

The Datalight folks conveniently provided an FTP site to download one of three desktop bitmap images to prove the remote reload and repair concept.

GOT A QUARTER?

After resetting the Elan-104NC, I initiated the Alt-F4 hotkey sequence during the "Testing RAM" phase. The Datalight SuperBoot banner informed me that I was booting from drive D. ROM-DOS was in control, Datalight Sockets was invoked, and a PPP connection was established via an 800 number to the Datalight FTP site. What I saw next is shown in Photo 4.

After completing the file transfer, I reset the Elan-104NC. This time, I left the Alt-F4 hotkey sequence alone, and the embedded PC booted from the primary Windows CE flash memory disk. As you can see in Photo 5, I now have a new bitmap image from the Datalight FTP site.

It just so happens that the bitmap I grabbed represents Arcom. The Arcom folks are responsible for the Elan-104NC hardware that I used to dial the FTP server to show you the downloaded image.

There's one more bitmap out there, and I'm anxious to see what it is. So, I initiated the Elan-104NC reset and performed the hotkey thing again. The FTP session kicked off and down came the image. Do you recognize the bitmap in Photo 6?

DOWNLOAD IT AGAIN, SAM

I repeated the FTP-download sequence dozens of times and it worked flawlessly every time. Playing with this demo conjured up all kinds of possible applications.

Wouldn't it be cool to have a system that, when requiring the services of SuperBoot, would call home to tell you about it? I picked up the phone and called Rob Krantz at Datalight and asked if they

had thought about that. Silly me. Datalight has an application that rides on ROM-DOS that sends e-mail at your command. It's called SENDMAIL.

In the old days, your disaster had to be in the presence of Clark Kent in order for Superman to save you. Nowadays, your embedded creation can practically jump into a phone booth and save itself using SuperBoot and ROM-DOS.

Thanks to Datalight and Arcom, once again we have proven that it doesn't have to be complicated, or from the planet Krypton, to be embedded. [APC](#). [EPC](#)

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

SOURCES

DataBoot

Datalight
(360) 435-8086 • Fax: (360) 435-0253
www.datalight.com

Elan-104NC

Arcom
(888) 941-2224 • Fax: (816) 941-7807
www.arcomcontrols.com

Mike Baptiste

Throw Away the Key!

An iButton Lock System

You know the routine: put the key into the lock, try to open the door, realize the key is upside down.... Mike came home with his hands full one too many times, so he set out to add an iButton lock system to his home automation setup.



People have been locking doors with keys for centuries.

What fairy tale would be complete without a locked door and a hidden key? Would you believe my house still has skeleton-key locks on all the doors (backed up with heavy-duty deadbolts, of course)?

Keys have evolved plenty, but the duty they perform is still the same. Insert the key, realize you have it upside down, flip it over, insert it again.... It isn't that hard, and we've done it forever. Nevertheless, when you have an armful of groceries, it would be really nice to get the door to unlock and open itself.

Electric door strikes have been around for years. They have improved some (they don't have to buzz if you get DC models), but the real chore is figuring out a device to control it.

Keypads work well, and you don't have to carry keys around. However, you still have to punch in the code. (Remember the armful of groceries?) Besides, in a densely populated area, someone with criminal intentions can try to see the code you punch in. Binoculars are easy to come by!

When I considered automating my door locks, I wanted to ensure they were secure, but I wanted something that would enable me to open the

doors with little effort. I decided to use iButtons from Dallas Semiconductor.

DON'T PUSH THIS BUTTON

Photo 1 shows some iButtons in a variety of holders, along with a simple probe for reading them. The basic concept is, you press the button into a probe and the data is read or updated using a simple 1-Wire interface. They require no power source because power is supplied on the same wire as the data. (See the datasheets and application notes for more info on parasitic power supplied via the data bus.)

iButtons come in a variety of formats, most with onboard, nonvolatile RAM. Their main application is to store digital information (up to 64 KB) in a small footprint. You can even get iButtons with temperature sensors, monetary logic and encryption, real-time clocks, and even Java.

Numerous 1-Wire devices can exist on the 1-Wire network at any given time. To tell them apart, every 1-Wire device, including iButtons, has a unique 64-bit serial number. That means 18,446,744,073,709,551,616 different serial numbers. Using that serial number as a key would be pretty secure! Technically, eight bits of the serial number are a checksum, and eight bits are a family code. Still, 48 bits adds up to a lot of guesses.

What about the RAM and Java that would be wasted? Worry not. Dallas has an iButton with a unique serial number and a 1-Wire interface. It makes an ideal key. Even if the thief knew how to interface to a 1-Wire network and tried to guess the serial number, they would have to make a lot of guesses. I had found my key.

STANDING ALL ALONE

As most of you know, I have an HCS-II home automation system in my house. Okay, I have a couple of them. The HCS-II can read iButtons using the AnswerMan Jr. network module (*Circuit Cellar* 78).

However, I really didn't want to devote one of my AnswerMan Jr. modules to reading iButtons. They were busy enough handling LCD displays and analog inputs. I needed to come up with a standalone controller

to store approved iButton serial numbers and unlock the doors when a valid iButton was read.

Touching the iButton to the probe is easy, reading the serial number and comparing it to an authorized list was another matter. It screamed embedded processor. Controlling the 1-Wire network wouldn't require a huge amount of CPU effort, but I also had to store the approved iButton serial numbers.

I'm familiar with PICs, so I immediately considered the '16F84. It has a reasonable amount of ROM and memory as well as 64 bytes of nonvolatile memory to store iButton serial numbers in. Now, all I needed was a name for this digital lock. DigiLock it is!

SPEAKING 1-WIRE

The biggest challenge of this embedded-control program would not be the main function. All the main function does is take the received serial number, compare it to a stored list, and return true or false. The fun part is having the interface actually request and receive the serial number. Supplying power and transferring data over the same wire can also be tricky.

Dallas designed the iButton with a parasite power circuit to keep the device powered during data transmissions. The 1-Wire network wire is pulled high so iButtons get power when they touch the probe. As long as the data bus is not driven low for too



Photo 1—Plastic key fobs as well as leather fobs and wallets will hold your iButtons. The probes are inexpensive and easy to connect. One wire is data and the other is ground. They can be mounted on any flat surface less than 10 mm thick.

long, the internal power circuit keeps the iButton powered until the bus returns to its idle high state supply external power again.

The Dallas 1-Wire network protocol requires the controller to act as a master and the iButtons as slaves. When an iButton touches the probe and powers up, it waits for the master to issue a reset pulse on the bus. Reset pulses must be 480 μ s (minimum). When the bus returns to its idle high state the iButton signals its presence by returning a presence pulse. This tells the controller there is at least one device on the network, and it is ready to be accessed.

It's unlikely that two iButtons will be touched to probes on my house at the same time, so I'll assume only one device is on the network at any time. (Read the 1-Wire application notes on Dallas' website for information on using multiple 1-Wire devices at the same time.)

Once the iButton replies to the controller with a presence pulse, the controller can then issue commands to it. The one command we care about is Read ROM. This will return the 64-bit serial number.

ALL IN THE TIMING

So far, it sounds easy. However, processing the serial numbers is not the tough part, requesting and reading the serial numbers. Remember, parasite power requires that the bus

not be driven low for too long. You can't just transmit bits across a 1-Wire network, because sending 0x00 will drive the bus low for too long. To handle this type of situation, iButtons and other 1-Wire devices use a protocol with specific bit timing.

Because the data bus is pulled high to supply power, a low pulse precedes each bit. This low pulse always comes from the master controller, even during data reads. When the master controller sends data to the iButton, the iButton samples the bus for the proper bit value between 15 μ s and 60 μ s after the bus is driven low.

Data is returned from the iButton in a similar fashion. The controller samples the bus between 1 μ s and 15 μ s after driving the bus low. This allows the iButton to synchronize the returned data to the master controller. Note that the 1-Wire bus is an open-collector driven bus because multiple devices may drive the bus low at any given moment. All highs are accomplished via a single pull-up resistor.

Consult the datasheet for the timing diagrams, which outline the Read and Write Time Slots. Listing 1 is the code used to read data from the iButton. Writing this routine in C made for a clean-looking code, but it also required that I check the op-code listing to ensure the timing was right. C compilers for PIC processors can do all sorts of tricks trying to handle bank switching and other PIC oddities.

The PIC never actually drives the bus high. Instead, it relies on a 4.7-KB pull-up resistor to bring the bus high again when it needs to send a 1. To send a 0, the PIC will drive the bus low. Even though it is discouraged for portability reasons, I used the TRIS command to toggle the tristate buffer on the 1-Wire data pin so it functions in an open-collector fashion.

With the code, I tried to ensure the bits were sampled in the middle of the given sampling window. You have to read data from an iButton in 15 μ s, which includes sending a start bit and sampling the returned data bit from



Photo 2—The DigiLock board is small enough to mount almost anywhere. The iButton status bits are brought out via the 2 x 8 header. For security, locate the DigiLock where it cannot easily be tampered with. If it's outdoors, use a weatherproof box with a lock. If you have an alarm system, wire a tamper sensor inside the box to trip the alarm if anyone opens the box.

the iButton. I used a 3- μ s start bit, a 3- μ s pause, and then sampled the bit.

The code samples the bit three times to ensure the proper value is read. Remember, touching the iButton to the probe is the same as holding two wires together to transmit data. Depending on how firmly you press the iButton into the probe and how still you hold it, who knows how much noise results from a weak connection.

The tricky part is that the start bit and bit sampling must occur in 15 μ s or less. Taking a cue from the common hardware UART circuit in bigger PICs, I wanted to sample the bit three times. Because I used a 3- μ s start pulse and allowed for 3 μ s before I sampled (to allow things to stabilize), that left 9 μ s for sampling. Because the PIC is running at 4 MHz with a divide-by-four clock, each instruction takes 1 μ s. This means there are nine instructions for sampling the bus three times and storing the result.

Using bit-test commands and a simple incrementing counter made for quick and efficient code. Doing it in C just makes it look slow. Listing 2 shows the resulting machine code, which is quite compact.

Once I sampled the bit, I had plenty of time to check the counter and see if it indicated a 1 or 0 bit. However, when I compiled the IF THEN ELSE statement to check the sample counter, it was huge. I reduced the code size out of principal (and fear of running out of PIC ROM space).

I soon realized that the second least significant bit in the sample counter would reflect the proper bit value. I just had to shift it into the buffer. Using an efficient bit test, I was able to test the sampled bit and store it in the buffer using five or six ticks.

Writing the bit out is easy. Just bring the bus low for 1 to 15 μ s and then go to the proper bit state for another 45 μ s while the iButton samples the bit. I take the bus low for 7 μ s and then shift out the bit onto the bus for another 50 μ s.

CHECK THAT CHECKSUM

Each iButton serial number contains an 8-bit checksum that lets you ensure that the received data is most

Listing 1—Reading in bytes from a 1-Wire device involves some tight timing. Sampling the bus multiple times helps ensure a valid read on the first try. If the counter hits 2 or 3, the code registers a 1 bit; otherwise it registers a 0 bit. Note that using an IF, THEN, or ELSE on the sample counter instead of the `shift_right` command resulted in bigger and slower object code.

```
// This routine reads in a byte from a microLAN device
// after a READ command has been sent

int read_byte() {

    int byte_in, i, bit_cnt;

    for(i = 0; i < 8; i++) {
        // Clear bit counter
        bit_cnt = 0;
        // Pulse microLAN low to clock bit
        set_tris_a(MLAN_OUT);
        mlan = 0;
        delay_us(3);
        set_tris_a(MLAN_IN);
        delay_us(3); // Allow some pullup recovery time
        // We now have ~7  $\mu$ s left to sample a few times
        if (!mlan) { bit_cnt++; }
        if (!mlan) { bit_cnt++; }
        if (!mlan) { bit_cnt++; }
        // Lets shoot for 2 out of 3!
        // Use a quickie bit test for efficiency
        shift_right(&byte_in, 1, !bit_test(bit_cnt, 1));
        // Lets allow for the microLAN device to release
        // (45  $\mu$ s MAX) and LAN to recover (15)
        delay_us(45);
    }
    return byte_in;
}
```

likely the data sent by the iButton. As I said earlier, the connection between the iButton and the probe is like holding two wires together. If you're in a hurry, it's worse.

