

EMBEDDED PC
MONTHLY SECTION

CIRCUIT CELLAR

INK[®]

THE COMPUTER APPLICATIONS JOURNAL

#97 AUGUST 1998

DEBUGGING TECHNIQUES

PROM Programmer Construction

Capturing Mixed Signals

Memory Management
in Smart Cards

Debugging an
Embedded PC
Application



\$3.95 U.S.
\$4.95 Canada

TASK MANAGER

Almost Made 100



few months ago, I wrote in this column about the beginnings of *Circuit Cellar INK* and described how I've been involved with the magazine in some capacity since the very first issue. I suppose that column could be viewed as a foreshadowing of what

was to come.

I've always been an engineer at heart. My parents like to tell tales of how, as a child, I had a see-through bulldozer that I took apart—at least a million pieces—and put back together without batting an eye. Through all the years I've been involved with editing *Circuit Cellar INK*, I've always kept a hand in the engineering side of things. It's been the therapy I could fall back on when the stresses of deadlines and printer mistakes became too much.

I've reached a point now that I've had to make a choice: engineering or editing? The answer has always been obvious, but not necessarily easy. As of this issue, I'm turning over the reins to my fellow editors on the *INK* staff and will be pursuing design activities full time. I'll continue to have a hand in the magazine's production on the side, so you'll continue to see my name from time to time. Maybe I'll actually have time to write an article or two. I'll also continue to be active in the *Circuit Cellar* newsgroups, so you haven't heard the last of me.

Thanks to the many of you who have become friends over the years, providing constructive feedback and encouragement. I trust we'll be able to maintain the relationships even though I've moved on.

I'd also like to take this opportunity to thank my colleagues at *INK* for all I've gained from them, whether it be Tom's enthusiasm and commitment on the West Coast, Elizabeth's exceptional attention to detail, KC's amazing Photoshop capabilities, or Janice's insistence on high editing standards in each and every issue. Such a strong team, along with Steve's determination to offer application-oriented, embedded-systems articles, will ensure *INK* readers the same level of editorial excellence. I pass my mantle on with confidence that the job will continue to be well done.

Now, before I start packing my files, let me show you the kind of editorial quality I'm talking about. We lead off with the first writeup of one of our Design98 contest winners. Norman Jackson took first place in the PIC16XXX category with BitScope, a mixed-signal capture engine. Mike Smith and Jason Wudkevich follow with a look at how to use virtual devices to test and debug your system and its peripherals. Bobby Crouch finishes the two-part series on smart cards by showing you how to juggle memory so that your smart-card design stays secure. Don Lancaster closes our feature section with an introductory look at how to keep vector-to-step conversions as fast and smooth as possible.

Stuart Ball launches a new MicroSeries on how to build a PROM programmer. His first article focuses on putting the hardware together. This month both Jeff and Tom are bent on power. Jeff is fed up with wall warts and is looking for transformerless power conversion, while Tom simply wants to check out the newer brawn. He takes a look at offerings from Linear Technology, Maxim, Motorola, and National Semiconductor.

Ernie Deel starts off the *EPC* section by examining interprocess communication via anonymous pipes, and Edward Steinfeld checks out FAT32, a file system for data-intensive applications. In the columns, Ingo looks at real-time data acquisition and Fred checks out how to debug using the Net186.

ken.davidson@pobox.com

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Sue (Hodge) Skolnick

EDITOR-IN-CHIEF

Ken Davidson

CIRCULATION MANAGER

Rose Mansella

MANAGING EDITOR

Janice Hughes

BUSINESS MANAGER

Jeannette Walters

TECHNICAL EDITOR

Elizabeth Laurençot

ART DIRECTOR

KC Zienka

WEST COAST EDITOR

Tom Cantrell

ENGINEERING STAFF

Jeff Bachiochi

CONTRIBUTING EDITORS

Ingo Cyliax

Fred Eady

Rick Lehrbaum

PRODUCTION STAFF

Phil Champagne

John Gorsky

James Soussounis

NEW PRODUCTS EDITOR

Harv Weiner

Cover photograph Ron Meadows—Meadows Marketing

Many thanks to Alfred Baier of Manchester Center, VT,

for sculpting the pipe on the front cover just for *INK*.

PRINTED IN THE UNITED STATES

ADVERTISING

ADVERTISING SALES REPRESENTATIVE

Bobbi Yush

(860) 872-3064

Fax: (860) 871-0411

E-mail: bobbi.yush@circuitcellar.com

ADVERTISING COORDINATOR

Valerie Luster

(860) 875-2199

Fax: (860) 871-0411

E-mail: val.luster@circuitcellar.com

CONTACTING CIRCUIT CELLAR INK

SUBSCRIPTIONS:

INFORMATION: www.circuitcellar.com or subscribe@circuitcellar.com

TO SUBSCRIBE: (800) 269-6301 or via our editorial offices: (860) 875-2199

GENERAL INFORMATION:

TELEPHONE: (860) 875-2199

FAX: (860) 871-0411

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

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

AUTHOR CONTACT:

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

ARTICLE FILES: ftp.circuitcellar.com

For information on authorized reprints of articles,
contact Jeannette Walters (860) 875-2199.




CIRCUIT CELLAR INK®, THE COMPUTER APPLICATIONS JOURNAL (ISSN 0896-8985) is published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (860) 875-2751. Periodical rates paid at Vernon, CT and additional offices. **One-year (12 issues) subscription rate USA and possessions \$21.95, Canada/Mexico \$31.95, all other countries \$49.95. Two-year (24 issues) subscription rate USA and possessions \$39, Canada/Mexico \$55, all other countries \$85.** All subscription orders payable in U.S. funds only via VISA, MasterCard, international postal money order, or check drawn on U.S. bank.

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

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

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

Entire contents copyright © 1998 by Circuit Cellar Incorporated. All rights reserved. *Circuit Cellar INK* is a registered trademark of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

- 12 BitScope**
A Mixed-Signal Capture Engine
Norman Jackson
- 22 Simulating Micro-Controlled Systems**
Mike Smith & Jason Wudkevich
- 60 Designing for Smart Cards**
Part 2: Practical Implementation
Bobby Crouch
- 66 Vector-to-Step Conversions**
An Introductory Tutorial
Don Lancaster
- 70**  **MicroSeries**
Build a Serial Port PROM Programmer
Part 1: Hardware Construction
Stuart Ball
- 78**  **From the Bench**
Transformerless Power Conversion
Jeff Bachiochi
- 82**  **Silicon Update**
Power Trip
Tom Cantrell

- Task Manager** 2
Ken Davidson
- Almost Made 100**
- Reader I/O** 6
- New Product News** 8
edited by Harv Weiner
- Advertiser's Index/
September Preview** 95
- Priority Interrupt** 96
Steve Ciarcia
**When Boilerplates
Won't Do**

INSIDE ISSUE 97

EMBEDDED PC

- 32 Nouveau PC**
edited by Harv Weiner
- 36 Interprocess Communication
Using Anonymous Pipes**
Ernie Deel
- 42 FAT32—File System for Data-Intensive Applications**
Edward Steinfeld
- 49** RPC **Real-Time PC**
Data Acquisition
Ingo Cyliax
- 55** APC **Applied PCs**
Debugging & the Net186
Fred Eady

www.circuitcellar.com
★ August Password: Sherlock ★

READER I/O

CONSIDER THE ALTERNATIVES

INK 96 came with a surprise. It included an article on FreeDOS, which complies to the GNU GPL and hence is totally free. But, freedom comes with a price: there's still no stable version (according to freedos.org).

What about using PTS-DOS? It is over 99.99% MS-DOS compatible, written in 80x86 assembly language, and comes with full sources. The makers restrict users to the extent that one user may not base a completely new OS on it, but you can overcome this by buying one CD with each version or by paying a fee to the makers.

PTS-DOS's native mode is "flat real mode." When I load a segment register with zero, I can use EBX as a 32-bit pointer in linear address mode. Now, I only have to load HiMem386.SYS and I'm off, with all the freedom of real mode but the addressing of protected mode.

PTS-DOS, with full sources, costs a mere \$50. Perhaps it's time to look at this software masterpiece that's been shipping, selling, and performing for over four years.

Jan Verhoeven
aklasse@tip.nl

AMAZING THE DIFFERENCE OF A MAGNITUDE!

After reading my article "Using the PC for Radiation Detection" in *INK* 96, one of my colleagues, Dr. Ed Webb, E-mailed me to point out a decimal error: 1 nCi of activity is equal to 37 disintegrations per second, not 0.37 as I said in the article.

Therefore, my basement should reach the EPA limit for radon in 0.388 days, not 38.8 days as I calculated. So, I should run my fan more often, about 2 hours every 12 hours, rather than once per month.

Dan Cross-Cole
crosscol@erols.com

WEARABLE COMPUTING—WHERE TO SHOP

The "WearCam" article (*INK* 95) neglected to mention that there's an entire community dedicated to wearable computing. For example, HandyKey's Twiddler keyboard provides an off-the-shelf method for one-handed typing and mousing at speeds up to 60 wpm. As well, there is a public mailing list for people

building wearable computers maintained at wear-hard@haven.org.

The annual conference for wearable computing, the IEEE International Symposium on Wearable Computers (ISWC), will meet this year October 19–20 in Pittsburgh, PA. Details may be found at <wearables.www.media.mit.edu/projects/wearables/>.

Thad Starner
testarne@media.mit.edu

CAN I SEE, TOO?

I was interested in Steve's Priority Interrupt editorial on "Design98—A Marketer's PICnic" (*INK* 94). If I am interpreting his comments correctly, he believes the Design98 contest was a huge success.

However, there's just not enough pages in *INK* to publish all those great design entries. Is there any chance that the rest of us will be able to share in these designs?

JR Morgan
AQFRTSTR@aol.com

For more information on the projects submitted for Design98, be sure to check out the new Design Forum on the Circuit Cellar Web site—especially the section entitled PIC Abstractions.

READERS, SPEAK UP!

A lot of computing magazines used to publish articles by people who were deeply interested in how things work. A lot of us never built the exact projects shown in the articles, but we learned something from each one of them. Frequently, it's the case that we get our best ideas from looking at what others have done. We improve on what we have learned, twist it, and turn it to new purposes.

Circuit Cellar INK is great because it still has the right elements, and I'll continue subscribing as long as it does. Although there may be ways *INK* can increase its reader satisfaction level, I certainly think readers have a responsibility to make their interests known.

Tom McGahee
tom_mcgahsee@sigmais.com

NEW PRODUCT NEWS

Edited by Harv Weiner



ELECTROLUMINESCENT LAMP DRIVER

The **IMP803** is a high-voltage, low-power electro-luminescent (EL) lamp driver that generates the 180-Vp-p drive signal needed to excite an EL lamp from a low-voltage DC source like a battery. EL lamps are used to backlight LCDs and keypads in low-power portable personal communications devices such as pagers, mobile phones, and personal digital assistants (PDAs).

The **IMP803** incorporates four EL-lamp-driving functions on-chip: the switch-mode power supply, its high-frequency oscillator, the high-voltage H-bridge lamp driver, and its low-frequency oscillator. By boosting an input voltage of 2–6 V up to 180 Vp-p (maximum), EL lamps of 30-nF capacitance are driven to high brightness. If the lamp-drive voltage reaches 180 Vp-p, an internal circuit adjusts the boost converter to save energy.

Two external resistors permit the switching regulator frequency and lamp-drive frequency to be adjusted. Connecting these resistors to ground places the **IMP803** into a powered-down mode.

The **IMP803** is available in an eight-pin SO package or in die form for high-volume, low-cost chip-on-board applications. The 1000-quantity price of the **IMP803LG** is **\$0.96** each. An evaluation board, the **IM803EV1**, is available at an introductory price of **\$25**.

IMP, Inc.
(800) 438-3722 • (408) 432-9100
Fax: (408) 434-0335 • www.impweb.com

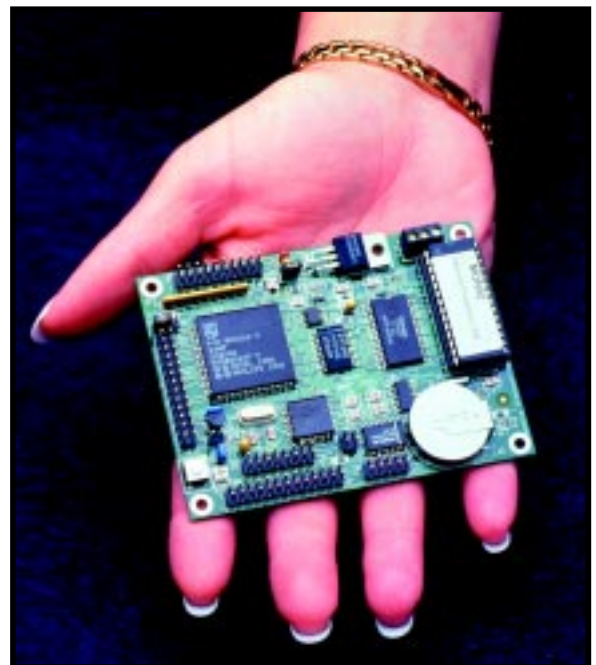
8051-COMPATIBLE EMBEDDED CONTROLLER

The **RPC-220** is a compact, 8051-compatible embedded controller that's designed for monitoring or simple control applications. The 3.85" × 2.85" form-factor board contains flash memory to enable remote program updates by modem or notebook, a year-2000-compliant real-time clock (which can wake up the card from low-power modes), and a battery backup of its 512 KB of RAM. Power inputs are 5.0/5.4 to 25 V, and a regulated 5 V is available to power user circuits.

Its 19 digital I/O lines can also be used for pulse output (two PWM and two square wave) and pulse width measurements. Analog I/O is accomplished through an eight-channel, 10-bit converter and a two-channel analog output. An LCD port enables display of current information, and hardware and software RS-232 ports are available.

The **RPC-220** development system runs on Windows-based PCs. The development system includes an **RPC-220** board, hardware and CPU manuals, cables, terminal boards, power supply, C compiler, and more than a dozen stand-alone application programs. Prices start at under **\$100** in quantity. A development system costs **\$435**.

Remote Processing
(800) 642-9971 • (303) 690-1588
Fax: (303) 690-1875
www.remotep.com



NEW PRODUCT NEWS

TWO-AXIS SERVO CONTROLLER

The **DC2-STN** is a stand-alone two-axis servo controller with two auxiliary outputs for stepper control. Its small footprint (9.5" × 4" × 5"), back-panel mounting tabs, and removable/pluggable screw terminals make it well suited for OEM applications. It operates on universal AC voltage (100–240 VAC, 50/60 Hz) and has an integrated DC power supply.

Features include an RS-232 communication interface for programming and networking as well as 32 KB of nonvolatile memory to store macros, programs, and point information. It offers optoisolated I/O at industrial voltage levels, status LEDs, and a watchdog relay output.

The DC2-STN supports point-to-point positioning, velocity control, master/slave and gearing, joystick input for manual positioning, circular and linear interpolation, PID filter with velocity and acceleration feed forward, and the ability to change parameters and targets on the fly. The command set supports macros, sequencing, user data-registers, and more. Its multitasking capability permits four independent tasks or routines to run simultaneously without interrupting motion control.

The DC2-STN costs **\$995** in single quantities. OEM discounts are available.

Precision MicroControl Corp.

(760) 930-0101 • Fax: (760) 930-0222 • www.pmccorp.com



NEW PRODUCT NEWS

MULTICHANNEL DATA ACQUISITION

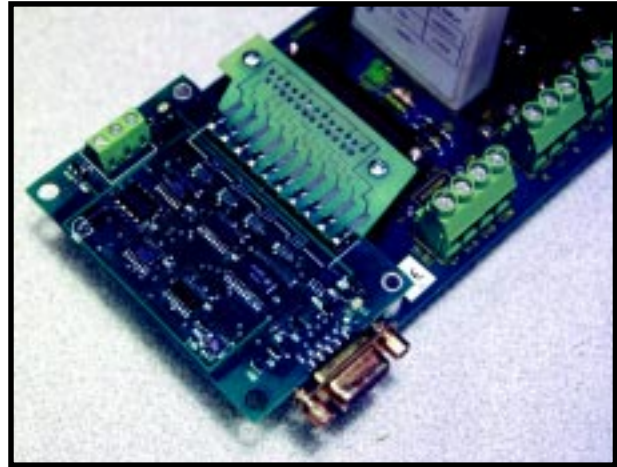
Point Six, an authorized software developer for Dallas Semiconductor, has developed an RS-232-powered multi-channel analog interface card based on Dallas's one-wire technology.

Features include eight channels of analog input on a credit-card-sized board. All eight channels are individually configurable as a 0-5-V 12-bit analog input. The **T8AH7BA** contains a built-in multidrop controller that provides a unique 64-bit registration number, assuring error-free selection and absolute identity of the device. No two parts are alike. The unique addressing enables inputs to be identified absolutely.

The T8AH7BA directly connects to 7B-series industrial-standard analog isolation back panels and modules. All necessary power is derived from the RS-232 port. The T8AH7BA has a built-in RS-232-to-one-wire interface, allowing network expansion to drive up to 200 one-wire devices (e.g., temperature, pressure, force, humidity, pH) over as much as 2000' of Cat-5 twisted-pair cable. All data transfers are CRC-16 error checked. A DDE driver permits effortless interface to most Windows applications.

The T8AH7BA is priced at **\$99.95** in single quantities.

Point Six, Inc.
(606) 271-1744
Fax: (606) 271-4695
www.pointsix.com



NEW PRODUCT NEWS

COMPUTER-CONTROLLED PAN-TILT UNIT

The **PTU-46-70** offers high resolution at 0.771 arc minutes (less than $\frac{3}{4}$ " at 100 yd.) with speeds at 60° per second and a load capacity of 6 lbs. Applications include computer vision, security, test and measurement, and teleconferencing, as well as photography, video, and special effects. The **PTU-46-17.5** is a high-performance, low-cost, computer-controlled pan-tilt unit that moves over 300° per second with 3.086 arc-minute resolution at a load capacity of over 4 lbs.

The unit features plug-and-play using an RS-232 terminal, enabling the host computer to accurately control pan and tilt speed, acceleration, and position. Self-calibration on reset ensures reliable absolute positioning. Microprocessor control and constant current bipolar drives provide efficient high-speed movement, and the host

can make on-the-fly position and speed changes. Built-in RS-485 multidrop network capabilities mean a single host port can control up to 127 pan-tilt devices.



The 3" × 5.11" × 4.25" unit has power-management controls and flexible input power requirements. Host commands control power consumption, which is 16 W peak at full power, 7.5 W peak at low power, and 1 W peak with motors off.

Single-quantity prices are **\$1800** for the PTU-46-17.5 and **\$2100** for the PTU-46-70. Options include international AC/DC power supply, joystick interface, nodal adapter, weatherizing, and C programmer's interface.

Directed Perception, Inc.
(650) 342-9399 • Fax: (650) 342-9199
www.dperception.com

FEATURES

12 BitScope

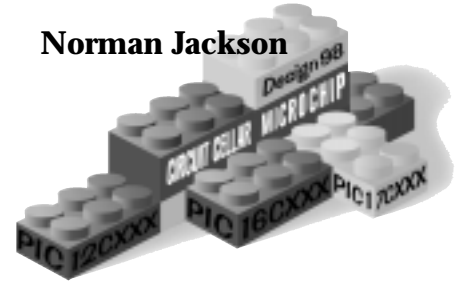
22 Simulating Micro-Controlled Systems

60 Designing for Smart Cards

66 Vector-to-Step Conversions

FEATURE ARTICLE

Norman Jackson



BitScope

A Mixed-Signal Capture Engine

Has your office become so cluttered that you can't find your oscilloscope or logic analyzer? No problem, Norman will help you build a low-cost, mixed-signal capture engine that connects to your computer via the serial port.



Some time ago, I had a bad experience with a bus—a logic bus. It had six ram-paging DSP cards and a SCSI controller all trying to ride at the same time.

About once an hour, there was a sickening crash. After going through the usual stages of blaming the software, I relented, admitted possible culpability, and borrowed a mixed-mode DSO.

This machine has a digital sampling oscilloscope and a logic analyzer effectively joined at the hip. They share a common trigger module that enables the user to identify a complex event and record the state of the target hardware before and after the trigger—in both the analog and digital domains.

In the case of my erratic bus logic, the culprit turned out to be a delinquent GAL with a ground bounce problem. The offending chip had its duties reassigned and the documentation police were alerted. Engineer triumphs over bug.

By employing a high-tech piece of test equipment, I could trigger on a complex digital event and correlate this event to an oscilloscope trace that showed what was really happening in the analog domain. I was saved in the nick of time, but despite having formed

a deep attachment to the trusty 'scope, I had to give it back.

Following this adventure, I started musing about how to roll my own version of that useful electronic gadget. After some mental tinkering and with the added incentive of Design98, I was soon sketching electronic stuff on the grid pad. BitScope began to emerge (see Photo 1).

THE BIG PICTURE

The basic idea behind BitScope is that of a specialized piece of data-capture hardware that doesn't include any user interface other than an RS-232 plug. Most engineers have more computers, mice, and keyboards than they know what to do with. If I was going to build a cheap 'scope, I certainly didn't want any more of that stuff.

What I needed was an electronic drone that could capture and disgorge data on command. No more, no less.

The commands had to be simple ASCII characters that are intuitive and easy to learn. The PC-based user interface can then synthesize functionality of arbitrary complexity by sending scripts of command characters and receiving the replies.

The answer: a virtual instrument where specialized hardware does the electronic test job and a PC lets the engineer drive it. One big advantage of this setup is that changing the way the virtual instrument works doesn't usually involve reprogramming chips (hard) but may be done by downloading a new program from the 'Net (easy).

As described in the sidebar "Virtual Machine Architecture," the microcontroller firmware is designed as a virtual machine (VM). The BitScope design is novel because it has an unusual arrangement of the VM program code. The

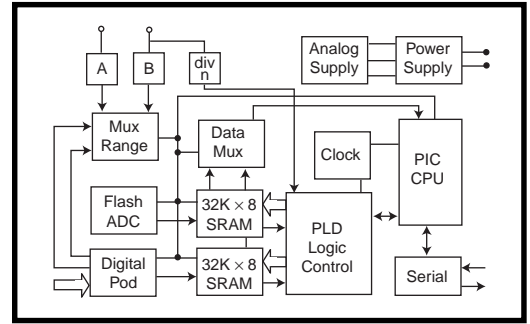


Figure 1—This block diagram of the mixed-signal capture engine shows basic design architecture.

instructions are not located in memory on the microcontroller but reside in the user interface and are executed atomically direct from the serial port.

If you study BitScope's virtual instruction set, you see that arranging things in this crazy way has its advantages. All instructions are atomic. In other words, there is no inherent syntax associated with any command byte.

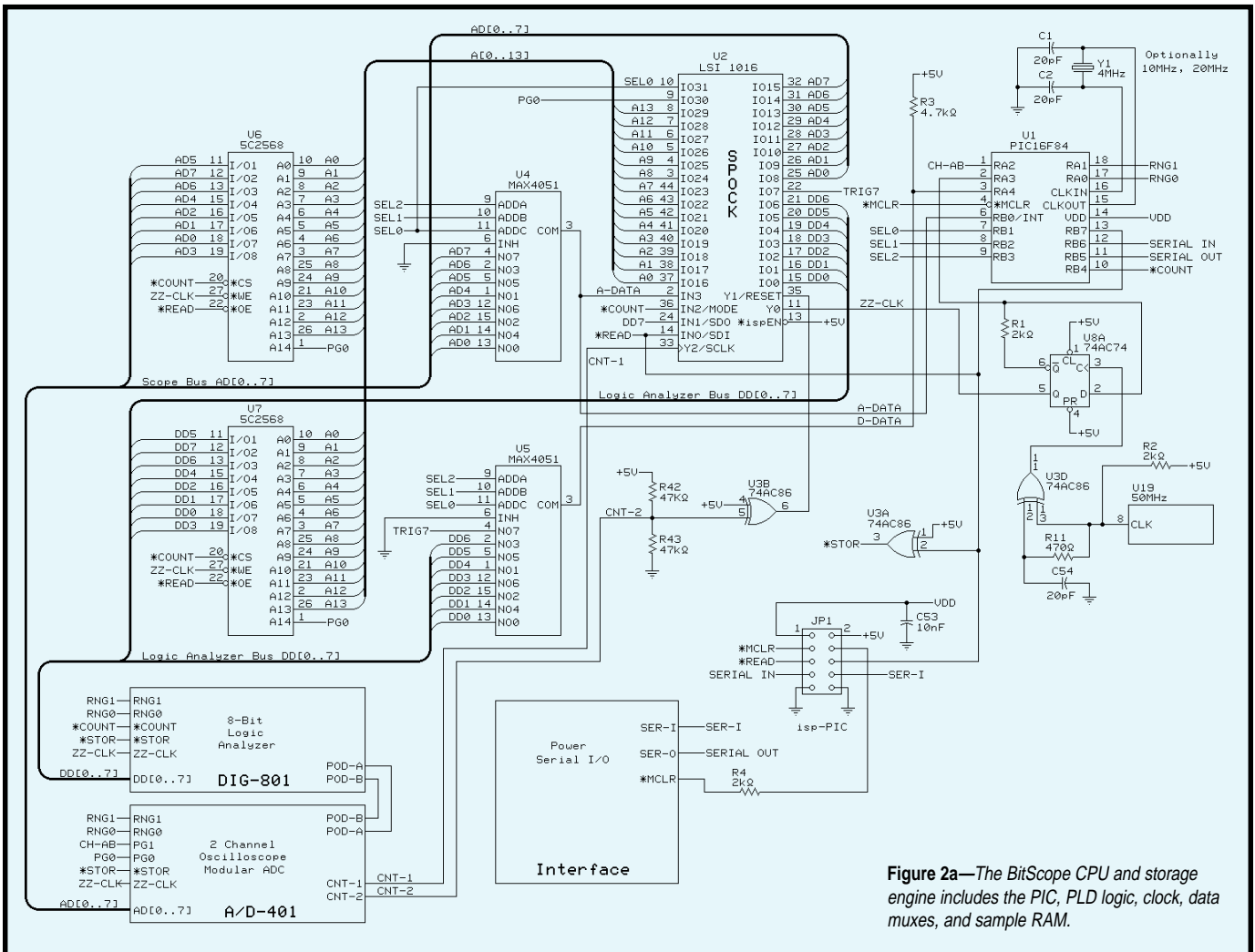


Figure 2a—The BitScope CPU and storage engine includes the PIC, PLD logic, clock, data muxes, and sample RAM.