While testing my code, I found that 1 out of 15 touches resulted in a bad read the first time. Note, I said touches. If the iButton was firmly attached to the probe, the data read never failed. However, casually pressing the iButton to the probe resulted in erroneous reads from time to time.

The checksum allows you to handle bad reads. Dallas Application Note 27 gives an in-depth overview of the 8-bit CRC used in 1-Wire devices. As the bytes are received, they are fed into the checksum loop. Listing 3 shows the code used to check the received checksum against the received serial number.

The checksum is calculated using the 8-bit family code and the 48-bit serial number. After each byte is read, it is shifted into the checksum logic loop using XORs. By the time the received checksum byte is received, the calculated checksum is ready to be compared.

If a bad checksum is received, the controller issues another reset pulse and reads the serial number again until it gets it right or until the iButton is removed from the probe, which is indicated by the lack of a presence pulse in response to each bus reset.

PUTTING IT TOGETHER

Now that I can read the iButton serial numbers reliably, the rest is straightforward. To give some type of success or failure feedback, I use bicolor LEDs located by each probe. Green indicates that the number matches a stored number, and red means no match was found.

Configuration is a snap. By shorting the configuration jumper and powering up, the next seven iButtons read will be stored in NVRAM. You can store less if you like. The one downfall of this simple method is, if you want to add a key, you need to rescan each iButton you already have because the NVRAM is erased when you enter configuration mode.

During normal operation, when you touch an iButton to the probe, the LED will turn green if the serial num-

ber matches a stored number. Pin RB0 will also pulse high. This signal can be fed into a home automation system, or it can feed an external relay board to directly drive an electric door strike. The pulse can be configured for 1 s or 10 s. The 10-s pulse allows for a reasonable amount of unlock time to get the door open.

If you insert an iButton that isn't stored in memory, the LED turns red and the DigiLock waits for 10 s before it will read another iButton. This was done mainly to prevent a thief from cycling through iButton codes as fast as the DigiLock could read them. With the 10-s delay, it would take 89 million years to cycle through all the possible 48-bit serial numbers.

If a thief tried for a day, he'd have a 1 in 32,578,122,304 chance of guessing the right code. He'd be better off playing the lottery.

I wanted to know which iButton was used, so I used the unused output bits on the PIC and pulled each one high when the corresponding iButton is read. If the fourth iButton stored is read, the fourth bit is pulsed high along with RB0. By feeding these eight bits into my HCS, I can tell when each valid iButton is used.

I can add more intelligence to my electronic access system without bothering the HCS with reading the iButton serial numbers. For example, if my neighbor has the sixth iButton, I may not want that iButton to open the door at certain hours. So, if the sixth bit goes high, I can check the time and then decide if I want the door to open. To do this, I have to control the door strike with an HCS RBUF-Term Relay instead of using bit RB0 on the DigiLock.

Photo 2 shows the completed DigiLock board. It takes up little room and, thanks to the 1-Wire network, can be mounted anywhere. Simply daisy chain all your probes together using twisted-pair wire and connect them to the DigiLock. If you want to use LEDs by each probe, use another twisted-pair daisy chained to each 3-lead bicolor LED.

Each LED should have a 180-W resistor between the common lead and ground to let you to link each

Listing 2—It may look strange, but this code compiled extremely small. With only 7 μ s to sample the returned bit, there wasn't a lot of CPU time available to perform the necessary checks. Overall, the bit is sampled three times and stored in the buffer in 12 μ s with a 4-MHz crystal.

```

..... // We now have ~7  $\mu$ s left to sample a few times
..... if (!m1an) { bit_cnt++; }
BTFSS 05,0
INCF 1F,F
..... if (!m1an) { bit_cnt++; }
BTFSS 05,0
INCF 1F,F
..... if (!m1an) { bit_cnt++; }
BTFSS 05,0
INCF 1F,F
..... // Lets shoot for 2 out of 3!
..... // Use a quickie bit test for efficiency
..... shift_right(&byte_in, 1, !bit_test(bit_cnt, 1));
BTFSS 1F,1
GOTO 062
BCF 03,0
GOTO 063
BSF 03,0
RRF 1D,F

```

door probe to the DigiLock using two twisted pairs. For noise immunity, I recommend using CAT 5 cable.

SAFETY CHECK

An article about electric door locks wouldn't be complete without touching on safety. Door locks can be wired in fail-safe and fail-secure modes. Fail-

safe means that the lock will disengage and unlock the door if power is lost. Fail-secure means the door stays locked when power is lost. Before you think that fail-secure is always the right way to go—read on.

Electric door strikes are usually wired in fail-secure mode so that, if the power fails, you can still get out

Listing 3—Because of the nature of the iButton connection with the probe, checksums ensure data integrity. The family code (0x01) and the 48-bit serial number are used to calculate the checksum. If the calculated result matches the received checksum byte, the data is assumed valid.

```

// Gen CRC
// This routine takes the old CRC value and a new byte
// and returns a new CRC value
int gen_crc(int crc_in, int newdata) {
    int i;
    for(i = 0; i < 8; i++) {
        shift_right(&crc_in,1,(bit_test(crc_in, 0) ^
            shift_right(&newdata, 1, 0)));
        // Check for the extra XOR
        if (bit_test(crc_in, 7)) { // We need an extra XOR
            crc_in ^= 0x0C;
        }
    }
    return crc_in;
}
...
// This is the checksum section used during iButton reads.
// family contains the received family code. The serial number
// is read and stored in the id_in array while calculating the CRC
crc_in = gen_crc(0x00, family);
for(i = 0; i < 6; i++) {
    // id_in[i] = read_byte();
    // crc_in = gen_crc(crc_in, id_in[i]);
    // The code below is more compact (15us vs 20us)
    tmp = read_byte(); // Read next byte from iButton
    id_in[i] = tmp;
    crc_in = gen_crc(crc_in, tmp);
}
// Read CRC from iButton and compare to calculated value
if (crc_in == read_byte()) { // The CRC is okay!
    return TRUE;
} else {
    return FALSE;
}

```

of your house. However, there are also electric deadbolts out there. A common type of electric deadbolt is nothing more than a solenoid with a steel shaft. Make sure if you install these, you wire them in fail-safe mode. If you don't, you risk locking yourself in your house, which during an emergency is a bad thing.

I don't recommend using these types of deadbolts. Instead, look for higher-quality electric deadbolts that still have a manual lever inside. If you can disengage the lock manually from the inside, you can wire it in fail-secure mode without risking being trapped in your house.

Regardless of what you use, I highly recommend connecting your DigiLock and electric door locks to a power supply fed off a small UPS to ensure that your locks continue to function, even if the power goes out.

HINDSIGHT

I still can't explain the one glaring omission from the Digilock design. I was so wrapped up in making it small

and flexible enough to tie into the digital I/O ports of the HCS-II or any other HA system, I forgot to include an onboard relay to drive the electric door strikes. Needless to say, the next run of circuit boards will have a heavy-duty relay onboard so you can drive the door strikes without an external relay card.

The DigiLock is a simple-to-use and secure electronic lock. I enjoy just touching the iButton to the probe and pushing the door open. Of course, I just found some RF-based security cards, but they'll have to wait for another column. ☒

Mike Baptiste runs his own company, Creative Control Concepts, which designs and sells home automation equipment, including the HCS-II. You may reach him at baptiste@cc-concepts.com.

SOFTWARE

The software and schematic for the Digilock project is available via the *Circuit Cellar* web site.

REFERENCES

Dallas iButton datasheet, www.dalsemi.com/DocControl/PDFs/1990a.pdf
Dallas 1-Wire Network application note, www.dalsemi.com/DocControl/PDFs/app108.pdf
Dallas 1-Wire Checksum application note, www.dalsemi.com/DocControl/PDFs/app27.pdf
Microchip 16F84 datasheet, www.microchip.com/Download/Lit/PICmicro/16F8X/30430c.pdf

SOURCES

iButtons and probes

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

DigiLock Kits and Boards

Creative Control Concepts
(919) 304-3107
www.cc-concepts.com

Microchip PIC16F84

Microchip Technology, Inc.
(888) 628-6247 • (480) 786-7200
Fax: (480) 899-9210
www.microchip.com

Tom Bishop

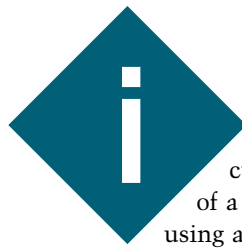
Rapid Gratification with FPGA Design

Quicker and Better Design

Part
2
of
2

Tom covered the
FPGA

background information in Part 1, now it's time to take a detailed look at the design process. It's all here, everything from designing functional blocks to simulation and debugging.



In Part 1, I discussed the design of a pulse multiplier using a Xilinx FPGA.

Now, I'll continue with a detailed look at the design process.

There are a few crucial steps for designing with modern tools like the Xilinx environment. Years ago, the designers of Sun Microsystem's Sparc-I computer said that the three most important things they had done in development were simulation, simulation, and simulation. I believe those are still wise words.

It's also critical that the designer clearly communicate the intent of the design. This can be in the form of annotated schematics, operational diagrams, or written theory-of-operation documents. One of the best and most overlooked places to communicate is in the design's source files.

VHDL allows the names of signals to be arbitrarily long, so the name of a signal can communicate its intent (see Figure 1). I worked with one engineer whose signal names read like short paragraphs. This was probably going overboard, but his intention was to communicate clearly. Well-written VHDL should be readable.

I tell my clients that they will be able to take my VHDL source, read it, and understand what the design is

doing—even if they are unfamiliar with VHDL. Clients like this, but I do it for myself more than for anyone else. I am the one who has to fully understand and debug the design.

Another way to communicate the design is by using comments. Comments are useful but are a distant second to writing high-quality communicative VHDL. It's human nature to read what the code is doing but skip over the comments.

Comments can even be harmful. I've spent a lot of time trying to understand someone else's designs and reconcile them with the comments. I later realized that the circuit was changed and updated, but the comments were not. If the reason for using a particular algorithm or method isn't clear, comments are appropriate, otherwise, the VHDL should stand on its own to tell what it is doing.

VHDL is a powerful language. It's easy to just write and let the synthesis tool convert it to hardware. Unfortunately, this can lead to circuits that operate slowly and have high gate counts. Always keep the hardware in mind when writing VHDL.

Before synthesis tools were commonly available, some designers would write VHDL models of the design blocks and simulate them to work out the bugs at a high level. The VHDL would then be hand synthesized, or manually converted, to gate-level schematics. I don't recommend doing this, but it is important to learn how the various VHDL statements will be implemented in hardware.

DECOMPOSE THE PROBLEM

Note that the heading isn't "Understand the Problem." Although an understanding is desirable, it will usually come with time.

Rather than working out all the internal details of the circuit, it is better to break the problem into manageable pieces. This will show the basic functional blocks of the design. Each block may have many internal details and requirements that are not visible at this level. This is good.