Virtual-Machine Architecture

A virtual machine (VM) consists of a fully functional processor hosted on an unrelated substrate machine. VM design has advantages over conventional coding. Each instruction may be highly optimized for performance—unlike a general-purpose interpreter like BASIC, which can do anything but inefficiently. VM instructions are compact like assembly but perform extremely complex tasks. Once a register and command set are devised, you can add new instructions to enhance the machine. The original instructions remain the same, which promotes modularity. Since the operational definition of the VM is rigid, firmware changes tend to be straightforward, even to the point of hosting the target architecture on a completely new substrate.

In this design, the PIC16F84 is a substrate to implement a custom BitScope machine with its instruction set becoming microcode to implement the VM. So, the BitScope VM has instructions and registers but they're unrelated to the PIC native instruction set. The virtual registers are hosted by PIC memory registers but have meaning only to the BitScope. Similarly, BitScope has no use for XOR- or DECFSZ-type instructions. Instead, it has instructions for manipulating registers, starting sample RAM, and dumping captured data. BitScope registers may be option bits, timer constants, sample address, and so on. The exact function of the register set is detailed in Table i. Table ii shows the current command set.

Most interpreters run from a program stored in memory. BitScope is different because it executes directly from

R0	Byte Input Reg	Assemble input data here
R1	Register Pointer	Pointer to R(0–ff)
R2	Register Source	Pointer to R(0–ff)
R3	Sample Preload L	Low byte of RAM addr to load to Spock
R4	Sample Preload H	High byte of RAM addr to load to Spock
R5	TRIG Logic Byte	Logic levels for Spock to match, loaded during Spock Init
R6	TRIG Mask Byte	Don't Care bits in trigger match, loaded during Spock Init
R7	Spock OPTION byte	TRIG and PG1 setup in Spock
R8	Trace Register	Trace Option controls Sample operation of BitScope
R9	Counter capture Lo	Counter low byte shifted out of Spock
R10	Counter capture Hi	Counter high byte shifted out of Spock
R11	DELAY-L	Post TRIG delay before halting
R12	DELAY-H	Post TRIG delay before halting
R13	TimeBase	TimeBase expander count
R14	Channel A/B	Channel Range settings for Chop
R15	Dump Length	Counter for number of samples transmitted per request
R16	EEPROM Data	Data register for EEPROM
R17	EEPROM Address	Address register for EEPROM
R18	POD Transmit	Register holds byte for POD
R19	POD Receive	Register gets byte from POD

Table i—The BitScope virtual machine has a set of 20 registers. The operation of the machine and all its instructions refer to these registers.

the serial port. BitScope's instruction set is designed to have no syntax, so there can be a maximum of 256 instructions and each is stand alone—just like a RISC instruction set. An atomic protocol means the software at both ends of the serial line is simple and does not have to preserve state information. In a PIC with 1024 words of program, it's advisable to be economical with code, espe-

00 •	Reset	Reset the machine and print its ID string	54 T	Trace with TRIG stop	Begin sample with Opt mode, until Trig then Delay, Halt Sample Clk, and print sample add.
23 #	Load Source Reg	Store R0 into R2. Set up R2 which is a source reg. A reg-to-reg move may be done by pointing to a source (R2) and destination (R1).	5b [Clear R0	Reg R0 is cleared. This usually precedes a nibble load
2b +	Inc REG	Incr the reg pointed to by R1	5d]	Nibble swap R0	R0:(0–3) is swapped with R0:(4–7). This command puts the entered nibbles in the correct order.
2d –	Dec REG	Decr the reg pointed to by R1	61 a	Enter nibble 'a' hex	Incr R0 by 10 and nibble swap R0
30 0	Enter nibble 0	Incr R0 by 0 and nibble swap R0	62 b	Enter nibble 'b' hex	Incr R0 by 11 and nibble swap R0
31 1	Enter nibble 1	Incr R0 by 1 and nibble swap R0	63 c	Enter nibble 'c' hex	Incr R0 by 12 and nibble swap R0
32 2	Enter nibble 2	Incr R0 by 2 and nibble swap R0	64 d	Enter nibble 'd' hex	Incr R0 by 13 and nibble swap R0
33 3	Enter nibble 3	Incr R0 by 3 and nibble swap R0	65 e	Enter nibble 'e' hex	Incr R0 by 14 and nibble swap R0
34 4	Enter nibble 4	Incr R0 by 4 and nibble swap R0	66 f	Enter nibble 'f' hex	Incr R0 by 15 and nibble swap R0
35 5	Enter nibble 5	Incr R0 by 5 and nibble swap R0	6c l	Load R0 from @R2	Copy contents of reg pointed to by R2 to R0
36 6	Enter nibble 6	Incr R0 by 6 and nibble swap R0	6e n	Next Address	Incr addr reg R1
37 7	Enter nibble 7	Incr R0 by 7 and nibble swap R0	70 p	Print REG value @R1	Print <CR>ASCII,ASCII<CR>
38 8	Enter nibble 8	Incr R0 by 8 and nibble swap R0	73 s	Store R0 to @R1	Copy contents of R0 to reg pointed to by R1
39 9	Enter nibble 9	Incr R0 by 9 and nibble swap R0	75 u	Update RAM pointers	Copy contents of R3,4 to R9,10. Updates sample addr value from sample preload reg.
3c <	Get ctr value from Spock	Shift the current 16 bit ctr value from Spock into R9, R10	78 x	Exchange byte with POD	Transmit byte in POD_TX to POD IO-0. Wait for reply byte on IO-1 and put it in POD_RX then return it to host.
3e >	Program Spock from Registers	Load 5 bytes of data from R3–R7 into Spock. Previous contents of ctr are destroyed	7c	Pass Through byte to POD	Transmit byte in POD_TX to POD IO-0. Connect IO-1 to Serial Out for host.
3f ?	Print Machine ID	Print <CR>CHAR8–CHAR1<CR> where CHAR <i>n</i> is part of a string identifying the type and revision of this device.			
40 @	Load Address Reg	Store R0 into R1. Use to set up reg ptr.			
53 S	Dump Sample RAM (CSV)	Dump lines of 16 Sample RAM values (digital and analog)			

Table ii—The command set for the BitScope virtual machine is a subset of the byte values between 0 and 255. Active commands are confined to the ASCII range from 0 to 127.

cially given the importance of reliably transmitting packets over a serial link.

I decided the BitScope command set should use common printable ASCII commands. Since the assignment of byte codes is arbitrary, any value could mean “enter hex nibble 3,” but obviously 3 is a good choice. The general scheme for allocating byte-code values and their ASCII symbol is:

- numerals—data entry
- operators—manipulation of register values
- lower case—general machine operation
- upper case—major machine functions
- nonprintables—reserved for future commands

An example script for loading R6 with 0x5a is [6]@[5a]s. It may seem obscure, but if you study it, it should make sense. Ultimately, a user interface will debug scripts and writing scripts will only be necessary if a user develops a new mode of operation or drives it directly from a terminal.

All BitScope operations, including wait on trigger, may be interrupted by any serial command. The first part of the software UART ensures that the sample clock is halted. When a serial byte is assembled and echoed, the UART turns on and, once activated, aborts all previous operation. In this sense, BitScope’s command protocol is truly atomic. Each command ends in a halt, if not prematurely aborted. ASCII code 00 is the reset vector, so it can get the PIC’s attention with a <break>.

Inevitably, a VM like this will get enhanced firmware. Microchip has devices that potentially double the number of byte codes implemented. To cope with the potential of other feature sets, ? returns a 32-bit ID code. User-interface software may keep a register of feature sets supported by each byte-code revision.



All instruction bytes are echoed to provide a simple handshake mechanism. And, all instructions are preemptive, so you can always abort the previous command and regain control simply by sending a new command.

SERIAL CONNECTION

While a serial interface may seem a bottleneck for a capture engine that can potentially store 64 KB of data, this is not a problem. Thanks to the Internet and 56k modems, most PCs now have fast, buffered UARTs.

The transmission speed of the BitScope serial link can be scaled to 115 kbps using a fast microcontroller. At this rate, you can transfer enough samples to draw a 640×480 screen—at most 640 bytes—in about 55 ms, or 18 screens per second.

For lower frequency data or simple sine waves, it's necessary to only send a handful of samples to the host and have the user interface do some curve fitting. Small bursts of contiguous sample data may be used to enhance a waveform display to show high-frequency noise.

Logic analyzers don't need to rapidly update their display at all. After a trigger event, the data may stay in the sample RAM and be downloaded only when the host needs it. At 115 kbps, the total contents of a 16-KB buffer can

download in less than 2 s. The user interface may then draw logic state or timing diagrams and manipulate them as necessary.

USER INTERFACE

Don't think shrink-wrapped monolithic Windows software for this design. Think more about the Linux model where the engineering community builds its own tools and can customize them as needs arise.

Because BitScope uses simple ASCII commands, in a pinch, you can use a terminal program and spreadsheet to display waveforms. For complex applications, you need more advanced software based on C, Delphi, or Visual Basic.

A BitScope user interface can run under many possible environments, including Windows, MAC, Unix, WinCE, Palm Pilot, Psion, DOS, or Amiga. Basically, it can work on any machine with a serial port.

No single person could write all that software. Instead, I made the BitScope design open and documented so you could create what you need.

On INK's Web site, you'll find some user-interface software with source listings to start the ball rolling. Via the Internet, you can also find existing programs that already simulate oscilloscopes, logic circuits, and data displays.

DESIGN PHILOSOPHY

A good place to start designing is with a functional specification. For BitScope, the main issue was sample rate. While it seemed clear that a 200-MHz sample rate was out of reach, I could easily get to about 50 MS/s and still be ahead on the price/performance curve.

For the engineer dealing with microcontroller circuits, it's unlikely that frequencies of interest will exceed 20 MHz—at least for the time being. Later on, when 3-V logic becomes more prevalent, that 50-MS/s rate can probably stretch to 100 MS/s in an SMT version of the design.

To make BitScope as useful as possible, I was determined that it should physically stand alone. It needed to be unconstrained to a particular machine or bus standard, and I wanted it to communicate with any computer using the ubiquitous RS-232 interface.

From my experience, the most commonly required features of this type of test equipment are two analog input channels and eight digital logic inputs. Combine those features with a flexible trigger capability, and you get a pretty useful instrument. I set a design goal of about \$100 for the cost of required components, all of which should be readily available.

For the core of BitScope, I selected a PIC16F84 micro tightly coupled with a Lattice 1016 PLD. The PIC controls the serial port and implements a VM architecture. The Lattice counts RAM addresses and waits for a trigger.

These chips are cheap and solid performers. Both are flash-memory based for easy upgrades, and they have excellent entry-level development software. Sample RAM is provided by two 32-KB 15-ns cache memories.

These devices will take the design to 50 MS/s and have the great advantage that about eight of them are perched on every '486 motherboard ever built. That should put their head count at about one billion, so don't tell me you can't find any!

Every DSO must be built around a flash ADC. These chips were exotic until a few years ago when digital manipulation of video became popular.