My personal experience is that a complex system (or chip) has far too many levels of detail to comprehend

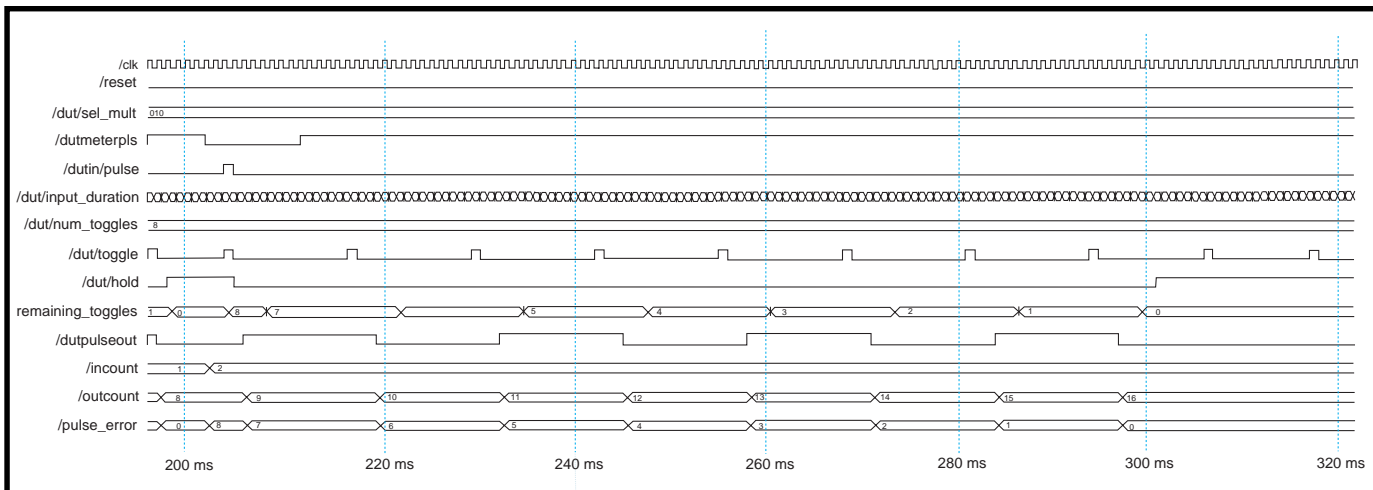


Figure 1—In this sample simulation of waveforms in the pulse multiplier testbench, the period since the previous pulse is recorded at the falling edge of /dut/meterpl. Signal /dutupseout is the multiplied output that cycles four times in that period. The output then holds until another input pulse is detected.

at once. Often when I finish a project, I will step back from it and see a tiny package that performs the specified functions. I understand the internal workings intimately, but it still seems like magic to have that much complexity in such a tiny package.

It is by dividing the problem that you can focus on only the details that are appropriate for that level. A term for this process is design abstraction. I think of a function in an abstract sense and ignore the details at the lower levels of design. If I try to keep all the details of a design in my mind, it becomes difficult to make progress.

For instance, you can probably convince yourself that a transistor can be fabricated on a piece of silicon, and understand how that transistor will work. (There are, of course, thousands of details at this level. I am assuming that you can ignore most of them.) Given that you accept that transistors will operate, you can connect a few transistors together and eventually understand how they will work together as a logic gate. Once you understand the concept of a gate, you can ignore the transistors and focus on the operation at the gate level.

Logic gates can be connected together to build small function blocks, such as single-bit half and full adders. You can work through the logic to confirm that a 1-bit adder will work as expected. You can then connect these small functional blocks and build arbitrarily large functions, such

as 32-bit adders. It is reasonable to understand a large adder in terms of many single-bit adders.

Using the 32-bit adder and other large function blocks, you can build a system. You can understand the operation of the system, by understanding how the large functional blocks will interact. But at a given level, you want to assume that the details of the lower levels will work without having to think about them. Ignore the details of all the lower levels as a convenience, so your mind can focus on those that are important at the current level.

There are still many decisions that will need to be made in the functional blocks. Decomposing the problem does not avoid this. It does defer these decisions until later, so you don't get bogged down in details early in the design cycle. It also breaks the problem into manageable pieces that can be implemented more easily.

FUNCTIONAL BLOCKS

The functional blocks are designed using schematics, state diagrams, or VHDL code. Verilog, ABEL, or other hardware description languages (HDLs) could also be used. For the purposes of this article, I will assume that VHDL is the HDL of choice.

Although, my methodology favors VHDL for most functional blocks, I sometimes use schematic sheets for the top level of design, because it is simple to create functional blocks,

which are particular to the FPGA that I am targeting. This might include on-chip memories, on-chip oscillators, input/output pins, or Global Set and Reset blocks. Alternately, I might use VHDL for the entire design. The frequency multiplier uses a single schematic sheet at the top level, with VHDL for the major functional blocks.

Many engineers who are not familiar with VHDL prefer to use schematic sheets for their designs. There are advantages and disadvantages to either method. VHDL can be intimidating and there is a learning curve to use the language effectively. Also, a schematic sheet can show a graphical flow of signals from module to module (see Figure 2).

At a high level, a schematic sheet can show the structure of a design better than a text-based model. In addition, a design can be most highly optimized using schematic sheets. A designer with an intimate knowledge of the FPGA architecture will know tricks that can do a better job than the current generation of synthesis tools.

On the other hand, as a designer becomes familiar with VHDL, they can design much quicker than with schematic sheets. VHDL is a powerful language. A few lines of VHDL can generate a complex circuit. A large design can be quickly simulated at the behavioral level. Well-written VHDL is more scalable than schematic designs. Number formats, bus widths, and the components that access these

buses can be quickly changed with VHDL.

In a number of ways, VHDL is generally more portable than schematic sheets. Because it is a text-based language, VHDL is portable between tool environments, where schematic sheets are not. Also, synthesis of VHDL can target different devices by simply changing the synthesis library. A final advantage of VHDL

is that you can express the intended function of circuitry in a clearer, more explanatory fashion than with schematic sheets.

Tools that allow you to draw a state diagram are also available. These then translate the state diagram into VHDL or Verilog that can be simulated and synthesized. The beauty is that a graphical picture of a state diagram communicates the desired operation of the circuit. I believe that any tools that help the designer and



Photo 1—An FPGA and serial PROM are the only active components in the completed design. The rest is DIP switches, status LEDs, decoupling caps, pullups, and connectors.

others understand the operation better, are definitely worth using.

That said, I mostly enter state machines directly into VHDL because I am used to that approach. Scary, but true—I have reached the point that I “think in HDLs” as I design. Either method is much simpler and more descriptive than designing a state machine with schematic sheets.

Fortunately, there is a way to integrate these design approaches. A state diagram will be converted to VHDL

for synthesis to gates. VHDL can represent components and their interconnections, so a schematic can be “coded” in VHDL. The computer tools will convert a schematic to a VHDL file quite easily.

TEST RUN

Even before the circuit is completely working, it’s a good idea to run the synthesis and implementation tools. This will check the

ability of the tools to synthesize the constructs you used. It will also give a starting indication about the size of the design. Both of these pieces of information are useful to avoid surprises late in the design cycle.

DESIGN SIMULATION

Simulation is the process of using the source models for a design to produce information about how the circuit will operate. Simulation can mean different things to different

people. When I simulate, I convert the schematic sheets to VHDL code and then compile the code and the other VHDL source files in the simulation program. Lastly, I generate inputs to the circuit, cause the circuit to operate, and check the output results.

There are at least two ways to simulate a design—simulation at the VHDL level, and simulation at the gate level. Many low-cost CPLD and FPGA tools include only a gate-level simulator. With these, a user must first synthesize and implement their design. The simulator then models the operation of the logic gates to determine how the circuit will work. This simulation will indicate whether a circuit works or not, but it is a less effective way to debug a circuit.

A better way is to simulate at the VHDL level, before synthesis. This enables you to debug at a higher level and use the source directly. You can watch the operation of the circuit as you step through the VHDL line by line. You can also see all the objects in the design, even if optimization

will later remove them. Because you are simulating at a high level, your simulations will also run faster. Speed probably isn't much of a concern because most designs are small enough that simulation speed isn't a problem.

Simulation is a critical design step. A good simulation gives you better information about how the circuit will run than about actually building a prototype of your design (see the "Why Simulate?" sidebar).

DEBUG THE DESIGN

This step is performed as part of the simulation task. As you drive logic signals into the inputs of the model, you can watch the circuit operate. If the operation does not meet our specifications at first, you need to troubleshoot the design.

Debugging a design in a computer simulation is conceptually similar to debugging a piece of hardware on a lab bench. I generally use some sort of a signal source (maybe a signal generator or pattern generator), or I might connect the circuit to the device I'm

trying to interface. If you are using a computer simulation, you can either generate inputs from the command line of the simulator or write a model to generate signals.

My preferred method is to write a VHDL test bench, which generates stimulus patterns for the logic design. In a test bench, you can use VHDL more like a programming language than a hardware design language. Much of the VHDL is not synthesizable. But I know the test bench will never need to be synthesized, so I can take advantage of whatever parts of the language suit me. For a simple test bench, you need only to generate input data. For a large design, you can check the circuit output values and confirm that they're what you expect.

The pulse multiplier is a fairly simple circuit, with that in mind, I used a simple stimulus generator in the test bench. I also have a simple output checker to verify the circuit operation. To see if the multiplication is working properly, I count input and output pulses. I multiply the input

pulses by the expected multiplier, and then subtract the output pulses.

If the difference is not zero at the end of an input cycle, I know I am not getting the expected multiplication. The power of the VHDL language is that it takes only a few lines of code to do this. I can observe this error count, along with all of the other signals, in a waveform window on the computer screen, much like a logic analyzer display.

To debug the circuit, I start by compiling the model of the test bench in the simulator. I then simulate the test bench, which calls in the model of my main design. I then add signals

from anywhere in the design hierarchy to the waveform display and tell the simulator to run. After a few seconds, the simulation results appear in the waveform display.

If the output is not what I expect to see, I use a divide-and-conquer approach to learn where I made my mistake. I first find a signal that is about midway through the design and then check to see if the signal has the expected values. If not, the first error is probably in earlier circuitry. Then I find a signal at a convenient place, about midway into the suspect block. I repeat this process until I isolate the block with the error.

IMPLEMENT THE DESIGN

Design implementation actually comprises a number of individual steps. Each of the steps is intensive and complicated. If you had to do these steps manually (which is possible using the software tools), the total design effort would be many times larger. Fortunately, the FPGA development tools are one flavor of electronic design automation (EDA) tools. "Automation" is a delightful word for any of us who have ever implemented a design manually.

The complex and often less-obvious steps described farther on are performed entirely by the develop-

Why Simulate?

For electronic design, the typical development strategy is to design a circuit, build a prototype, and then debug the prototype. You would debug these circuits by tracing expected values through various locations in your circuit.

When you build a circuit entirely inside a chip, you can't probe the intermediate stages inside a design to look for problems. With ASIC devices, you must verify that the design works before making prototypes. Simulation is a tool that allows you to model your circuitry and view its operation. Only then, do you build the circuit.

VHDL and other hardware description languages allow you to write a computer model that describes your hardware. You can then simulate those models, which means you apply stimulus to the inputs and observe the results through the circuit and at the outputs.

With an FPGA, if your logic has a bug, you don't have a large NRE expense for making changes. The only cost is the time to regenerate the design and run the tests again. So, you could say that simulation is not strictly required when designing with FPGAs. If a design goal is to make reliable circuits that operate consistently, simulation becomes a necessity. Simulation provides better visibility into the operation of a circuit than just building the circuit and putting it into operation.

For instance, designs will often have internal registers. The stored data in these can affect the operation of the circuit. What if you forget to initialize these registers? If you build a prototype device, the internal registers might wake up in a state that allows reasonable operation of the circuit. But when the device begins production, the registers from a different batch of chips can take a different set of values at powerup. This can cause your circuit to fail, and in a manner that can be difficult to troubleshoot.

VHDL (and the IEEE 1164 standard) use more than the values 0, 1, and high impedance. All signals wake up with the value U, which shows that no value has been assigned. If you simulated the design using VHDL, any nodes you forgot to initialize would have the value U.

Any downstream gates, whose values depend on this signal, would also propagate the value U.

So, instead of the simulation showing the device working properly, at some point you would be able to see output signals changing to Us, instead of zeros or ones. You would immediately know there is a problem, and the problem was a result of uninitialized values in the circuit.

Another insidious error might occur if you put multiple drivers onto a single wire, which is fine if the gates have three state drivers, but perhaps your error is that you do not control the enable signal correctly and have data contention. A prototype of the circuit might work correctly, but when you take the device to production, different batches of chips might have different logic thresholds or drive strengths, which could cause the circuit to fail. This type of production crisis is usually more expensive to fix than if you had solved the problem in development.

Had you simulated this design in VHDL, you would have quickly learned about the problem. VHDL resolves signal contention issues in the simulation. Whenever a resulting value is not clear, the signal gets the value X, or unknown. Gates will likewise propagate unknown inputs to their outputs. When signals with the value X are seen in a simulation, you should realize that a design error is the likely cause.

Critical race conditions are also more likely to be caught in a simulation. A design you plan to synthesize will not specify propagation delays. (The actual delays can only be computed later, after a design is synthesized, placed, and routed.) Simulation, by default, assumes zero propagation delay throughout the circuit. A race condition that requires propagation delay to operate properly is likely to fail in the simulator. The actual circuitry might work properly with certain production runs, but not others.

The immediate benefit of simulation is that it shows how the circuit operates before the circuit is built. The less obvious benefit is that simulation provides more information than would a physical prototype.

ment software. The computer tools are designed to make all of the implementation steps a fast pushbutton operation, and they are largely successful. For the pulse-multiplier project, my computer performs all of these steps in around 40 s.

Implementation is mostly a task for the FPGA development software. In the simplest case, the user tells the tool to make gates, or attempt an implementation pass. This process involves synthesizing the VHDL and combining it with the other schematic or VHDL modules. The statements in the VHDL are interpreted and the synthesis tool generates logic for the various modules. If the VHDL contains statements that are not synthesizable, the VHDL will need to be modified to use statements that the synthesis tool can interpret.

This sounds like a challenge. In fact, there is some learning curve for this step. There are several books that teach how to write VHDL for synthesis. The most accurate reference should be the "design for synthesis" document that comes with the FPGA design software. In my opinion, these documents contain information that is well worth reading. Besides showing what will work for synthesis, they should help you understand why some structures work better than others.

After the design passes the synthesis step, the tools will attempt to optimize the design. Think of a skilled designer analyzing a circuit to see where logic can be minimized, redundancies eliminated, or where gates can be combined into more efficient units. There can be tradeoffs here, of course.

What's the best design? Do you need something that uses the fewest gates possible, even if it runs slower? Or, do you need to get the highest speed from a block? The tools will probably choose to optimize for the highest speed. An option menu will let you to specify any tradeoffs.

The next step in implementation is mapping the gate-level design into the functional blocks specific to the FPGA family. After that, the tool runs a place-and-route pass. The locations of functional blocks on the FPGA silicon must be specified. The goal here is to locate the blocks in order to require only the fewest signal routing channels, and to keep the length of the routed wires the shortest.

The number of wires interconnecting logic blocks is limited. Any working placement cannot use more wires

What's Next?

I could have implemented the pulse-multiplier project using a microprocessor instead of an FPGA. The algorithms could be implemented in software. In fact, much of the FPGA design cycle is similar to a software development task. For the pulse multiplier project, it would probably be cheaper to use a low-cost microprocessor.

Why would you ever want to use an FPGA instead of a microprocessor? Speed is the main reason. A microprocessor would implement the algorithms as a sequential program, spreading the processing out over time. In an FPGA, I implemented the design using dedicated hardware for each operation. Every plus or minus sign causes an adder or subtractor to be built in hardware—the ultimate in parallel processing.

One analogy that helps show the differences involves the construction of machines. A lone mechanic can (and must) do all of the steps to produce the end product. Likewise, a microprocessor sequentially completes all of the steps that make up an algorithm. If you instead use an assembly line to build the machines, the tasks are distributed to many workers who can use specialized machines at each step. The assembly line uses more resources but can produce many more machines per day.

The FPGA does not process a sequence of instructions. It, like the assembly line, uses dedicated hardware for each operation in the algorithm. If the number of required operations is large, the FPGA must also become large. But, the speed increase over a microprocessor becomes proportionally higher as well.

The pulse-multiplier circuit is not speed critical. I instructed the synthesis tools to optimize for the fewest gates, not the highest speed. According to the timing analysis that the tools provide, I could in-

crease the clock rates by 20,000×, without changing any other logic. The circuit would then operate on much higher input frequencies.

Designs often have speed-critical areas that must use dedicated hardware, and non-speed-critical areas (such as a user interface) that can operate more slowly. Typically, a design might use an FPGA for the speed-critical areas and a microprocessor for the user interface. Why not put the user interface into the FPGA as well?

In the past, there have been good reasons not to do this. Cost is one. I am assuming that the circuit will use an FPGA anyway, and there are probably enough gates to do a simple user interface. If you need a larger FPGA, the incremental cost will probably be low, plus you eliminate the need for the micro and its support logic. This arrangement allows a smaller board outline with fewer components.

The idea of using dedicated hardware to do user-interface functions seems cumbersome at first, but this can be written using VHDL, or even directly using a state-machine editor. This is similar to coding the user interface in a high-level software language. Pushbutton synthesis and optimization make the actual implementation trivial. Complex state machines use relatively little logic.

Memory in FPGAs has been limited in the past. For instance, it would use a lot of resources to display a screen of text. With older FPGAs, this would make for costly memory. But now there are many FPGA architectures that provide dedicated memory blocks, which can be configured as RAM or ROM. These are separate from the FPGA logic blocks. If these are not all used by the high-speed logic, then they are free to hold any other information you wish. So, even a text-intensive user interface may be practical with these devices.

than are available at any location on the die. Also, the length of the wires on most FPGAs can incur more propagation delay than the delay through the gates. Properly locating the functional blocks to shorten the wires will make the design run faster.

Lastly, the tools must take the block placement and routing information and put it in a format that can be loaded into the device. The signal routing, logic configuration, and other parameters are programmed into the FPGA by changing the values of internal memory cells. The configuration is placed in a format that can be downloaded as a serial bitstream.

DOWNLOAD THE DESIGN

Now that you have a bitstream file that contains the program for the FPGA, you need to load that information into the FPGA. For production, this information would be loaded into the FPGA at powerup. The simplest way to hold the configuration information is in an 8-pin serial PROM. But for development, you can download the information directly from your computer into the FPGA.

A download adapter is used to send the configuration through a serial or parallel port on your computer to the target device. The download adapter is provided with the implementation tools. Again, a mouse click starts the download operation, which takes between 1 and 4 s on my computer, depending on the size of the FPGA. When the download is completed, the reset pin is briefly activated, and the design comes to life.

IN-CIRCUIT TEST

You can now test your design using real hardware. The demonstration board makes this easy. You've built your hardware—it is inside the FPGA. The pulse multiplier uses just one input and one output, plus DIP switch inputs to select a multiplier value. Because the circuit uses an oscillator internal to the FPGA, the circuit doesn't need a clock input.

I chose a pin that is connected to a pushbutton for the input. I also added an output pin that drives an LED. The circuit buffers the input pin, and

drives the LED output. I chose the output pin as one that was connected to the adjacent LED. When I pressed the pushbutton, I could see the input LED change. That gave me confidence that the FPGA was being configured correctly. As soon as I pressed the button, I could see pulses appearing at the output.

I could then test the design by pressing the pushbutton at a regular rate and see the output synchronize to my button presses. This was quite gratifying. I did not have much visibility into the performance, however, other than counting output pulses.

For the second phase of testing, I connected a signal generator to the input, using a mini-clip to attach it to the header connector. The signal generator had no problem overdriving the pull-up resistor on the board. For easy viewing, I connected an oscilloscope to the input and output.

The pulse-multiplier circuit was designed to work with slow input pulses. I chose internal counter sizes and bus widths to meet this requirement. The slow pulse operation worked fine, but the slow sweep speed was difficult to view on my non-storage oscilloscope. I easily fixed this development by driving the clock from a higher frequency internal source. With this change, I could increase the pulse rate and see nice stable waveforms on the 'scope.

From start to finish, I made a handful of changes to the pulse multiplier and fixed several errors. I initially designed the circuit to use the simple counter method as the multiplier, but I changed to the DDA algorithm in my second revision.

I also changed the output control circuitry. I wanted to avoid a potential problem with short output pulse widths. I found that these would occur when the output toggled, and then a new input pulse would synchronize the input again a short time later. The output-control block maintains a minimum high and low pulse width.

A third change was to allow the DIP switch inputs to select the multiplier value. Until then, I had programmed the design to use a constant multiplier. The changeable multiplier

value made it easier to observe the multiplier at work with the LEDs.

There were a number of errors that I fixed during development. Almost all of these were fixed during simulation, without using the actual hardware. I made a mistake with the handshaking to the output-control circuit, which blocked all output pulses from the multiplier. No problem. A simple logic error and a quick fix.

Another problem was the input counter. With low input pulse rates, the counter value would increase until it wrapped at zero, then continue increasing. This wraparound would give the appearance of a higher frequency input. Again, the correction was a simple change to the VHDL to prevent the wraparound.

A third problem was an off-by-one error in the circuit that counted the output pulses. For each input pulse, I would produce an output stream one half-cycle shorter than I expected. The total number of output pulses is critical for my application, so I fixed this error as well.

I did have one error that I did not find in my simulation. It showed up only in the hardware. These are the types of bugs that cause designers to mistrust the tools and pull their hair during debug. As with most areas of design, there are many ways to find these problems.

One way is to use the download port from the host computer. This device is also a hardware-debugging device that can be used much like a microprocessor emulator pod. Instead of using an external or free-running clock, the user controls the clock pulse generation from the host computer. At any time, you can upload the entire contents of the design and investigate any signals in question. I chose not to use this method.

If you are synthesizing VHDL, it's important to remember that the VHDL will be used to generate the

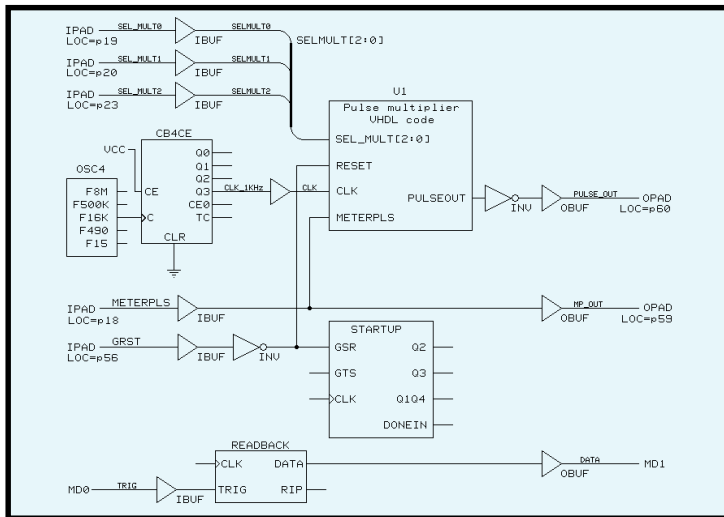


Figure 2—This schematic sheet was used at the top level of the design. This is the simplest way to include the Startup block and other FPGA primitives. The pulse multiplier VHDL block contains the majority of the circuitry.

actual hardware. Therefore, it eventually becomes important to understand how the various VHDL constructs translate to the circuitry that the synthesis tool builds. A good way to learn is to write simple models, synthesize them, and then generate a schematic sheet. This is possible with the Xilinx Foundation tools using the synthesizer directly. By the way, the Foundation tools are also useful to learn what hardware will be produced from the various VHDL constructs.

To debug the last error, I used a two-step process. First, I exported the synthesized gate-level net list in a VHDL format, and then ran a VHDL simulation on it. Normally, I would expect the simulation of the source VHDL to give the same results as the synthesized VHDL. This was not the case because internal signals had uninitialized values, and there were no expected outputs.

Secondly, I brought the VHDL model of the pulse-multiplier core into the FPGA Express synthesizer. I synthesized the design and then viewed the result as a schematic sheet. Machine-generated schematics aren't pretty—a human can do a much better job. Still, the logic is there, and I was able to look for the constructs that I expected to see.

The error turned out to be my problem. I found a group of flip-flops that had no reset input. I had forgotten to initialize a counter in the reset

state, which allowed it to wake up with unknown values. Why didn't the VHDL simulation catch this? Uninitialized values are supposed to be obvious (see the sidebar "What's Next?").

The reason became clear when I read through the source file. I had built a counter that internally used the type Integer. Integer types wake up with their minimum value, which was zero for the range of values that I had specified. So in the VHDL simulation, the counter always started

with the value zero.

After synthesis, however, I no longer have integer types in the design. The net list has only logic gates and their interconnections. The simulation of the net list showed the uninitialized values. In this case, the net list simulation showed more accurate real-world performance than the source VHDL.