Now, several companies have devices that can provide 40 MS/s or better for

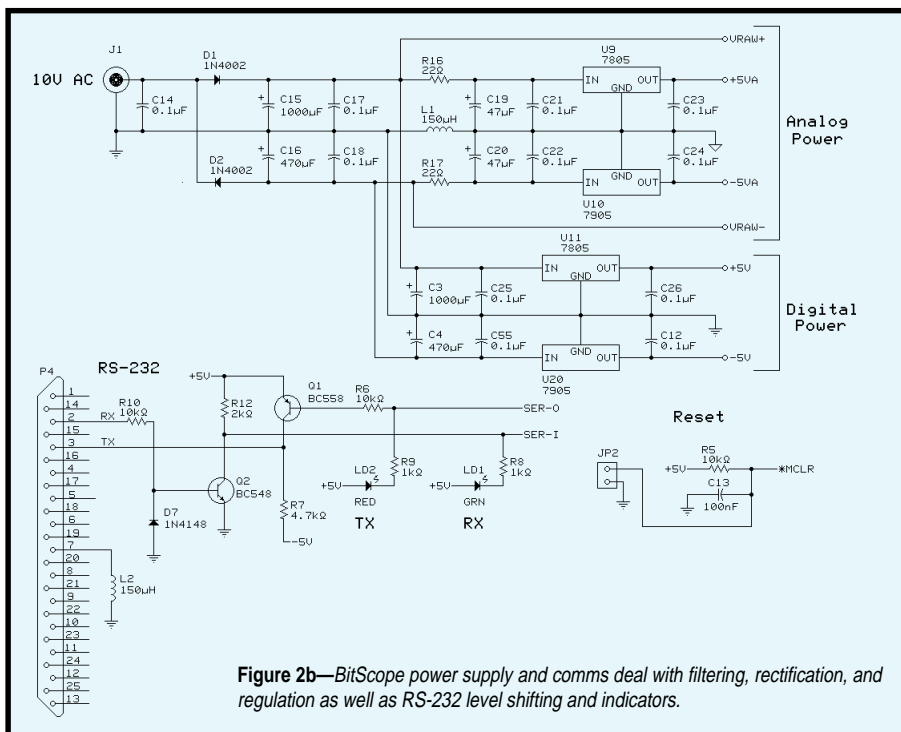


Figure 2b—BitScope power supply and comms deal with filtering, rectification, and regulation as well as RS-232 level shifting and indicators.

less than \$10. Even an older device like Motorola's MC10319P can sample from DC to 25 MS/s and is available in a DIP package.

In fact, I used this device for BitScope. By selecting a 600-mil DIP package, I could accommodate any of the new SMD devices as a plug-in module and avoid the need for multiple PCB versions.

For vertical amplifiers that process analog signals to the ADC, the video industry again provides a solution. Maxim and Analog Devices both have cheap, stable 300-MHz op-amps that make wide-band amplifier design easy.

Using these devices lets the vertical-amplifier bandwidth get close to 100 MHz, matching the input specs on the new flash ADC chips from Analog Devices and TI. For an insight into why we need such wide-bandwidth vertical amplifiers, see the sidebar "Subsampling—Bending Nyquist."



Photo 1—BitScope was prototyped on a two-layer PCB. Notice that the components are arranged to separate analog and digital sections of the circuit.

WALKING THRU SCHEMATICS

Before delving into the schematics, take a look at Figure 1, which overviews the functionality of the BitScope design.

The PIC, the Lattice PLD, and the SRAMs are shown in Figure 2a. These chips are closely coupled to form the sample capture functions at the core of this design.

By using a synchronous tristate clocking circuit, the PIC is able to stop,

start, and preload the Lattice PLD using just a handful of signals. Notice that it's necessary to read in data from the RAM chips one bit at a time because there are no spare eight-bit ports available.

One fundamental rule in mixing analog and digital circuits is to avoid contamination of the analog grounds. Figure 2b shows that great care was taken to isolate the analog and digital sections of this circuit at high frequencies.

Similarly with the RS-232 port, it's best not to allow PC noise to have any path to a test circuit.

Digital test signals and two spare analog signals are shown on Figure 2c connecting to the DB25M pod connector. Logic levels are latched and conditioned ready for storage in the digital sample RAM.

You might guess from the extra signals on the pod that it's not just

eight logic levels in. As well as fused balanced power supplies, there is a digital I/O communication port. Everything you need is there to connect an active, programmable extension module.

Most of the analog conditioning circuits and the flash ADC are shown in Figure 2d. The circuit consists of an amplifier chain driving through a pair of 4:1 analog mux devices.

Modern video op-amps help here. They give you high input impedance, low output impedance, and unity gain stability.

The PIC controls the mux sources that allow implementation of range switching and channel chop functions. To accommodate different ADC chips, there are adjustment pots for both the range

and offset voltages as required by the manufacturers.

Figure 2e shows the final part of the analog conditioning circuit. Channels

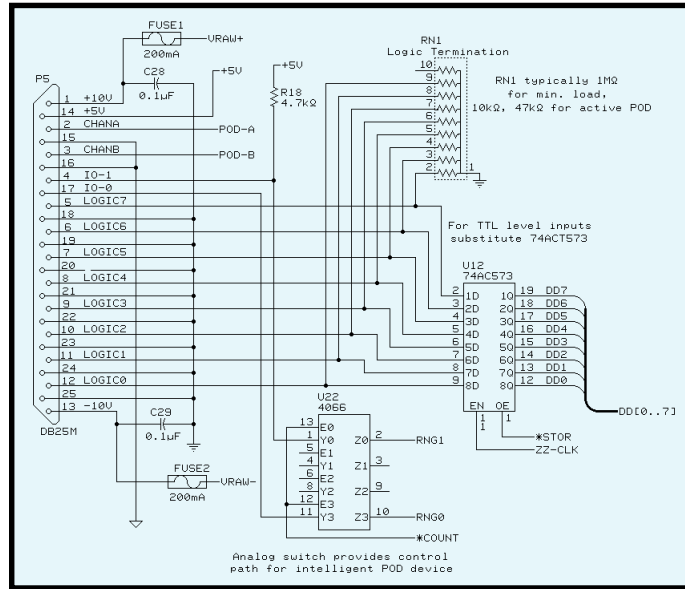


Figure 2c—The BitScope digital capture unit has a logic pod circuit with latching buffer and pod I/O switches.

A and B are standard 1-MB input impedance AC/DC BNC connectors. A classic source follower tree driving a unity gain buffer for each channel completes the vertical-amplifier section.

For engineers who like to measure high frequencies, I added a small 1-GHz prescaler circuit, which includes a switchable 50-Ω terminator hanging off the Channel B input circuit. Note that BitScope has a couple of ways to measure the frequencies of applied signals. I explain the motivation behind this in the sidebar “Subsampling—Bending Nyquist.”

THAT IS LOGICAL, CAPTAIN

PLDs such as the Lattice 1016 can swallow a whole swag of logic func-

Subsampling—Bending Nyquist

In data-acquisition applications, there is often some confusion about the relationship between bandwidth and sample rate. The Nyquist rate of half of the sampling frequency (F_s) is well known to be the maximum frequency that can be captured by periodic sampling at F_s . Given that mathematical constraint, why would we want an instrument that has a bandwidth of 100 MHz and yet samples at a maximum rate of only 50 MS/s? The answer lies with subsampling.

The Nyquist rate applies to continuous time varying signals. In that general case, the highest-frequency component should be less than half of F_s (25 MHz at 50 MS/s) to avoid aliasing. Repetitive waveforms are a different matter. They’re the only high-frequency waveforms you ever see on an analog CRO. The same waveform is redrawn each sweep, and the eye sees a solid trace. Subsampling is similar. You use multiple samples and overlay them to build an image. Providing that your ADC has a wide bandwidth and a small aperture, it is possible to sample a repetitive waveform over many cycles and build up a snapshot of the exact waveform, limited only by the bandwidth of the signal path. This technique, known as subsampling, is just an example of the RF mixer in the digital world.

Subsampling has a few constraints. It isn’t possible to subsample a waveform that’s harmonically related to the sampling frequency. Practically, this means that if the waveform of interest is related to the sample frequency, the sample points always fall at the same relative position

on the waveform and the regions between will forever remain a mystery.

Another concern has to do with resolving the ambiguous period of the subsampled waveform. Let’s say you have a signal of 28 MHz and are sampling at 40 MS/s. In the sample buffer, you’ll see a sequence of values with components at 12 and 68 MHz. How can these be plotted to build up a profile of the original 28-MHz signal? Well, if you can measure the fundamental frequency of the sampled wave, that will imply period. Since you know the sample rate accurately, you can fractionally chop the sample buffer up into segments of n wave periods and then plot them overlaid. You will have traded the freedom for those n waveforms to vary in exchange for n different points on the waveform. It may now be apparent why the BitScope design has provision to measure the frequency of any signal presented to the ADC.

Even if you can’t measure the frequency of a subsampled waveform directly, all is not lost. DSP engineers have some fancy autocorrelation algorithms that can be let loose on a chunk of acquired data to pull a waveform out of meaningless numbers. It is important to note, however, that for resolving single event (such as high-frequency pulses like logic glitches), there is only one solution: oversample by at least a factor of 10. This performance is exclusively in the domain of specialized test equipment using state-of-the-art circuit techniques to resolve samples to 1 ns or better.

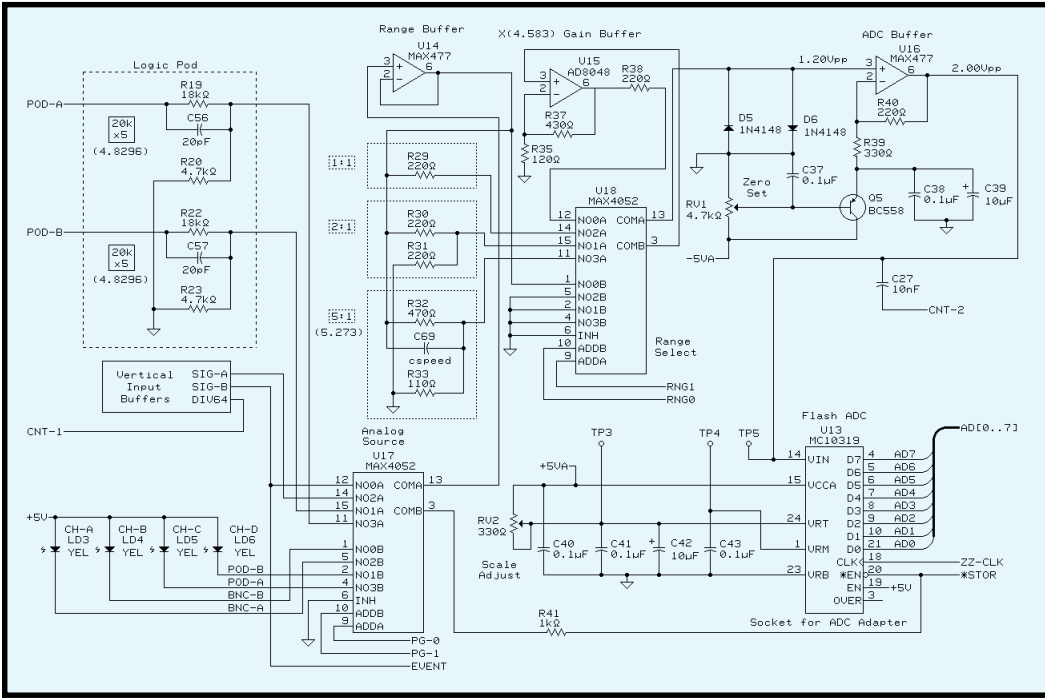


Figure 2d—The BitScope analog capture features the vertical channel muxes, attenuation switch, ADC buffer, and ADC.

wonderful “data lens” you care to attach via the data pod. Because the pod architecture and protocol is open and documented, anyone may design a specialized data lens for BitScope.

VOLTAGE RANGES

The BitScope DSO includes four internal attenuation ranges and four channel inputs. Channel A and B are BNC

tions. In this case, about 18 medium TTL devices with all their wiring disappear into a 44-pin PLCC device.

Radial PLDs like the Lattice are like eight PALs in a circle surrounding a big breadboard. This architecture favors the tight timing requirements of counters and glue logic.

Mostly, this PLD is a 16-bit shift register and counter with a configurable comparator for triggering. The PIC can load a five-byte configuration word that sets the operation of the chip, after which it may be clocked at full speed.

THRU THE LENS MEASURING

The SLR lens-mount system from the photographic world is a great design that has stood the test of time. You start with a camera body with a general-purpose 50-mm lens, and for specialized work, you screw in any of a hundred matching lens types. From fisheye to telescopic, as long as the mounts match, you have a new camera.

I tried to use the same SLR principle in the BitScope design. The device on its own is an extremely useful DSO

and logic analyzer, but it is not everything.

The pod connector provides an electronic lens mount for test equipment. Think of the sample RAM in BitScope as a roll of 35-mm film, and the data you store there may come from either built-in connectors or any weird and

connectors that may have $\times 1$ or $\times 10$ probes connected. Channel C and D (pod inputs) have a fixed attenuator, and possibly, there’s some extra circuitry in the pod.

Table 1 details the range sensitivities. The ranges aren’t nearly as comprehensive as a bench CRO, but it covers

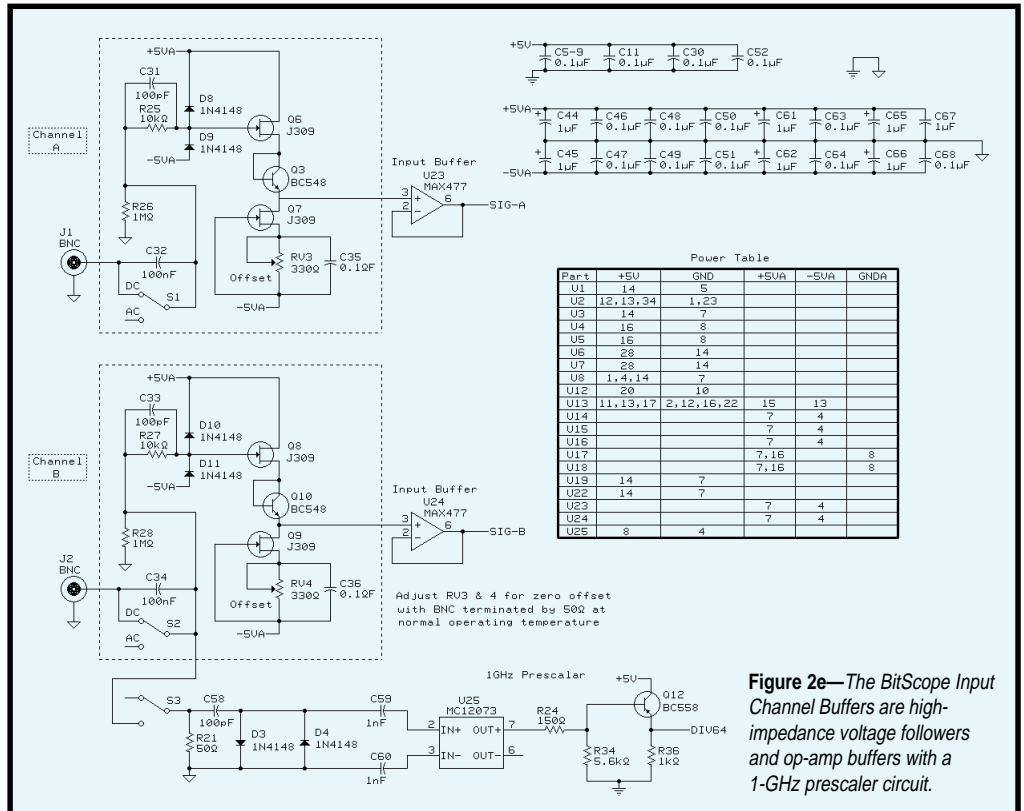


Figure 2e—The BitScope Input Channel Buffers are high-impedance voltage followers and op-amp buffers with a 1-GHz prescaler circuit.

Range	BNCx1	BNCx10	POD
00	±130 mV	±1.30 V	±632 mV
01	±600 mV	±6.00 V	±2.90 V
10	±1.20 V	±12.00 V	±5.80 V
11	±3.16 V	±31.60 V	±15.28 V

Table 1—Here are the BitScope input ranges for an ADC span of 2 V. Resistor attenuators can be found in the schematic.

those most useful to digital and analog circuits. As well, I intended for the pod connector to deal with unusual or high voltage signals by way of an active pod adapter.

It's also possible to alter the gain of some ranges. Since the ADC output is an eight-bit number that ranges from 00 to FF, the final interface just needs to ratiometrically apply this hex value to the voltage range of each stage.

A little thought reveals that for a digital oscilloscope, volts per division and microseconds per division are quite arbitrary notions. Provided that the signal under consideration is within the ADC range and the sample-buffer size, a display can be of any size and grid spacing. Similarly, the notion of y offset

becomes a display function, which has nothing to do with the sample engine.

IN YOUR HANDS

With this design, I hope to have presented a low-cost solution to the engineer's needs for sophisticated test equipment. I have heeded the call for more open designs and liberation from the single-platform juggernaut.

In the coming months, I look forward to hearing from any of you who can think of applications for this device that I haven't even dreamed of. ☐

Norman Jackson is principal hardware design engineer for Discrete Time Systems P/L in Sydney, Australia. He designs DSP-based digital audio systems for use in film and TV postproduction. You may reach Norman at normj@discrete.net.

SOFTWARE

The Circuit Cellar Web site has downloadable software listings, technical documents, programmable binaries, and PCB overlays. Informa-

tion about BitScope is available at www.discrete.net or via bitscope@discrete.net.

SOURCES

PIC16F84

Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 786-7277
www.microchip.com

1016 PLD

Lattice Semiconductor Corp.
(503) 681-0118
Fax: (503) 681-3037
www.latticesemi.com

MC10319P ADC

Motorola SPS
(800) 521-6274
Fax: (602) 897-5725
www.mot-sps.com

Preprogrammed PIC, 1016 PLD, MC10319P ADC, and PCB

Discrete Time Systems
+612 9212 3469
Fax: +612 9212 3470
bitscope@discrete.net
www.discrete.net

Simulating Micro- Controlled Systems

FEATURE ARTICLE

Mike Smith &
Jason Wudkevich

When you're short of cash, you're always looking for less expensive ways to build embedded systems. If you're like Mike and Jason, you'll use virtual devices to test and debug your system and its peripherals.



Are you wanting to use some hardware you don't have (or can't afford)? Many useful embedded-system ideas wither on the vine at the development stage for lack of resources.

Virtual devices, however, provide an avenue to the leading edge in the embedded-system market. They let you test how the embedded system controls all its peripherals, even if every peripheral isn't physically available.

In recent years, at least two tutorials on generating such hardware emulators have appeared—Mike's "Developing a Virtual Hardware Device" (*INK* 64) and an article by Larry Mittag [1].

These tutorials produced usable virtual devices. But, the concepts now appear outdated when compared to devices that can be developed with newer commercial virtual-device construction packages like the micro-processor simulators and debugging environments from Software Development Systems (SDS).

SDS's GUI packages can be used to debug embedded systems based on the PowerPC and 68k processor families. ColdFire 5102 and Z80 processor support is present on the most recent release (V.7.11).

The Single Step Peripheral API development diskette option in SDS's

full 68k development system enables a software engineer to quickly create realistic and sophisticated virtual devices. The sophistication available is easiest to demonstrate using an SDS UART virtual device which can be controlled as a peripheral from within any SDS microprocessor simulator.

The lowest two levels of virtual devices that can be generated with the Peripheral API interface roughly correspond to the capabilities of the virtual devices developed by Smith and Mittag [1].

Basic read and write operations change the values of the virtual UART registers. Information from a file or a keyboard dynamically changes the device registers to represent the UART's interaction with the outside world. Writing to a virtual UART register places the transmitted characters into a GUI window or file.

The API's third sophistication level enables simulation of time-dependent hardware register changes in a straightforward fashion not possible with the Smith and Mittag approaches. You can use a time specification file to produce UART-driven interrupts and update registers (e.g., timer registers) on-the-fly.

We particularly liked the Peripheral API system's ability to handle the fourth level of virtual-device sophistication.

Imagine running an SDS processor simulation on a PC. At the flick of a software switch, the virtual UART takes control of the PC's UART to send information into the real world from within the simulation. You can emulate communication with a networked embedded system or control a physical device over a real serial interface.

All the functionality required to produce a sophisticated virtual device is in the C++ classes provided with the SDS Peripheral Interface. So, why are we writing this article?

Unfortunately, this clever development interface is poorly documented. And to complicate matters, there was a hidden defect in the concepts for developing virtual devices using the approaches of both Smith and Mittag which makes it difficult to transfer the knowledge gained from building

these devices into constructing an SDS virtual device.

DEVICE CONCEPT

Suppose you see a market for the McVASH—a micro-processor-controlled, voice-activated shower head. It synchronizes the water pulse rate to the voice of a person singing in the shower and even offers a prerecorded beat of solemn music stored in EEPROM for those Monday morning blues.

For safety, the microprocessor is solar powered and communicates to the shower head over an IR link driven from a serial port. A small water impeller provides power for the shower head.

Before seeking venture capital, you want to get a feel for the associated software size and required hardware control. A virtual device seems to fit the bill.

Table 1 shows a simplified programmer's model of the hardware registers. To make the problem a little more realistic, not all the registers have the same byte size. Offsets from the base address (0x20000) are given.

OPERATIONAL DEMONSTRATION

This virtual device should simulate the operations expressed by the pseudocode in Listing 1. In the code, the device is reset, and the waterstream turns on and off in synchrony with the peaks and valleys of the notes in the bather's voice.

Most embedded-processor development environments on the market can directly debug code using information from the original C source file. So, you may be tempted to build a virtual device using a direct C-code approach.

Register	Size	Offset from Base Address	Action
Water Valve Control	8 bit	0x00	Turns waterstream on and off
Timer	32 bit	0x04	Indicates number of milliseconds since system startup
Period	16 bit	0x08	Gives period of bather's current vocal rendition
Infrared Receive	8 bit	0x0C	Permits communication between solar-powered microprocessor (on bathroom window) and water-impeller-powered shower head
Infrared Transmit	8 bit	0x10	(Same)
Shampoo	8 bit	0x14	Puts shampoo and conditioner into waterstream at the command of voice-recognition circuit (patent pending)
DSP 1	32 bit	0x18	Example of available DSP register
DSP 2	32 bit	0x1C	Example of available DSP register

Table 1—The McVASH device registers offer a wide range of functionality and bit sizes.

Listing 2a illustrates this approach for function `ResetMcVASH()` which initializes the virtual device's 8- and 32-bit registers. We represent the virtual device registers using the array `device_reg[]`.

All read and write operations are handled via `#define` statements. This technique is more natural than directly manipulating `device_reg[]`.

But, this approach only simulates the most basic reads and writes. For example, problems occur with simulating a simple `while` loop. This loop never exits since `TIMER_REGISTER` never changes! These operations must be translated by the tester as in Listing 2b.

ADDING SOPHISTICATION

Listing 3 shows a more sophisticated virtual device that handles time-dependent register operations with a realistic addressing mode. With the peripheral access function—`paccess()`—this virtual device can differentiate between memory-mapped accesses to itself or to other system devices.

Managing the required time-dependent register changes (e.g., incrementing `TIMER_REG`) as part of `paccess()` is straightforward. The different sizes of

memory operations (byte, word, and long word) are also handled in a manner reflecting the real device registers' structure.

Such an implementation works with any processor—simulated or real—and any debugging system that handles C code. Changing the `#define` statements into macros that place parameters on the stack before calling subroutines lets the virtual device be manipulated directly from assembly-language code.

This simple approach is usable with code running on an evaluation board or in a processor simulator, providing a device with the sophistication level of the Mittag virtual device [1].

However, the programmer is still open to many fundamental errors if the simulation code has significant length. How long before the developer attempts to access the virtual-device registers using the same format as any other C variable (e.g., `*device_reg = value`) when a contrived form (e.g., `WriteByte(device_reg, value)`) is needed?

A successful simulation implies that accessing any virtual-device register should occur transparently!

VIRTUAL-DEVICE COMPLEXITY

Minimal virtual-device sophistication should let us code the use of the device directly in C. Direct manipulation of the virtual device in assembly code should also be possible—at least at the level of the 68k section in Listing 4.

This higher level of sophistication is available with the Smith virtual device, which runs by producing exceptions produced when the processor accesses the virtual-device registers.

Listing 1—This pseudocode for the McVASH device demonstrates initializing the device registers and controlling the water flow according to period of the note being sung.

```
long int initial_time
ResetMcVASHdevice() // set device regs to known value
initial_time = *timer_register // Synchronize water with voice
while (initial_time + *voice_register > *timer_register)
    *transmit_reg = ON;
initial_time = *timer_register // Synchronize water with voice
while (initial_time + *voice_register > *timer_register)
    *transmit_reg = OFF;
```

REGISTER ACCESSES

It's necessary for the processor, rather than the programmer, to recognize that virtual-device register accesses are occurring.

Table 2 shows a simplified (but typical) memory map for an embedded processor system running in real life or from within a simulation. Some address ranges are associated with physical memory chips, while others aren't.

Memory accesses to the RAM associated with the user program and variable space (0x30000-0x3FFFF) are handled properly by the processor. But, memory accesses in the 0x20000-0x2FFFF range attempt to read nonexistent memory locations, triggering a BUS ERROR exception and a jump to the system exception handler.