The use of integers in a synthesizable model is a convenience. It allows the design to capture the system requirements in a readable manner. In this case, however, it masked another error—my unspecified initial behavior of the counter. A better way to model the counter would be to use the Unsigned type. Signed and Unsigned types form a bit vector that is interpreted as a two's complement value, or unsigned value, respectively. I use signed and unsigned values elsewhere in the model for the arithmetic in the DDA module and in other areas.

IN OPERATION

The last step in this project was to put the pulse multiplier into operation at the water plant. I assembled the circuit on a small piece of perf board (see Photo 1). This board was placed in a small aluminum enclosure and installed in the panel containing the controller. The controller provided access to 5-V power at a terminal strip, so power was brought down with the signal wires.

A serial PROM was included on the circuit board to hold the FPGA configuration. The data file was programmed, and the chips plugged into their sockets. The multiplier value was selected from the DIP switch, and the controller input scaling factor was modified to match. I used 128 as the multiplier value to provide a reasonable number of pulses per half-second sample window. It would also allow the input rate to increase to 3 Hz, while keeping the output rate below the maximum 400-Hz rate.

The installation and testing were uneventful. I tested the controller by first measuring the flow using the indicator on the turbine meter and a stopwatch. I then compared this value to what was indicated on the flow meter. They were within one gallon per minute of each other. Because the turbine meter is the source of the input pulses, the values of the two meters should be close.

I then needed to confirm that the circuit was never producing more or less total pulses than it should. I took

note of the total capacity indication on both the turbine meter and the controller. Nine days later, I rechecked the values again. Both indicated that just over 200,000 gallons had passed through the filter. In regular operation, the controller I designed worked as expected and, so far, has shown no surprises.

Was the project successful? Yes. Did I complete the project in one day? Well...not exactly. Projects seem to expand to fill all the time that is allotted for them, and a little more. This was no exception, and I had no real deadline for completion. Plus, I was having fun.

Over the course of the project, I generated 13 revisions to the design. Most of these were to change how I manipulated the displays, or because I used different internal clock rates for testing. Also, I took a few sidetracks that I knew were unnecessary. Each time I learned a little more.

This incremental learning process helps to put the FPGA design within the reach of most designers. The

FPGA tools allow a design to be implemented with little user intervention. Still, with a complex design, each user will want to optimize the process for either speed, gate counts, or debugging, depending on the application. The tools not only allow this tuning, but also allow a novice user to get started quickly. ▣

Tom Bishop has been a design engineer for 18 years, specializing in FPGA and ASIC design and verification using VHDL and Verilog. As a consultant for nine years, his larger projects have involved computer graphics, video, telecom systems, processors, PCI and FireWire buses, and customer training. You may reach Tom at tbishop@asiccess.com.

SOURCE

FPGAs

Xilinx, Inc.

(408) 559-7778

Fax: (408) 559-7114

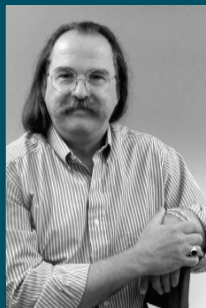
www.xilinx.com

FROM THE BENCH

Jeff Bachiochi

In Theory and in Practice

Part 2: Clock Adjustment and Digital Filters



Those of you who've been around

awhile will recall the HAL project. This month, with the assistance of a digital filter, Jeff converts the concept into a standalone unit.



Last month, I left you without a reason for needing a digital filter. I intended the article to be a standalone tutorial on filter design, but I introduced a specific design for a low-frequency narrow-bandpass switched-capacitor digital filter.

One of the most response-generating projects originally presented in Steve's old *BYTE* column was HAL, the hemispheric activation level detector. People still e-mail us about that project. So much so, that I revisited HAL, or at least the analog filter front-end optimization, in a past *FTB* column (*Circuit Cellar* 27). This month, I'll pull together the digital filter scheme from Part 1 along with some other circuitry to modernize the HAL concept into a simple standalone unit.

LOGIC VS. EMOTION

Your brain contains two distinct hemispheres. Each half may look like a mirror image of the other, but each half is organized for separate tasks—the logical and the emotional.

Logical activities include speech, reading, writing, spelling, and recalling names and addresses. Processing sequential and numerical information, as well as the literal interpretation and rational evaluation of facts are natural logical functions.

Emotional activities involve feelings and imaginative visuals. Exploring the relationship between events past, present, and future is best handled by the emotional side. We tend to process these activities in parallel.

The stimulation of these two areas often results in conflicting output. Your conscience may be the arbitrator of the final outcome, but I'm not here to debate the issues of the conscience.

SHOCKING INFORMATION

The brain produces tiny electrical signals indicating activity. Making sense of these electrical signals is best left to the professionals, but these signals have been categorized by certain ranges. Activity output in the 0.5–4-Hz range is known as Delta waves. Delta waves seem to indicate the deepest stages of sleep.

The range of 4–8 Hz is the Theta band. Theta waves include active mental imagery and creativity. These may be associated with sleep as well as with enhanced learning ability.

Electrical activity in the 8–12-Hz range is known as Alpha wave. Alpha waves suggest a state of rest (i.e., the lack of any problem solving activity).

Beta activity is in the range of 14–25 Hz. Beta waves correlate to the brain's engagement of thinking or acting on a stimulus.

Although individual decisions can be processed in a snap, a change of focus (or state) may take much longer. This is why we can't see any meaningful changes in less than 0.5 s.

TINY VOICES?

Not only must we contend with electrical activity resulting from bodily functions such as muscle activity, but also interference from outside sources (including electrical appliances and the dreaded 60-Hz hum).

To remove these unwanted signals, you need a good filter. The filter must be capable of reducing these signals by at least 50 dB (or 1×10^{-5}). That's, one reason for the switched-capacitor filter design in Figure 1. See what I've added to last month's filter design?

The circuit starts with an instrumentation amplifier. The in-amp allows differential inputs to amplify

microvolt-level signals while continuing to reject common-mode signals.

My in-amp has a common-mode rejection (CMR) rating of +70 dB, yet costs about \$4. A single resistor sets the gain of the in-amp. I used a maximum gain of 1000 for the front end. A 5- μ V signal across the differential inputs results in a 5-mV in-amp output.

The maximum output swing of the in-amp using a single 5-V supply is within 75 mV of the rails. Using a 2.5-V bias, this means that a \pm 2-V swing would be no problem. However, we must make room for an additional 12.5 \times gain boost further on down the line. So, a differential input needs to reach 200 μ V to approach a full-scale output level into the A/D (200 μ V \times 1000 \times 12.5). The total theoretical gain of the system is 12,500.

I placed the digital filter between the two gain stages. The filter has no gain associated with it. To prevent any post-gain amplifier from amplifying any generated noise, you would normally complete all the gain before any attempt in filtering. In this case, I didn't want unwanted signals to peg the gain stage such that any filtering would be useless, so I placed some of the gain after the filter.

The filter design was based on a bandwidth of about half the center frequency. With these criteria, I could center the filter in a number of different locations to cover the Delta, Theta, Alpha, and Beta bands.

Here's a second reason for using this switched-capacitor digital filter. Without changing any filter component values, I can move the bandpass filter around simply by changing the clock input. More on this later.

Following the bandpass filter is the final gain stage. I used a National LMC7111 single op-amp (about \$3). This CMOS op-amp is a single-supply

device that can reach within 20 mV of its power-supply rails. The potential output swing is important because it can limit the following A/D's range.

The output of the final gain stage is presented to the A/D converter. The PIC16C73 has a built-in 8-bit A/D. The gain stage's output is an AC waveform biased at 2.5 V (or $1/2 V_{CC}$). Because the input will vary above and below 2.5 V, we lose half of the A/D's resolution. We'll pay attention to conversions from 128-255.

A note here about Microchip's new PIC16F873. This flash-memory based replacement for the 'C73 has improved performance. The present 8-bit A/D is replaced by a 10-bit A/D. But, even more important is the addition of a VREF-. Presently, the A/D can be referenced at V_{CC} or a VREF+ input and ground. In an application like this where the input is biased at 2.5 V, you lose half the resolution. If the VREF-, which is always ground on the 'C73, could be 2.5 V (as with the 'F873's new VREF- input), there would be no loss.

If the LSB resolution of the 8-bit A/D is \sim 20 mV (5 V/256), then the minimum differential input to vary the A/D conversion by 1 bit will be

\sim 2 μ V (20 mv/12500). The input will be an AC signal so how do you know when to make an A/D conversion?

You need to make a number of conversions and look for the highest conversion value of each cycle. How can you stay in-sync with the input?

USER-SELECTED CLOCKING

The LTC1068 requires a clock of 100 \times the center frequency of the filter. This means that if the filter is to be centered at 10 Hz, you must provide a clock of 1 kHz.

I planned to use the PWM function as a background task. Unfortunately, even with a divide-by-16 prescaler, the minimum frequency was greater than the 100 Hz I needed for a 1-Hz center-frequency bandpass filter. So, I was destined to use a 16-bit timer. Now, instead of a single look-up table, I'd need one for the least significant byte and one for the most significant byte of the timer reload value.

To allow you to select the center frequency, six digital inputs are used. The first five set the frequency 1-32 Hz (0 = 32 Hz). The last input enables manual control.

This project is manual control, but

I wanted the hooks to have RS-232 control. Maybe I'll come back to this one day. It would require some kind of isolation between this unit and the serial device. Note that you should *never* use this project with any power source besides batteries. No AC/DC converters.

Three things happen when timer1 overflows. First, the timer is reloaded from the T1VAL_H and T1VAL-L timer1 reload registers. These are the values read from the two 32-entry tables. The offset into the table is based on the first five digital inputs.

Once reloaded, the timer is started again. While still in the interrupt routine, the FCLK

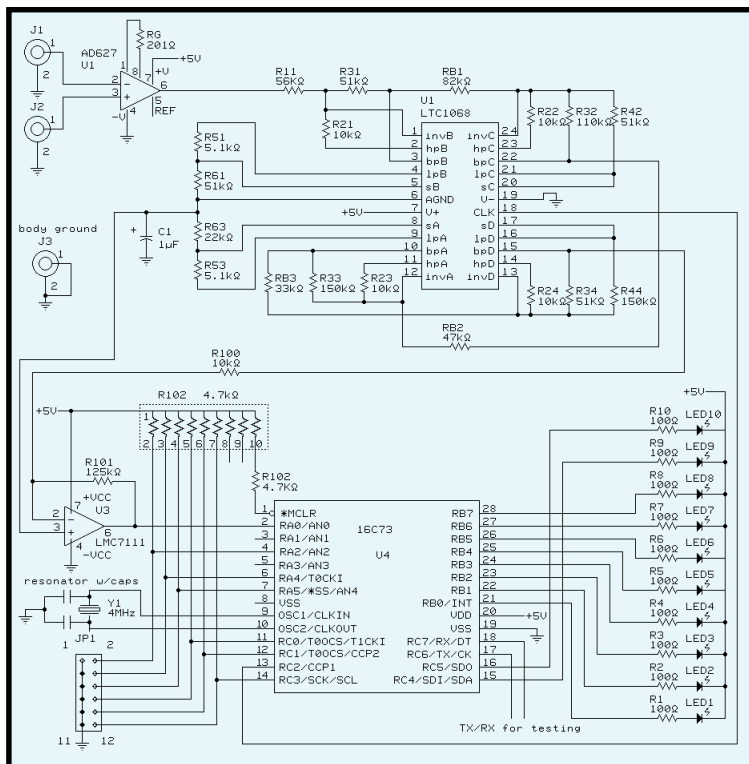


Figure 1—The top half of this schematic should look familiar if you read Part 1. The new switched-capacitor filter design gives you more to look at and helps eliminate unwanted signals.

output is complemented. This is the clock output that controls the center frequency of the digital filter. Finally, before leaving the routine, an A/D conversion is performed. The conversion value is compared to the TEMP_A2D register (highest sample value) and the TEMP_A2D is updated if the conversion value is higher.

Every 200 times through the interrupt, the TEMP_A2D value is transferred to the working register A2D, and then cleared to compile a new high value. Two hundred is a magic number here. The FCLK output bit flips 200 times for each 100 cycles.

One hundred cycles is the clock rate equal to one cycle of center-frequency input through the filter. No matter what the center-frequency, the A/D will always sample 200 times for each cycle. This way it stays in sync with the pass-band signal.

While not in the timer1 interrupt routine, I process two pieces of data in a continuous main loop. The first operation is sampling the digital inputs for a change in user configuration (i.e., a change of center-frequency) and updating the Mode register accordingly.

Second, I want to display the A2D register value. I didn't do any averaging of the A2D register because good data actually changes slowly and the A2D register only changes once a cycle.

To report the amount of signal measured by the A/D, I use multiple compares on the A2D register and jump to alternate routines to turn on a single LED. The ten LEDs are programmed to function as a linear bar-type display of the A2D register's value.

I could have used a logarithmic function by simply changing the values used for comparison. By preventing only one LED to be on at a time, precious battery current is preserved.

BODY CONTACT

To connect this project to the body, you need three leads. The first is a body ground, which places the body and the circuit ground at the same potential. The remaining two leads are the differential inputs to the in-amp. These leads must be shielded (with the shields tied to circuit ground). You can use a 6' (or longer) phono cable.

Because it already has a phono jack at each end, just cut it in half. Strip back the remaining ends 1" and remove the shield. Next, strip off 1/4" of the center conductors and solder them to female halves of #4 sew-on snaps (from a fabric shop). This snap receptor is just the right size for disposable pregelled electrodes.

You can make pseudo electrodes by soldering the conductor directly to silver dimes. You'll need a way to attach these, because there's no adhesive (like there is on the disposable electrodes). You can permanently mount these to a sweatband if the placement issue can be overcome. To assure proper electrical contact with the skin, a conductive gel can be applied to the electrodes.

The most common placement for the differential electrodes is the forehead and lower rear hairline. The pair can be centered over the left or right hemisphere. Pair placement, which bridges the hemispheres, will indicate differences between the two. The body ground lead should be attached some distance from the pair. You might use an antistatic wrist strap for this connection (or a third electrode).

This project is an engineering example of the design techniques used in acquiring brainwave signals. *This project is not medically approved*, nor do I make any medical claims for it. Power the circuit with batteries only. This circuit requires <50 mA and will run on four alkaline batteries without the need of a regulator.

HAL COMPARISON

HAL is a multichannel device that uses a wide bandpass filter and samples the amplified input once every 1/64 s. A PC is required for HAL. The PC receives the sampled datastream and, because the data rate is known, performs an FFT on the real-time data to extract frequency and signal strength from the two channels.

This narrowband activity project (NAP) is a single-channel device. It uses a narrowband filter and samples the amplified filtered input at a rate of 200 samples per cycle. Unlike the HAL, a PC isn't required. A 10-segment LED display indicates the rela-

tive strength of the input that falls within the chosen band.

OTHER POSSIBILITIES

I can think of lots of ways this project can be altered. A sound output might be of benefit to some. You would probably want the pitch or volume to change based on the A2D register value. The PWM function on this processor might work well for this.

As mentioned previously, if an isolated RS-232 interface was added, a PC could send instructions to move the bandpass filter. Application software might search for the strongest signals and report on what it sees.

This is a vertical niche project that certainly won't appeal to all. But you might find a small part that can help you in your future endeavors. That's what these columns are about. ☒

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.

SOURCES

AD627 in-amp

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

PTC1068 quad filter block

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

LMC7111 op-amp

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

PIC16C73

Microchip Technology, Inc.
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

A-10005 disposable electrode

Vermont Medical, Inc.
(802) 463-9976
Fax: (802) 463-9228
www.vermed.com

SILICON UPDATE

Tom Cantrell

SoC it to Me



System-on-Chip technology works

well if you've got the budget and the manpower to forge the chip. But, Tom has found an alternative that makes SoC technology available to the masses.



As we enter the year 2000, it's quite fitting that the silicon wizards are conjuring up a new way of thinking known as System-on-Chip (SoC).

SoC brings to mind bleeding-edge VLSI design techniques that give IC designers half a chance of getting their arms around million-gate megachips. It's a matter of necessity because the good old schematic just can't keep up with the march of silicon.

The idea is certainly appealing. Simply mix-and-match made and/or bought intellectual property (design know-how), punch a button, and voilà!—instant net list. Well, it isn't quite that easy, but at least the tools—hardware description languages (HDLs), IP catalogs, simulators, and so on—continue to improve.

SoC is dandy, but only for companies that can afford six-digit design software, huge minimum ASIC order quantities, and the army of engineers needed to forge a huge WunderChip.

But now, an interesting new breed of chips is emerging that brings the benefits of SoC to the mere-mortal designs and designers that make up the majority of the embedded business. Just like the big boys, you now have the ability to build a custom chip, but you don't need to pony up for the 100,000 parts to do it.

Consider the standard MCUs we all know and love—PICs, '51s, '68s, and so on. The price is right, but other than changing the software, there's no way to customize the chip to optimally match your application.

Then there are the field programmable gate arrays (FPGAs), which are a blank canvas of silicon just waiting for your design inspirations. However, they're rather pricey and don't achieve the performance of a dedicated chip.

What to do? The answer according to Silicon Valley startup, Triscend, is simple. Why not just put a hardwired MCU and a block of programmable logic on the same IC?

As shown in Figure 1, that's exactly what Triscend has done with their 'E5x configurable System-on-Chip (CSoC), which combines an enhanced turbo version of the venerable 8032/52 with configurable system logic (CSL). The 'E5x lineup utilizes the MCU in common while varying the amount of SRAM (for code and data), the number of configurable logic cells, and the pin count (see Figure 2).

The entry-level TE505 goes for under \$10 in volume, fulfilling the promise of an ASIC for the masses. Total up the cost of an MCU, FPGA, and RAM chip, and you'll see that the 'E5 is comparable. You've got one chip versus three, and the CSoC can do things a multichip lashup can't.

TURBO TIME

Given that it's only a part of the equation, Triscend may have been tempted to cut corners and save some silicon for the fancy stuff. However, the 'E5 turbo MCU is no slouch.

Of course, as the turbo moniker implies, it's faster than a traditional 8032/52. Instead of the meager 1 MIPS or so throughput of the older chips, the 'E5 significantly boosts performance by combining a faster clock rate (up to 40 MHz) and reducing the clocks-per-instruction byte to four.

The legacy peripherals, which include three 16-bit timer/counters, a full-duplex UART, and a watchdog timer, also have new features. For instance, the timer/counters can run off the system clock either divided by 12 for compatibility with the 8032/52

or divided by 4 for higher resolution, faster transfer rates, and so on.

The UART adds an automatic address-recognition feature to the traditional ninth data-bit mode for low-overhead networking. An extra data-pointer register eases the data-transfer bottleneck of the original design. There are modern conveniences like wait-states, power-on reset, a built-in oscillator, and even an NMI-like high-priority interrupt (HPI).

All this is a big improvement over the original, but to be fair, neither the turbo speedup and feature tweaks are especially new concepts. At this point, other players in the 8032/52 market, such as Philips and Dallas, already offer similar improvements.

But, the 'E5 does break new ground with the addition of a two-channel DMA controller and an expanded address space to go with it. Any DMA controller, and certainly one with a lot of features, is a rather big-ticket item to find on a lowly 8-bit MCU.

When you talk DMA, speed comes to mind. The Triscend unit doesn't disappoint. It can dish up single-cycle transfers if you can digest the resulting 40 MBps. If necessary, the DMAC also allows wait states to be inserted and even keeps track of up to 64 KB of pending requests if service falls behind.

Beyond the basics, the DMAC has plenty of extras with features like auto-initialization for repetitive transfers, four-byte receive FIFO, and a built-in CRC (CCITT-16) generator.

There's still a 64-KB logical address space, each for code and data—a must for 8032/52 software compatibility. However, built-in address mapping extends the physical address space to a whopping 4 GB with up to 32 address lines. Note that the unneeded high-order address lines can be used as general purpose I/O.

Speaking of I/O, with 128- and 208-pin PQFP and 436-pin BGA package options, there's certainly no shortage. Unlike the original 8032/52, the 'E5 provides a dedicated memory bus (not shared with port pins)

and furthermore, I/O related signals (UART, timers, interrupts, etc.) can be assigned to any I/O pin. Although the 'E5 runs at 3.3 V, the I/O lines are 5-V tolerant and, offer programmable output drive current (4 or 12 mA), optional input hysteresis (± 150 mV), and pull-up, pull-down, or bus-follower modes.

ALL ABOARD

The 'E5 MCU-core would make a decent chip in its own right. But, that's only the start. Let's look at the Configurable System Logic (CSL) that puts the "C" in CSoC and sets the 'E5 apart from other 8032/52 chips.

Although the documentation isn't explicit, CSL logic cells appear to be similar to the SRAM-based look-up table design found on many FPGAs.

A cell can perform any combinatorial function of up to four inputs, and it's quick at that (<5 ns). Cells are grouped together to widen the fan-in. For instance, two cells can handle any function for five inputs and a subset of functions for six to nine inputs.

For arithmetic tasks, each cell acts as a 1-bit slice of an ALU with ADD, SUB, ADD/SUB, or Multiply capability. These provide the building blocks for higher level functions of arbitrary width. For example, 16 cells can implement a 16-bit counter or a comparator with 25-ns performance.

Finally, the LUT can be hijacked for use as a 16×1 RAM or ROM (RAM without a write line). Cells can be paired to implement a 16×1 dual-port RAM, and a single cell can also act as an 8-bit shift register (i.e., FIFO).

WIRED

As vital as the MCU and CSL are, it's equally important to consider how the two are connected.

The simplest approach is to hang the CSL on the MCU like any other peripheral, perhaps throwing in a chip-select or interrupt. After all, this mimics the approach used when connecting separate MCU and FPGA chips. However, Triscend champions a much tighter and more intimate coupling that truly takes advantage of putting everything on one chip.

To that end, their configurable system interconnect is more powerful than the usual 8-bit expansion bus. Besides the full 32 address lines and separate read and write ports, a lot of consideration is given to getting the MCU and FPGA partnership working.

There's not just a chip-select line, but dozens or hundreds (more on the larger devices) of selectors that just about eliminate the need to use CSL silicon for decoding bus transactions and addresses.

Furthermore, selectors can link

CSL functions with specific MCU hardware like the memory bus. That means, logic can directly access any chip or resource the MCU can. Similarly, a selector can steer request and acknowledge lines between logic and the DMA controller.

To make sure the MCU and CSL march to the same drummer, the MCU clock is broadcast into the CSL. Meanwhile, the CSL has six global-buffered, low-skew lines available for other clocks or high-fanout signals. Clock sharing even extends to power management. When the MCU enters powerdown, each clock takes appropriate pre-programmed action either by

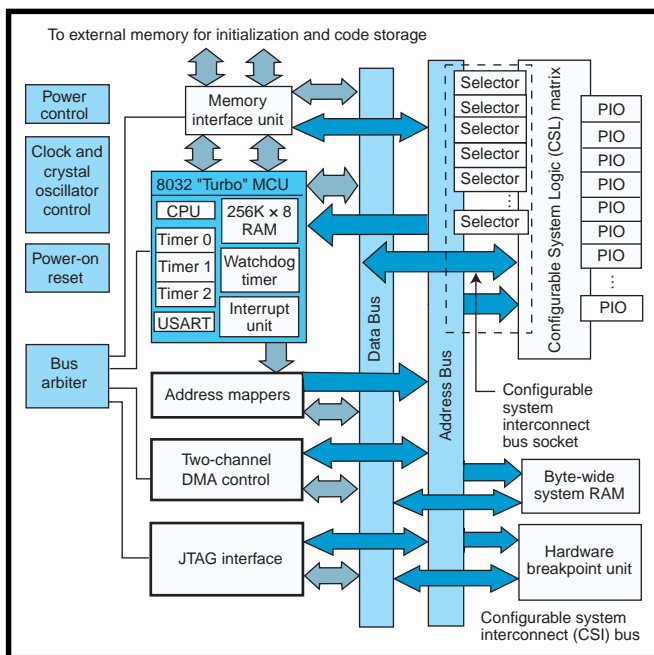


Figure 1—The Triscend 'E5 combines an MCU and CSL in order to craft a field programmable System-on-Chip.

continuing to run, holding the last state, or being forced low.