You can use this exception-handling capability to create a more sophisticated virtual device that manages standard coding practices in a user-friendly way. The new virtual device captures all BUS ERRORS produced by memory access in the 0x20000-0x2001F range.

Listing 2a—The `ResetMcVASH()` function is simulated with an unsophisticated virtual device. However, even the simplest `while` loop (b) from Listing 1 will fail.

```
a)
long int device_reg[8];    // Array to represent device registers
#define WATER_REG 0       // Pointers to device registers
#define TIMER_REG 1       // Functions handle op on registers
#define ReadByte(reg_name) {device_reg[reg_name] & 0xFF}
#define WriteByte(reg_name, value) {device_reg[reg_name] = value&0xFF}
#define ReadLong(reg_name){device_reg[reg_name]}
#define WriteLong(reg_name, value){device_reg[reg_name] = value}

void ResetMcVASH(void)    // Reset device registers
{
    WriteByte(WATER_REG, 0); // *water_reg = 0;
    WriteLong(TIMER_REG, 0); // *timer_reg = 0;
}

b)
while (initial_time + ReadWord(VOICE_REGISTER) >
        ReadLong(TIMER_REGISTER))
    WriteLong(TRANSMIT_REG, 0N);
```

In *INK* 64, Mike explains how a virtual device can run on RISC (AMD 29200) and CISC (Motorola 68332) microcontroller evaluation boards. Obviously, such implementations require detailed processor-dependent knowledge, but Listing 5 captures the main concepts.

RISC VS. CISC

Developing a sophisticated virtual device using the bus-exception approach is straightforward for the simple load/store architecture of a RISC processor (e.g., the AMD 29k family). Values needed during exception handling

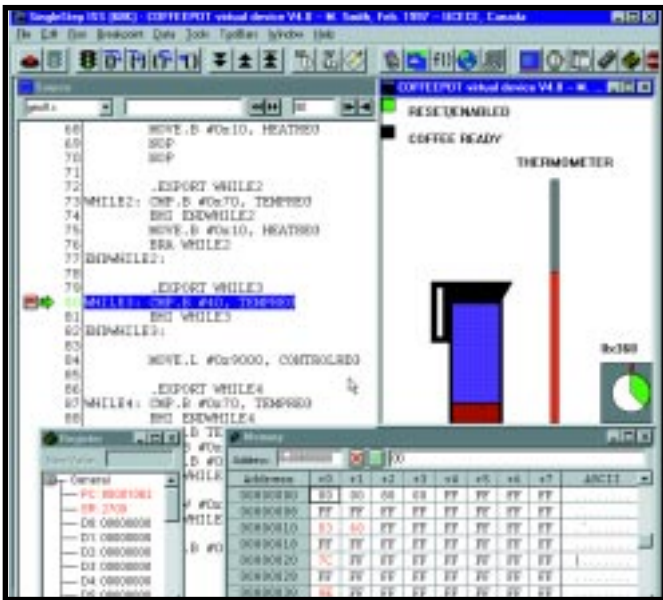


Photo 1—Water levels, water and heater temperature, and an elapsed time clock are all animated. It was no small feat to develop a virtual device with this level of animation.

were available from dedicated registers rather than stored on an external memory stack.

The regular format of the 29k RISC instructions, together with special “indirect register” registers, enabled simple recognition of the source and destination registers involved in the virtual-device operation. We generated virtual devices that ran directly on the AMD evaluation boards and within AMD debugging simulators [2].

Generating a virtual device for a CISC processor (e.g., Motorola 68k) using the bus-exception approach was more problematic. One major difficulty was the different stack frames for BUS ERROR exceptions across a processor family (compare those from the Motorola 68332 and 68020 processors) and for a given processor (e.g., Motorola 68020).

Also, some processors move onto the next instruction after an exception is handled. Others expect the addresses on the stack frame and data values to be corrected so the instruction execution can continue on return from the exception.

Such complexity means several BUS ERROR exceptions can be triggered in one instruction. An example is the 68k instruction `MOVE.L OFFSET1(A0), OFFSET2(A0)`.

Other problems arise with any processor pipeline occurring when the exception is forced. Take the following instructions:

```
MOVE.L D0, OFFSET1(A0)
MOVEQ.L #5, D0
```

The second instruction will have completed loading register D0 before the processor handles the BUS ERROR associated with storing the original D0 register value from the first instruction.

In theory, these different cases must be handled in the virtual-device code. In principle, you must construct a complete CISC-processor simulator as part of the exception handler. But in practice, the situation wasn’t too problematic for virtual-device write operations.

Most C compilers don’t produce a wide range of instructions when accessing a fixed memory location that may represent the absolute address associated with a peripheral. Since such instructions aren’t produced by the compiler, you don’t need a virtual device that handles them.

By contrast, even the simplest CISC reads on the virtual device proved troublesome. Without significant decoding, it’s hard to determine the instruction’s destination register.

Despite this fact, we developed some useful virtual devices that only respond to reads sending values to one specific register, as in:

```
MOVE.L (address_reg), D0
MOVE.L OFFSET(address_reg), D0
```

Of course, this approach works if you write assembler code where you

can specify the D0 register. Surprisingly, it also typically works with automated code generation since typical C compilers reuse D0 as a temporary or volatile register.

EXCEPTION-HANDLER PROBLEMS

The exception-handling approach to virtual-device development works user-transparently. We've produced several virtual devices that work with Motorola 68020 and 68322 CISC processors running in the SDS simulation and on evaluation boards.

However, some serious problems remain. For one, you need to develop different virtual-device code for each processor your company plans to use.

Also given how time-dependent register operations are implemented, no register changes occur unless the device is being accessed by the programmer's code. So, it's difficult to ensure that the timing of device operations

has a proper relationship to the processor clock.

It's also hard to debug virtual devices unless you know where the array representing device-register values is located. Direct access to the "physical" device-register locations doesn't work because many debuggers modify the processor's exception handling, blocking virtual-device operation.

Complicating the issue is the fact that some processor simulators don't properly handle all processor exceptions. The appropriate exception stack frame is constructed, but you can't recover from the interrupted instruction. So, operation becomes more complicated because you need to modify the stack frame to represent an exception handled by the debugging environment.

Finally, an unsophisticated virtual device can't simulate real-time operations. An instruction that should interact with the real peripheral over

several processor clock cycles takes several *thousand* cycles due to the exception-handler code needed to cause the virtual device to function.

DLL VIRTUAL DEVICE ADVANTAGES

These problems pale in comparison with the fact that you can never be completely sure that you, or the compiler, haven't used processor instructions your virtual device can't handle. This snag becomes more likely as the project code size and optimization level increase.

A virtual device generated using the SDS Peripheral API offers a good solution. This device takes the form of a DLL, so one virtual device can be used in conjunction with simulators from any processor family.

The DLL acts in conjunction with the SDS processor simulation rather than being parachuted on top of it. So, access to the device registers occurs user-transparently.

Time-dependent operations, including complex concepts like triggering the virtual-device interrupts at specific times, are easily linked to the processor clock because the DLL becomes part of the processor simulation.

Since operations on the virtual-device registers are handled identically to other memory operations, any instruction sequence the compiler can generate from your product code is guaranteed to work.

The virtual-device DLL is developed using C++ peripheral classes provided by SDS. Through the DLL interface, the virtual device can access any physical peripheral belonging to the PC the simulation is running on.

A UART virtual-device demonstration illustrates the power of this DLL approach. Connect the UART to the processor simulation at memory location 0x20000 by adding mem 0x20000 periph=c:\sds\periph\xuart.dll to the simulation start-up file.

Random- or fixed-period timing for UART character transmission is achieved by changing options or activating a timing file. Incoming characters can be made to produce interrupt events. Another command causes the UART virtual device to take over an actual serial port for more in-depth simulation and debugging.

Listing 3—A more sophisticated virtual device can handle memory-mapped I/O and time-dependent register operations in a realistic manner.

```
char device_reg[32];           // Array to represent device regs
// McVASH base address is 0x20000
#define BASE_ADDRESS 0x20000
#define WATER_REG 0x20000     // Device reg addresses
#define TIMER_REG 0x20004
// Functions to handle operations on registers
#define ReadByte(address) {paccess(address, NULL, READ, BYTE)}
#define WriteByte(address, value) {paccess(address, value, WRITE, BYTE)}
#define ReadLong(address) {paccess(address, NULL, READ, LONG)}
#define WriteLong(address, value) {paccess(address, value, WRITE, LONG)}

long int paccess(long address, long data, short operation, short
                data_size){
// Determine if manipulating virtual device or system component
if (address < BASE_ADDRESS) Other_Memory_Operation();
else {
    accessed = address - BASE_ADDRESS;
    if (operation == READ)
        switch(data_size) {
            case BYTE: value = device_reg[accessed]; break;
            case WORD: value = device_reg[accessed];
                value = (value * 0x100) + device_reg[accessed +1]; break;
            ... }
        else if (operation == WRITE)
            ...
            Handle_Time_Dependent_Register_Operations();
            return(value);
        }
} // Return reg value as required

void Handle_Time_Dependent_Register_Operations(void)
{
    device_register[TIMER_REGISTER - BASE_ADDRESS]++;
    // Increment device TIMER_REGISTER
    ...
}
```

FIRST ATTEMPT

The Single Step Peripheral API offers generic UART and timer support, but it's easy to expand peripheral support by creating DLLs. We followed SDS's advice of "modify a copy of either DLL" provided as examples, and we deleted all code associated with UART capabilities we deemed unnecessary.

However, our limited knowledge of Windows and DLL interfacing, combined with the absence of a generic template, meant this approach was incorrect for producing a working virtual device.

THE SECOND TIME AROUND

Our second prototype attempt was more successful. We modified the API routine `Iperipheral::paccess()` (see Listing 6), which is called when the peripheral's memory is accessed.

The SDS API peripheral virtual device operates much like the Smith and Mittag devices. Details of each memory operation are passed to `paccess()` and used to manipulate device registers.

Note that the DLL interface automatically provides all the information that needs to be laboriously decoded from the Smith device's exception stack or hard coded into the Mittag device. Thus, the virtual device can be coded processor independently.

We ported the existing C code in an exception handler from a Smith-designed virtual device into `paccess()`. The prototype device was quickly up and running, responding to code written in C and assembler.

The device ran well, but only if we didn't use any debugging tools to check it—a sort of Heisenberg Uncertainty Principle for microprocessors! For example, using the MEMORY window to examine the virtual registers slowed the simulation and caused crazy operations in the virtual device.

After spending weeks attempting to get around the problem, we discovered a defect in both the Smith and Mittag virtual-device concepts. These virtual devices change their internal state register values whenever a register is accessed. If no access occurs, the device doesn't operate.

Listing 4—A more sophisticated device should be capable of handling *while* loops in assembly code using a variety of addressing modes.

```
MOVEA.L #BASEADDRESS, A0 // Set pointer to virtual device
MOVE.L TIMER(A0), D2 // initial_time_D2 = *timer_reg
WHILE:
MOVE.W VOICE(A0), D3 // while (
EXT.L D3
ADD.L D2, D3 // initial_time + *voice_reg
CMP.L TIMER(A0), D3
BLT END_WHILE // > *timer_reg
MOVE.B #OFF, TRANSMIT(A0) // *transmit_reg = OFF;
BRA WHILE
END_WHILE:
```

They can't distinguish between a user program and a debugging tool accessing the registers. Viewing the device registers through the MEMORY window unintentionally changed the register values.

Of course, the fix was straightforward once we found the problem. The API interface function `Iperipheral::pkick()` enables regular, random, or file-driven updates of the virtual-device registers.

`pkick()` is used in conjunction with an internal DLL variable `pstate` that controls when the next device internal-state change occurs. Debugging-tool access doesn't change `pstate`, but normal user access does. The problem disappears when the two types of memory access are distinguishable.

`pstate` is also used in conjunction with `Iperipheral::irqquack`, which performs the hardware interrupt acknowledgment cycle for interrupts generated by the virtual device.

ADDING A WINDOWS GUI

We added a graphics interface to display properties of the peripherals (e.g. temperature gauges, elapsed time, etc.). Photo 1 shows the GUI for the COFFEEPOT virtual device.

Since the current SDS documentation doesn't explain how to add basic graphics to a device, we recommend Charles Petzold's book [3]. He leaves the wimpy, specialized, integrated-development environment behind and uses makefiles to generate Windows programs in C using a Visual C++ compiler directly from the command line.

With Petzold's help, we developed an operating GUI prototype with all the graphics routines necessary to display the virtual device's behavior. These routines were placed in a C source file and linked to the virtual-device code.

Unfortunately, no documentation reminded us how impossible it is to

Listing 5—This pseudocode demonstrates the basic operation of a virtual device handled through the generation of memory BUS ERRORS on device register accesses.

```
BUSERROR_HANDLER:
Save Volatile Registers to stack to prepare to jump into C code
Access Information from Exception Stack Frame {
Find Exception Address information
if (Memory Access outside of device range)
Do Normal Exception Handling( );
else {
Determine from stack frame if need READ or WRITE operation
Determine WRITE value (from the stack frame) if appropriate
Determine manipulated DATA size from stack frame
Launch paccess routine(address, value, operation, data_size)
}
}
Recover Volatile Registers from the stack
Clean and/or transform the Exception Stack Frame
Perform actions informing processor of completed inst
Return From Exception
```

link Windows C code to API C++ code without some background on how both languages are implemented at a low level.

We were unable to persuade `pkick()` to access our GUI routines. The graphics routines it requested were perceived at link time as being different than the actual routines present in the working interface! It wasn't a question of hidden characters in the function names. Both code sets appeared to recognize the graphics function prototypes from a common .h include file.

Fortunately, a colleague reminded us of name mangling. Name mangling is a technique used in C++ to permit overloading of function names.

Function overloading lets you make calls to both `void Some_Function(int, int)` and `void Some_Function(int)`. C doesn't let you use the same function name under different circumstances.

When these functions are translated with a C++ compiler to the as-

0x60000	Interrupt vectors and exception code
0x50000	Stack
0x40000	Empty
0x30000	User program and variable space
0x20000	Empty; Future site of the McVASH device
0x10000	System working space
0x00000	Start-up ROM code

Table 2—This simplified memory map for an embedded microprocessor system shows filled and empty memory ranges. The future location for the McVASH device, with a base address of 0x20000, is also shown.

sembler-code level, the first function name is modified (i.e., mangled) to the equivalent of `_Some_Function_void_int_int`, and the second name becomes `_Some_Function_void_int`.

Thus, the two similar function names are distinguished at the assembler level. With a plain C compiler, they both generate `_Some_Function` and are not distinguishable.

Something along these lines explained what was occurring at link time with the combined GUI interface and API code. However, it wasn't obvious how the problem was caused as we weren't using function overloading.

Even though we only used one compiler, we had somehow compiled one code set with a Visual C++ compiler and the other with another compiler. No wonder the object files wouldn't link!

We had stored the API source code with the extension `.cpp` and the Petzold GUI code with `.c`. But, the Microsoft compiler

acts as a C++ compiler for files with the first extension and as a C compiler for the second type.

The problem occurs when the function names are mangled as C++ functions when they are called from the virtual-device code and are being treated as nonmangled C functions in the GUI. Modifying all file extensions to `.cpp` cleared the problem.

VISUAL C++ V.4.0

The SDS user manual describes various Microsoft compiler options needed to create a peripheral using the DLL approach. However, those options are for an earlier Visual C++ compiler than V.4.0.

We were successful using these options when compiling the V.7.02 peripheral classes:

```
/nologo
/W3
/GX
/D "WIN32"
/D "_WINDOWS"
/D "MSDOS"
/D "MSWINDOWS"
```

But, we had our share of difficulties. When adding the graphics interface to the SDS peripheral code, you must add a WM_PAINT case statement to DWORD FPASCAL UARTWndProc() which controls the virtual-device operation (original file `xuartdev.cpp`).

This option is activated when your graphics window is resized or uncovered. It should cause the repainting of the GUI for the virtual device.

You can't animate graphics at a regular interval using a built-in Window's timer (e.g., WP_TIMER inside the `WndProc()` loop). When running inside SDS processor simulations, this timer has a very low priority. Normally, it doesn't generate any interrupts for controlling animation features.

We brute-forced over this problem by updating the graphics screen any-time changes might occur. This solution decreased the simulation speed and caused significant screen flicker.

It would be useful to pause the animation at regular, controlled intervals, but a Window's timer doesn't work here, either. One workable approach is the SDS Breakpoint window debugging tool, which provides breakpoints that briefly stop the simulation. It's straightforward to set a "pause and resume" breakpoint at a point in a loop where the embedded system's code is continually accessing the device registers.

We also had to add a series of global variables to pass the virtual-device register values from the SDS virtual device to the graphics functions. A similar problem occurred with the pointers to the various graphics windows generated during the virtual-device initialization but used by our peripheral GUI screen.

Listing 6—A virtual device developed using the SDS Peripheral API is driven by code that shares a common heritage with the Smith and Mittag [1] devices.

```
IFCFUNCTIONDEF(int, McVASH::paccess)
(PSTATE *pstate, ulong offset, uchar *ptr, int size, int write){
  rulong rid; // Register being accessed
  // Access the memory-mapped registers one byte at a time.
  while (size) { // Manipulate value of size bytes
    rid = offset;
    offset++; // Advance offset to next byte
    if (write) switch (rid) { // Is WRITE operation requested?
      // Manipulate 8-bit virtual device register—rvalve
      case VALVE: rvalve = *ptr++; // ptr is DLL-related parameter
        break;
      // Manipulate 32-bit virtual device register—rcontrol
      case TIMER: rcontrol=(rcontrol&0x00FFFFFF) + ((*ptr++)<<24);
        break;
      case (TIMER+1): rcontrol=(rcontrol&0xFF00FFFF)+((*ptr++)<<16);
        break;
      ...
      default: ptr++; // Ignore value written otherwise
        break; }
    else /* read */ switch ( rid ) {
      // Manipulating an 8-bit virtual device register
      case VALVE: *ptr++ = rvalve;
        break;
      ...
    }
    Handle_Time_Dependent_Register_Operations(); }
  return(1); }
```

FUTURE PLANS

Although its documentation was poor, we're impressed with the capabilities of the SDS Peripheral API interface option. It's useful to compare other embedded-system debuggers to it.

April 1998 saw the finish of a new fourth-year computer-engineering team project course. Two projects involved the development of an Ethernet virtual device using the SDS peripheral API interface [4].

We created Ethernet virtual devices capable of taking over the PC's Ethernet device. We'll be using the Ethernet DLL in future classes.

As for the real McVASH device, well, maybe we'll enter it in the year-2000 INK design contest! 📧

Thanks to Geoff Revill, Louis Meadows, and Joeseeph Fao for their donation and support of the SDS development environments. Many thanks to Bob Davidson of Microsoft for a significant donation from Microsoft to the ECE Dept. at the University of Calgary. Thanks to NSERC and the University of Calgary for their continued support.

Mike Smith is a professor of electrical and computer engineering at the University of Calgary, Canada. He teaches

about CISC, RISC and DSP processors and researches many aspects of image and signal processing. You may reach him via smith@enel.ucalgary.ca.

Jason Wudkevich graduated from the University of Calgary and now works in the area of telecommunications software with Nortel, Calgary.

REFERENCES

- [1] L. Mittag, "Device Drivers for Non-Existent Devices," *Embedded Systems Programming*, **9:8**, 1996.
- [2] M. Smith, "The Laboratory Companion," www.mhhe.com/compsci/compscience/labcompan and www.enel.ucalgary.ca/people/smith/mcgrawhill/index.htm.
- [3] C. Petzold, *Programming Windows 95—The definite guide to the Windows 95 API*, Microsoft Press, Seattle, WA, 1996.
- [4] www.enel.ucalgary.ca/People/Smith/new_web/projcour/index.htm.

SOURCE

Microprocessor simulator
Software Development Systems
(630) 368-0400
Fax: (630) 990-4641
www.sdsi.com



32 Nouveau PC
edited by Harv Weiner

36 Interprocess Communication
Using Anonymous Pipes
Ernie Deel

Photo courtesy of Paradigm

42 FAT32
File System for Data-
Intensive Applications
Edward Steinfeld

49 Real-Time PC
Data Acquisition
Ingo Cyliax

55 Applied PCs
Debugging & the Net186
Fred Eady

PALM-SIZE MULTIMEDIA EMBEDDED PC

The **PCM-4825** is an ultra-compact, '486-processor-based SBC with onboard 32-bit SVGA and LCD interfaces, as well as a 16-bit audio controller. Its small size (about the size of a 3.5" hard disk drive) makes it ideal for limited-space embedded applications such as car PCs, GPS devices, and portable instruments.

The PCM-4825 includes an onboard CPU (AMD's 5x86-133) and 16-bit Sound Blaster Pro-compatible audio interface. An onboard socket provides for flash-disk expansion up to 72 MB using M-System's DiskOnChip2000 flash-disk module. A PC/104 interface is also provided. The PCM-4825 is based on the PC/AT architecture, so it is compatible with most off-the-shelf software.

An optional case, measuring 7.48" x 4.49" x 1.56", allows for flexible expansion. The PCM-4825L is identical to the PCM-4825 except that it has no onboard audio.

The PCM-4925 sells for **\$332** in 100-piece quantities.



Advantech Co., Ltd.
(800) 800-6889 • (408) 330-9399
Fax: (408) 330-9393 • www.advantech.com

DEBUGGING SOFTWARE

The **OSE Illuminator** is a toolsuite for debugging and analyzing real-time embedded systems applications during runtime. Integrating OSE's Evact Handler, System Browser, and Memory Profiler, Illuminator provides application-level debugging for complex applications such as high-availability systems in telecommunications or safety-critical fault-tolerant systems in industrial process control and medical instrumentation.

Operating at a higher abstraction level than source-code debugging, Illuminator's application-level debugger examines events in the application (e.g., messages sent between processes). Additionally, Illuminator offers clear overviews of a system's run-time characteristics, including memory usage.

The Evact Handler enables the user to combine a specific event with a specific action, and the action is performed when the event occurs. Any combination of events (e.g., sent message, switched context, created or terminated process) and actions (e.g., monitor, trace, catch) can be used, so the user can track a complicated sequence of events in a system and collect detailed information about the specific event of interest.

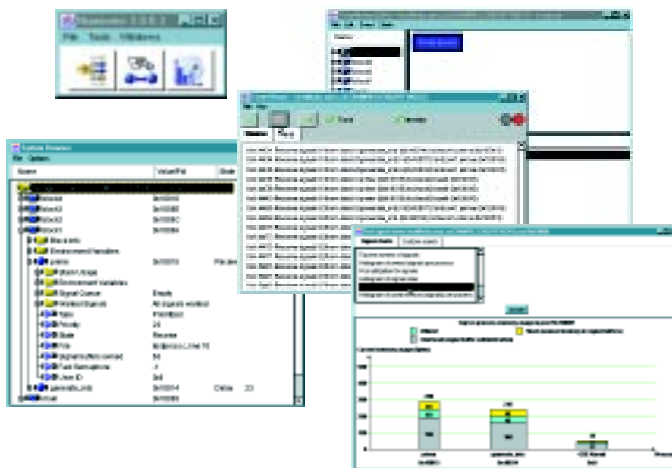
The System Browser gives a clear overview of an entire distributed system and allows the user to easily spot errors that lead to starvation or data overflow.

The Memory Profiler lets users analyze and debug run-time memory usage in an OSE system even if the system uses many distributed processors. The Memory Profiler presents all available targets on a distributed system's network and enables the user to

perform a number of functions that range from fine-tuning memory use to locating major programming and design errors.

The OSE Illuminator starts at **\$2000**, with versions available for Windows NT, Windows 95, and Solaris, as well as any system with a Java virtual machine. The OSE RTOS and related tools are also available for the Motorola 68k and PowerPC families.

Enea OSE Systems, Inc.
(214) 346-9339 • Fax: (214) 346-9344
www.enea.com



Nouveau PC

edited by Harv Weiner

SINGLE-BOARD COMPUTER

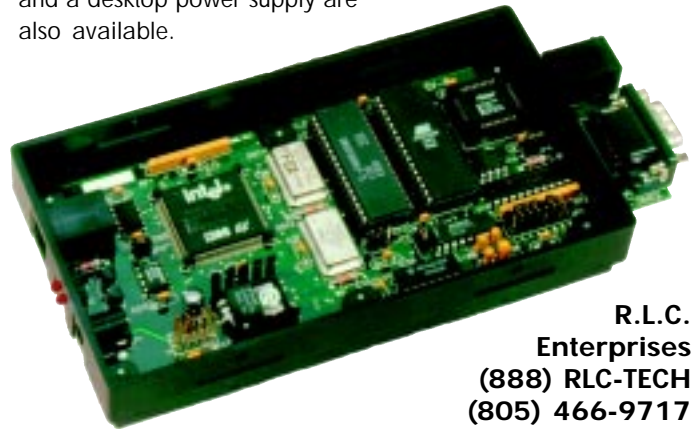
The **Micro-Plus** '188EB and '386EX CPU cards are designed to be the engine of a typical high-performance embedded system. Their small size, low power, and high-performance operation make them ideal for compact, low-power, rugged applications such as portable equipment, factory automation, or controlling a remote field site.