The 'E5 even allows mapping unused MCU SFR (Special Function Register) addresses to CSL logic. Now you can use the quick and easy direct- and bit-addressing modes that the 8032/52 reserves for SFRs, with your own logic. Another great example of the 'E5 being able to do things a separate MCU and FPGA can't.

The tell-all relationship between MCU and CSL extends to debugging as well. The CPU informs CSL logic when it encounters a breakpoint, thus, CSL logic can force a breakpoint.

WHERE AM I?

With the entire personality (code, data, and logic) of the 'E5 living in RAM, you have to initialize the chip after power-on. Fortunately, there are a variety of startup options that make the job as easy as it can be.

Three pins control the behavior of the chip after powerup. SLAVE* determines whether the 'E5 itself initiates and controls downloading or whether

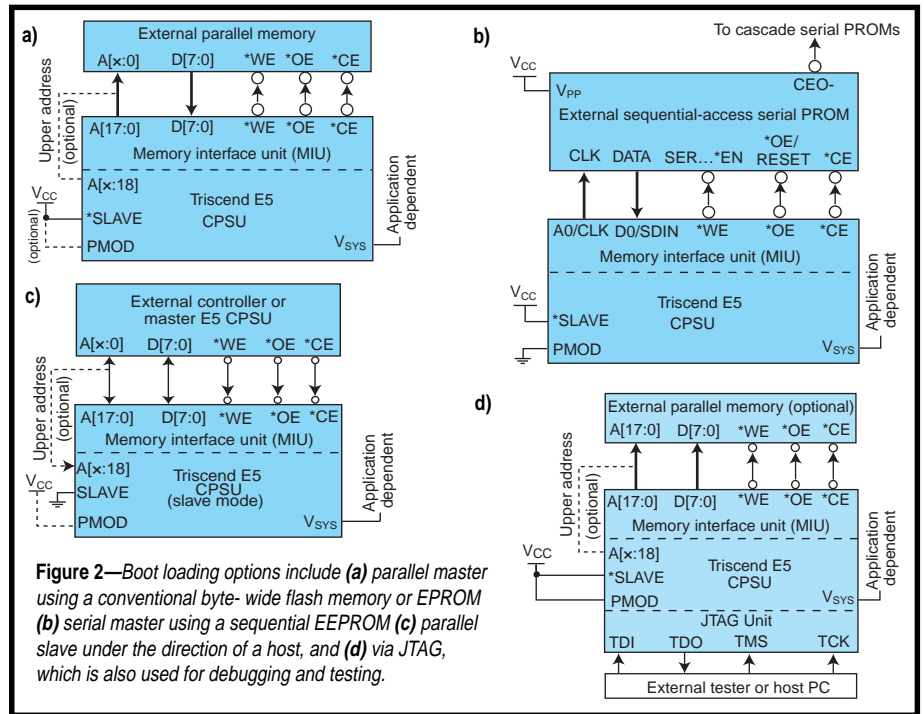


Figure 2—Boot loading options include (a) parallel master using a conventional byte-wide flash memory or EPROM (b) serial master using a sequential EEPROM (c) parallel slave under the direction of a host, and (d) via JTAG, which is also used for debugging and testing.

it waits for external logic to take charge. PMOD selects between parallel- or serial-bus modes. Finally, V_{sys} specifies what action the chip will take should initialization fail. If V_{sys}

is high, the 'E5 goes into power-down mode waiting for another power-up or RESET. If V_{sys} is low, it will automatically retry the initialization sequence indefinitely.

In parallel-master mode (see Figure 3a), the 'E5 reads the byte-wide memory (usually EPROM or flash memory) to initialize itself and begins execution (at address 0000H) in the same memory. The external memory needs a program to map the code and data to the internal or external memory.

Serial-master mode (see Figure 3b) utilizes the same sequential access serial EEPROMs popular with the FPGA crowd. As the name implies, these chips don't need an address. They start at address 0 after RESET and proceed sequentially bit-by-bit with each CLK.

The parallel-slave mode wiring (see Figure 3c) is just like parallel-master mode, except the direction is reversed. For instance, addresses and control lines are inputs to the 'E5 from an external controller. To the host, the 'E5 looks like RAM. The 'E5 waits in RESET until configured and released by the host. External code fetches aren't allowed in this mode, so the entire program must fit in 'E5 internal RAM.

Besides testing and debugging, the 'E5 JTAG port is put to use, providing the equivalent of a serial-slave initialization mode (see Figure 3d) with a number of unique capabilities. First, the JTAG port is treated as a bus master by the 'E5, giving the host complete visibility into the target system. In JTAG mode, the external memory is optional and can be of any technology (e.g., DRAM) because it doesn't play a role in the boot process. In-system programming is easy because the host can blow an external flash memory or EEPROM via JTAG and the 'E5 memory bus.

A final option is the so-called stealth mode. Once the program is loaded into internal RAM, setting the security bit

disables both the JTAG and memory ports. JTAG bus master privileges are taken away (although, boundary scan still works) and no external program or data accesses are allowed. Global chip RESETs are ignored even though the MCU-specific RESET still works. The only way to get out of stealth mode is to cut the power off, posing quite a challenge for anyone trying to sneak a peek inside.

FAST CHIPS FAST

The 'E5 brings it all under one roof but raises the specter of an unwieldy lashup of software and hardware tools. Never fear—Triscend offers their FastChip software and a development board (see Photo 2) to get you off on the right foot. The best way to describe FastChip is to show it in action using the "My Design" tutorial that's included.

Photo 1a shows the main-design entry screen. The row of icons along the top correspond to the general development sequence, starting with project definition and proceeding through chip creation (Bind, Download, Debug, etc.).

The tutorial example is simple. It uses the watchdog timer as a once-per-second-or-so interrupt generator. Upon each interrupt, the program increments the Result register and displays it on the EV boards two 7-segment LED displays (i.e., cycles from 00 to FF, indefinitely). There's also a flip-flop (Heartbeat) that toggles the decimal point on the display. On the left you can see the library from which these and other functions can be dragged and dropped onto your design.

The icons in the second row are used to configure the MCU peripheral functions. For example, clicking on the watchdog timer brings up a

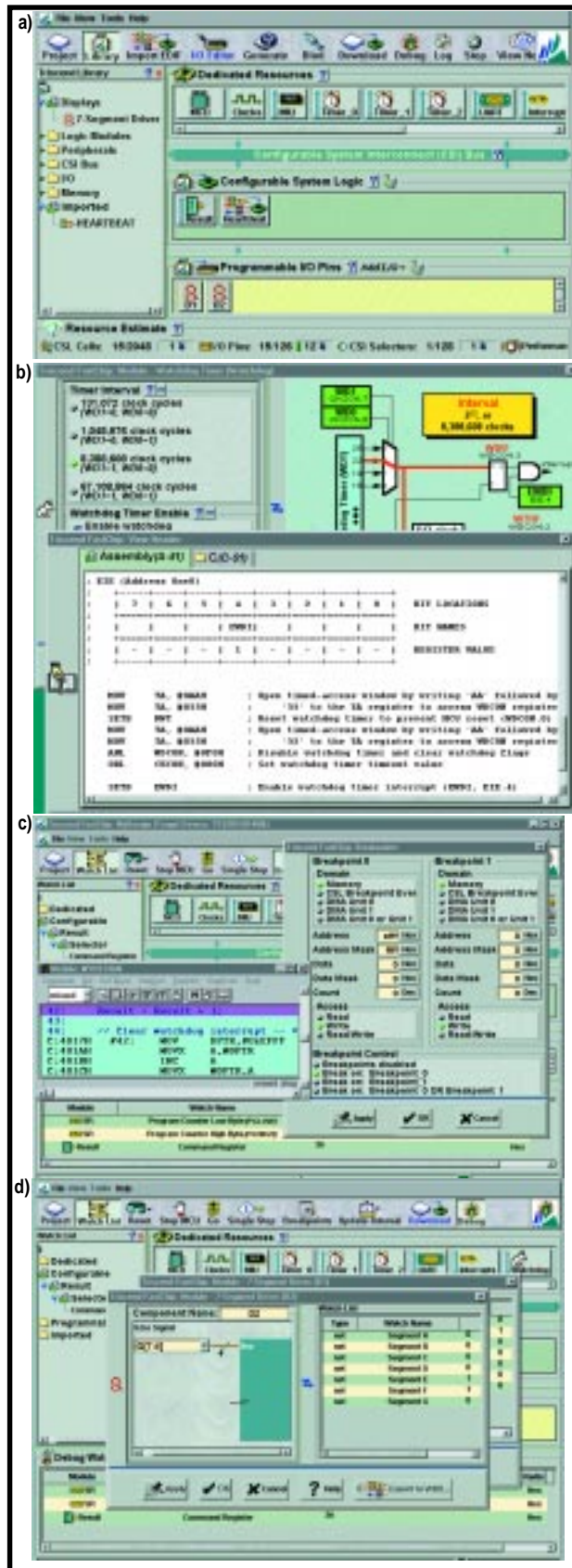


Photo 1—Here's a look at a custom SoC from start to finish. Begin by dragging functions onto your chip (a). Next, configure the MCU peripherals (b), such as the watchdog timer and FastChip, which automatically generates the initialization code. After binding and downloading your design, set up a breakpoint (c). At the breakpoint, examine what's going on (d). The LEDs are showing 36 (i.e., RESULT = 36, which matches the pattern on the 7-segment drivers).

window, allowing simple point-and-shoot setting of the operational characteristics. As shown in Photo 1b, FastChip automatically generates the code (C or ASM) to accomplish your request, comments, and all. This is great—now you can tell your boss that you spent all day writing exceptional code when, in reality, it only takes a few clicks of the mouse.

At that pace, it won't be long before you're ready to download and debug your design. In Photo 1c, I've set a breakpoint for each watchdog interrupt, specifically at the point the Result register is incremented. Notice in the watch area near the bottom, that Result equals 36, and the status line at the very bottom shows the MCU is halted at the breakpoint.

Now, you can poke around to your heart's content. Let's say the LEDs don't actually show 36. Most likely a design bug, but it could also be a short on the board, bent pin, or a bad LED. Just to confirm, in Photo 1d we've

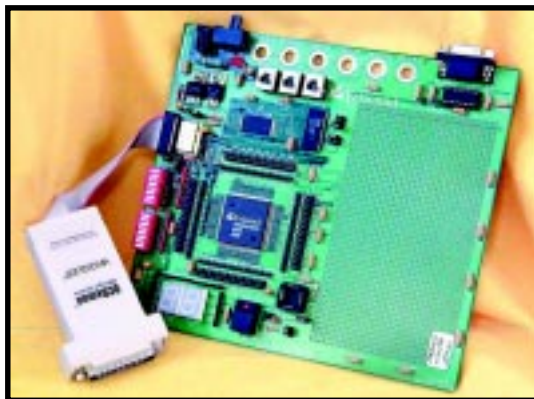


Photo 2—Fittingly, an evaluation board for a configurable System-on-Chip doesn't need much besides a CSoC. There's an SRAM, a parallel flash-memory chip (20-pin socket for the serial option), an RS-232 transceiver, and the requisite LEDs and switches.

clicked on the LED drivers to take a closer look and can confirm that they are indeed driving the proper segment pattern. Much as I like poking at a rat's nest, it's rather enjoyable to debug hardware on a screen!

A lot can be accomplished by using FastChip and its built-in function library. All you need to do is provide a .HEX file with your MCU code, and FastChip does the rest.

However, this only scratches the surface. Triscend is working to integrate popular third-party tools more closely. The software comes with demo versions of Keil for MCU software and Orcad for schematic capture. Other third-party names bandied about include Archimedes (software), ViewLogic (schematic), and Synopsys, Synplicity, and Exemplar (synthesis).

No complaint about the FastChip price. It's only \$499, which is peanuts by chip-design tool standards. Better yet, they offer a free 30-day fully functional trial version so you can try before you buy. The entire development kit, including FastChip, the evaluation board, and a PC parallel port to JTAG adapter, sells for \$1395.

The only caveat? During installation, FastChip reported that it really would be happier with 192 MB of RAM. I only have 64 MB, which was enough to fool around a little with the tutorial, but not much more. In discussions with Triscend, they indi-

Embedded processor core	Dedicated resources	Device	System RAM	Configurable system logic (CSL) cells	CSI address selectors	PIO pins (max)
8032 Turbo (3) 16-bit counters USART Watchdog timer Interrupt controller	High-speed internal bus Memory interface unit 2-channel DMA controller Power management Power-on reset Hardware breakpoint unit JTAG port	TE505	16K × 8	512	32	124
		TE512	16K × 8	1152	72	188
		TE520	40K × 8	2048	128	252
		TE532	64K × 8	3200	200	316

Table 1—With up to 64 KB of SRAM, hundreds of pins, and thousand of logic cells, the 'E5 is definitely an 8-bit MCU with attitude.

cated I would have problems if I tried to actually create my own chip (i.e., do a Bind) with so little RAM. Of course, I tried it immediately.

Finally resorting to CTL-ALT-DELETE after about half an hour of my hard disk playing washing machine, I've got two things to say to the Triscend engineers. First, you were right about the RAM, and second, you might add a progress bar and ESC key feature for the next RAM-poor fool who tries pushing their luck.

SoCCESS STORY

Overall, I was impressed by the 'E5. The MCU does a good job modernizing the popular 8032/52, and the

FPGA seems decent enough. I give Triscend a lot of credit for being one of the first to run hard with the MCU + FPGA concept.

Yes, there are a lot of great MCUs and FPGAs around, and other companies can, and will, introduce similar combinations. However, Triscend seems to understand there's more to the CSoC concept than just cut-and-pasting separate chips onto a single piece of silicon.

The real keys to success are how cleverly the functions are combined (interconnected) and development tools that live up to the "S" in SoC by handling the entire design spectrum from gates to C.

Keep an eye on Triscend—I think they get it. 📧

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for over a decade. Reach him by e-mail at tom.cantrell@circuitcellar.com, by telephone at (510) 657-0264, or by fax at (510) 657-5441.

SOURCE

TE505 CSoC
Triscend Corp.
(650) 968-8668
Fax: (650) 934-9393
www.triscend.com

CIRCUIT CELLAR Test Your EQ

Problem 1—Johnny is tasked with designing the trigger circuit for the free world's latest super-secret weapon. This circuit must be built on a double-sided PCB to reduce cost and it must operate in an extremely noisy environment with considerable ESD and EMI.

Earl, a guy at the local tavern, suggested that Johnny should flood all of the space between traces on the PCB with copper using the PCB layout tools copper fill function since a changing electromagnetic field (EMI) cannot penetrate a conductive surface (See Figure). Should Johnny use Earl's advice?

Problem 2—In the world of computer science what is priority inversion?

Problem 3—Given a two-phase stepper motor that can be configured either as bipolar or unipolar, which configuration saves energy?

Problem 4—A pair of generic 4 MB dynamic RAM chips configured as 1048K × 4 are used to implement a 1 MB × 8 memory array for an embedded 8-bit microprocessor. The dynamic RAM chips are rated 60-nS access time. In operation, a minimum of 120 nS per byte is required to transfer a block of data into or out of the memory using standard read/write cycles. Why?

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

ADVERTISER'S INDEX

The Advertiser's Index with links to their web sites is located at www.circuitcellar.com under the current issue.

Page		Page		Page		Page	
93	Abacom Technologies	C4	Dataman Programmers, Inc.	54	megatel	15	Sealevel Systems
91	Ability Systems Corp.	94	Decade Engineering	84	MetaLink Corp.	91	Senix Corp.
94	ActiveWire, Inc.	87	Designtech Engineering	93	microEngineering Labs, Inc.	87	SiGEM, Inc.
34	ADAC	4,5	DreamTech Computers	39	Microchip	84	Signum Systems
18,31	Advanced Transdata Corp.	49	Earth Computer Technologies	35,66	Micromint, Inc.	92	Sirius microSystems
38	AllElectronics	47	ECD (Electronic Controls Design)	47	Midwest Micro-Tek	86	SMTH Circuits
92	Amazon Electronics	85	EE Tools (Electronic Engineering Tools)	87	MJS Consulting, Inc.	91	Softaid
94	Andromeda Research	87	ELNEC	17	Motorola	92	Software InnoVations, Inc.
90	AP Circuits	52	EMAC Inc.	25	NetBurner	82	Solutions Cubed
64	Arcorn Control Systems	92	Embedded Micro Software	84	Nohau Corp.	84	Square 1 Electronics
92	Ascendco Scientific, Inc.	47	Engineering Express	55	On Time	89	TAL Technologies
90	Autotime Corp.	85	FDI (Future Designs, Inc.)	78	OS Systems, Inc.	8,48,53	Technologic Systems
92	Avocet Systems	14	Fire, Wind, and Rain Technologies, LLC	94	PCB Express, Inc.	86	Technological Arts
33	Axiom Manufacturing	91	General Device Instruments	C2	Parallax	59,C3	TechTools
94	Bagotronix, Inc.	48	General Software	88	Peter Anderson	87	Tern, Inc.
90	Bay Area Circuits	85	Hagstrom Electronics	86	Phoenix International Corp.	93	Triangle Research
91	Bectern Inc.	92	Huntsville Microsystems, Inc.	84	Phytec	28	Trilogy Design
93	Borge Instruments Ltd.	89	IMAGEcraft	94	Picofab, Inc.	78	Trinity College
7	CAD-UL	89,90	Intec Automation, Inc.	69	Point Six, Inc.	85	Vesta Technology
90	Capital Electro-Circuits, Inc.	81	Interactive Image Technologies, Ltd.	85	Pontech, Inc.	86	Vetra Systems Corp.
29,91	CCS (Custom Computer Services)	88	International Electronics Corp.	92	Prairie Digital, Inc.	88	Virtual Tools, Inc.
91	Ceibo	85	Intronics, Inc.	88	Pulsar, Inc.	89	WCSC
8	ChipCenter	79	Jameco	74	Rabbit Semiconductor	61	Weeder Technologies
91	ChipTools, Inc.	65, 89	JK microsystems	92	R.E. Smith	1	Wilke Technology GmbH
94	Conitec	69	JR Kerr Automation & Engineering	61	Remote Processing	90	Wirz Electronics
29,93	Connecticut microComputer, Inc.	86	J-Works, Inc.	90	RLC Enterprises, Inc.	89	Z-World
90	Copeland Electronics, Inc.	9	Link Instruments	93	RMV Electronics, Inc.	94	Zanthic Technologies Inc.
85	Creative Control Concepts	88	Lynxmotion, Inc.	13,71,86	Saelig Co.	23	ZiLOG
86	Crystalfontz America, Inc.	88	Matrix Orbital Corp.	69	Scidyne		
41	Dallas Semiconductor	94	MCC (Micro Computer Control)	28	Scott Edwards Electronics, Inc.		

Keep Your Signals Straight—Hadamard Encoders and Decoders in C++

A Quick Meter Made—Fast Frequency Measurement for Low Frequencies

Building a RISC System in an FPGA—Part 2: Pipeline and Control Unit Design

Buying Power—Low-Cost Power Supply for Embedded Applications

The Shocking Truth about EMC—Part 1: Design for Compatibility

 **MicroSeries: Op-Amp Specifications**—Part 1: Served Italian Style

 **From the Bench: Dropping the Incredible Bulk**—Using Capacitors as Isolating Components

 **Silicon Update: A Winter Timer Tale**

EPC Real-Time PC: POST It—Build a Power-On Self Test Card for PC/104

EPC Applied PCs: Under the Covers—Get Embed(ded) with Windows NT 4.0

Signal Processing

PK 117

PRIORITY INTERRUPT

Y2K Phooey



swear this is the last thing I'll write about Y2K. OK, a year ago I said there could be *some* Y2K glitches. As engineers, we know computers are only as good as the program code they run. If a lazy programmer didn't insert some kind of error checking or fallback provisions, *it's possible* that the code could stop dead in its tracks.

Maybe it was that fault of engineers like myself, who when asked if there could be difficulties, answered it honestly.

What we didn't add was whether this uncertainty was significant or not. We should have thought more about who was listening to the message and not just that we were answering a technical question. I can't speak for all of you, but I know that when someone asks me if a technical problem can be solved, I tend to answer it in engineering speak. In essence, if the task is 99% solvable and 1% trouble, I will say, "Yes, it can be done, but situation X could make it not work." It can easily sound to someone that the odds are 50/50 for either situation.

Because we technical people are so involved in solving tasks, we often feel responsible when we are unable to do something. As a result, we automatically emphasize the tiny gotchas that inhibit 100% positive results rather than underscoring the 99% we might get right. Certainly, it has a lot to do with our personalities, but it also has a lot to do with how engineers view technical tasks. An engineer designing an anti-lock brake system for a car considers it a total failure if the end result has a 0.001% probability of not engaging at the right time. This is a radical example of course, but I think you get my drift.

There is also media-speak. These days, accuracy is a matter of interpretation and journalists will often see events with the opposite slant that you might. For example, if there is a new dot com IPO on the horizon with a 1% potential and 99% risk you'll read or hear, "In light of the vast fortunes being made on dot com companies these days the *potential* for success is assured."

As for Y2K, we've been had. Informed technical people answered the media's questions about the Y2K risk that *there was some possibility of isolated problems*. This was reinterpreted by less-technical pundits like Ed Yardeni, an economist for Deutsche Bank, who told Fortune magazine that the probability of recession was 70%! In the end, the media frenzy prophesied a scenario of 100% catastrophe. A lot of scared people started thinking about bomb shelters for the first time in 35 years and bought enough bottled water to fill Lake Erie.

By now we all know what really happened on Jan 1st. Nothing! Nada! Zero! I mean, the media needs to have their clocks cleaned for this fear fest! And remember people, this the same gullible group that we count on to provide the news tomorrow!

Were all of us who knew it shouldn't be all that bad, stupid? Or, just not vocal enough? Was this Y2K hype so well orchestrated that we were blindsided?

I'd like to say I was totally unaffected, but I wasn't. No, I didn't stock up on water, buy a generator, or fill all the kerosene heaters (we have all that stuff already out here in snow country). Instead, we threw a New Year's party! That was the good news. The bad news was all the hysteria again. We had a few people tell us that they don't venture out on New Year's Eve, but we also had one couple call at the last minute and tell us they were just too frightened to go out. Of course, they live in a town that had block watchers on every street with emergency flags ready to signal roving police cars because (according to the media) the power and telephone system would inevitably fail. It's no wonder they were frightened.

It was interesting that night to watch the midnight celebrations as they progressed across the Pacific and Europe. By the time the Eiffel Tower lit up the midnight sky in Paris, everyone at my party (primarily business professionals) was questioning why there were no reports of catastrophe anywhere. The media was certainly on the lookout all over the world but there hadn't even been a traffic light that didn't work correctly from what I saw.

I read someplace that the US spent about half a trillion dollars on Y2K. AT&T spent \$500 million alone. How much do you think Belize spent on Y2K readiness? OK, I'll put a damper on my own hype here but wasn't Russia supposed to implode on Jan 1st? Weren't Greece, Italy and a pile of other modern countries that depend upon computers, and who weren't spending 50% of their GNP on programmers, supposed to have gone belly up on the 1st?

Another columnist recently said that if we went to the doctor and ended up with a heart bypass instead of an antacid that should have been prescribed, we'd probably go berserk. As a matter of expenditure versus the reality of the problem, I think we did far too much to achieve "Y2K compatibility." For all of us businesses who spent thousands of dollars updating software and operating systems to satisfy the demands of frightened customers and financial institutions, all I can say is phooey. Y2K was a significant non-event and it cost a lot more than it should have. It is said that history is destined to repeat itself if we don't learn from the past. Surely, if we don't want to repeat this fiasco, then we should also heed another wise old adage as it applies to Y2K beneficiaries: Follow the money!