The Micro-Plus is expandable like a bus system via the two onboard RS-485 network connectors and stackable ABS frame. Using network expansion offers a more compact, cost-effective solution to many embedded applications.

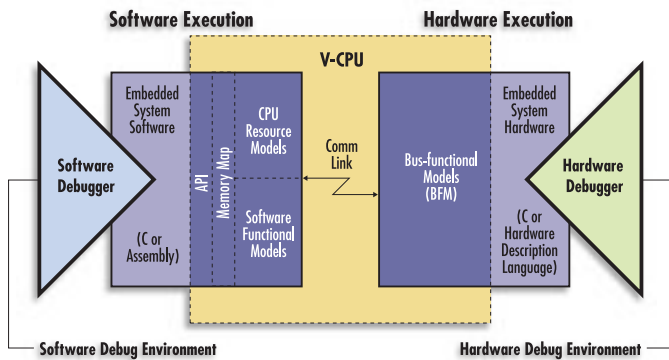
Micro-Plus offers a choice of a '188EB 16-bit or '386EX 32-bit CPU chip. Onboard memory includes up to 512-KB battery-backed SRAM and 512-KB flash memory (new boot-block). Additional onboard functions include interrupt controller, three 16-bit counter/timers, two RS-232/485 ports, watchdog timer, software-selectable jumpers, and LEDs. The two RS-485 network expansion ports can interface directly to a wide variety of off-the-shelf network I/O expansion modules for analog I/O, digital I/O, serial I/O, parallel I/O, opto I/O, relay modules, relay drivers, motion control, and LCDs.

Software development using Microsoft and Borland C/C++ and/or assembly language is supported, as is a full symbolic target debugger that can be used with the flash memories. A resident BIOS is factory programmed into the protected area of the flash memories to enable their downloading and programming. Embedded OS device drivers, program downloader, start-up code, demo programs for the onboard functions, and quick-start procedures are provided as well.

The Micro-Plus SBCs are priced at **\$249** for the '188EB version and **\$312** for the '386EX version. SRAM, flash memory, lithium battery, real-time clock, and extended temperature ranges are all standard features. Connector kits, mounting hardware, and a desktop power supply are also available.



**R.L.C.
Enterprises**
(888) RLC-TECH
(805) 466-9717
Fax: (805) 466-9736
www.rlc.com



VIRTUAL CPU SOFTWARE

V-CPU 3.0 provides a virtual environment for designers to debug hardware and software early in the design process by enabling simulation of the full system prior to silicon.

V-CPU replaces the target processor with a bus functional model (BFM) that can be an existing Verilog, VHDL, or C model of the processor bus interface. By attaching the V-CPU bus-independent driver to the BFM and linking the embedded software development environment with the V-CPU API library, a virtual environment is created that gives high-performance system-level simulation with full software and hardware debug capability.

V-CPU 3.0 offers added logic simulation support for Verilog VCS from Synopsys on the Windows NT platform. Added features include improved synchronization between the software application and hardware design via asynchronous interrupt handling and support for Denali memory models.

V-CPU's memory-map capability, coupled with its implicit access technology, makes it an extremely flexible coverification tool that is nonintrusive for the designer. After the design is partitioned, V-CPU's memory map lets the user configure to the hardware or the software side, enabling the designer to control performance and accuracy.

The memory map also supports configuring C and C++ based software functional models (SFMs). An SFM can model external environments and peripheral devices or replace incomplete portions of the hardware design.

V-CPU has solution support for a wide range of embedded processors and cores, including various MIPS, Intel, Motorola, and ARM processor models. Single-quantity floating licenses for the V-CPU 3.0 cost **\$40,000**.

Summit Design, Inc.
(503) 643-9281
Fax: (503) 646-4954
www.summit-design.com

Nouveau PC

66-MHz PCI-BUS ANALYZER

The **PBT-415** is an advanced 32-/64-bit self-contained 66-MHz bus analyzer for the PCI bus with an onboard exerciser, its own processor, and firmware. It operates via an RS-232 serial line or USB port from either a standard ASCII terminal or a PC with BusView for Windows.

The unit can capture and trigger on all bus activity in 32-bit as well as 64-bit PCI-bus motherboards through a PCI expansion slot at speeds up to 66.7 MHz. The unit may also act as a 32-bit 33.3-MHz PCI master or target, controlled fully through the user interface or with the built-in script capability.

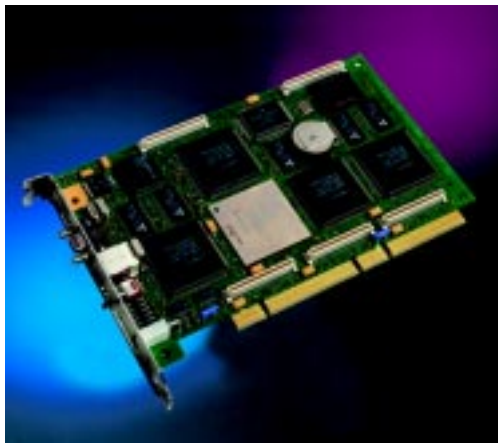
Features include 10–66-MHz sampling using CLK, transfer, or CLK transfer synchronous sampling, as well as an onboard 32-bit, 33-MHz PCI-bus exerciser with DMA capability. It has a trace

buffer up to 256 KB, demultiplexed address/data and command/byte enables, and real-time statistics, which include event counting and bus utilization. The PBT-415 offers 128 sampling channels for 32- and 64-bit PCI support as well as a 16-level sequencer that can trigger after count or delay.

An optional piggyback module (PTIMBAT400-PB) uses a combination of two diagnostic tools for analyzing the PCI bus. A 64-channel 400-MHz PCI timing analyzer and a PCI anomaly trigger unit automatically detect up to 68 PCI protocol or timing violations.

Pricing for the PBT-415 starts at **\$8750**.

VMetro, Inc.
(281) 584-0728
Fax: (281) 584-9034
www.vmetro.com



Nouveau PC

PISA-BUS SINGLE-BOARD COMPUTER

The **SBC-554V** is a half-size PISA (PCI + ISA) bus CPU card with onboard CRT/LCD controller and the revolutionary new DiskOnChip flash disk. The PISA-bus architecture supports both ISA and PISA-bus expansions on a single half-size CPU card.

The onboard C&T 65554 video controller is a high-performance LCD/CRT Windows accelerator. It supports a wide range of flat-panel displays, including 36-bit TFT panel displays. The DiskOnChip supports system boot-up and memory storage up to 72 MB.

The onboard VGA/LCD controller and DiskOnChip flash disk free expansion slots for peripheral devices. They also eliminate the need for additional boards and their compatibility problems.



The SBC-554V also includes two high-speed serial ports (one RS-232/-422/-485 and one RS-232), one multimode (ECP/EPP/SPP) parallel port, a floppy-drive controller, an Ultra DMA/33 enhanced IDE controller, and a keyboard/PS/2 mouse interface.

The SBC-554V supports ISA- and PCI-bus expansion cards and accepts Intel Pentium P54C and P55C, AMD K5 and K6, and Cyrix M1 and M2 processors.

In low quantities, prices range from **\$470 to \$490** (not including CPU, RAM, or DiskOnChip).

Aaeon Electronics, Inc.
(732) 203-9300
Fax: (732) 203-9311
www.aaeon.com

Nouveau PC

Interprocess Communication

Using Anonymous Pipes

While you can link DOS and Win32 applications using a virtual device that creates a wormhole between the systems, Ernie proposes a much simpler way—the anonymous pipes already native to any Win32 environment.

In his article “Interprocess Communication” (INK 87), Craig Pataky presented a technique for building a local, high-speed communication link between a DOS program and a Win32 application.

As he noted, such a link is handy for many purposes, including extending the lifetime of older DOS code by grafting on a modern GUI face with only minimal rework. Craig’s solution involved installing and interfacing with a custom virtual device driver (VxD) to build a wormhole that bridges the disparate universes of DOS and Windows.

On the surface, DOS and Windows may appear to be worlds apart, but not only did they originate in the same universe, they come from the same planet, even in the same neighborhood—Redmond, Washington. Surely there is a more down-to-earth way to connect the two.

In this article, I examine an alternate approach to building a Win/DOS communications link using nothing more than native Win32 and DOS features. This low-

tech solution is sufficient for most common needs involving user interface and simple data transfer.

INFOSTRUCTURE

Science fiction aside, in the real world of the twentieth century, we’re still struggling to connect distant points using such mundane technology as roads, bridges, and pipelines.

Pipelines are a vital part of the infrastructure, providing an efficient means of transporting materials from place to place. The basic pipeline concept can be applied to the transport of information as well—“infostructure” as opposed to “infrastructure,” if you will.

As an example, most computer operating systems support a localized form of information piping. Remember the DOS redirection symbols < and >?

Using these symbols on the command line, a user could control and direct information flow to and from other programs and disk files. The only requirement was for

software that used the standard system I/O devices and avoided direct hardware I/O.

With Win32, the command line has all but disappeared, but pipes still persist. Not only are pipes still available, but they have been significantly improved and enhanced to include user-defined, two-way, multiuser pipes with buffering, network communications, and other features.

BLUEPRINT

At this point, you’ve probably surmised that I plan to use pipes to build my Win/DOS communication link. In this discussion, the applications to be linked (both DOS and Win32) are running locally in a multitasking mode on the same computer.

Under these conditions, I don’t need any of the newer, more advanced Win32 pipe features. All that’s required are simple, unnamed, one-way pipes, similar to those found in DOS.

Reflecting their lack of a user-specified name, these are known as “anonymous pipes” in Win32, and as I will show you,

they are compatible with DOS. Win32 provides an alternate, more powerful pipe known as a named pipe. However, they aren't fully supported under Win95 or DOS, so I will avoid them here.

For my purposes, anonymous pipes offer adequate generic communication links that work in any Win32 environment (NT or 95) with any type of application (DOS or Win32).

One other Win32 feature figures prominently in my pipeline infrastructure—the ability of a Win32 application to manipulate and control the standard I/O channels of any child process launched via the `CreateProcess()` API call.

A general understanding of these two fundamental Win32 constructs—anonymous pipes and processes—are all you need for basic Win/DOS communications.

In my setup, the Win32 app serves as the master and must be launched first. The DOS app acts as a slave and is launched by the Win32 app as a child process using `CreateProcess()`.

As you'll see, the majority of the communication effort takes place on the Win32 side of things. In fact, given a DOS app that supports standard I/O, communication may

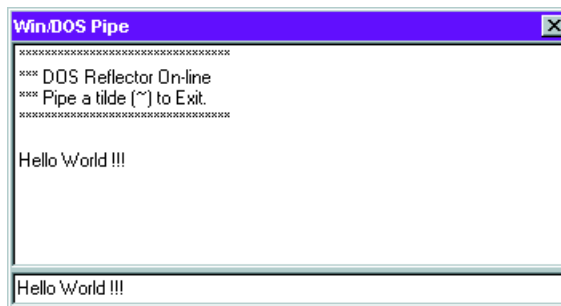


Photo 1—Text input on the command line at the bottom is sent to the hidden DOS app and reflected back for display in the window above.

not require any code changes at all on the DOS side. This is a definite advantage when you're working with third-party applications where source code is not available.

With this in mind, let's turn to my proposal—a generic blueprint for the construction of a Win/DOS pipeline.

To build a Win/DOS pipeline, the Win32 app must start by creating two anonymous pipes using `CreatePipe()`. Win32 pipes are buffered to help prevent data loss during read/write latency. If no buffer is requested, a default buffer is used. A 4-KB buffer should be sufficient for most purposes.

Every pipe has two ends, so a call to `CreatePipe()` returns two handles. One handle identifies the "read" end of the pipe, the other the "write" end.

The handles for one pipe should be named or somehow designated for use with `STDIN`, and the other with `STDOUT`. In the interest of simplicity, the `STDOUT` pipe can also serve `STDERR`.

To help keep the terminology straight, I identify the pipes as `STDIN` and `STDOUT` from the perspective of the DOS app. The individual ends are identified as `STDIN/Read`, `STDIN/Write` and `STDOUT/Read`, `STDOUT/Write`.

Once the pipes are created, the next step is to launch the DOS app as a child process using `CreateProcess()` with DOS `STDIO` redirected to the appropriate pipe handles. To achieve this, the previously created handles must be provided in the `STARTUPINFO` structure used by `CreateProcess()`.

The `hStdInput` element should contain the `STDIN/Read` pipe handle. Both `hStdOutput` and `hStdError` should contain the `STDOUT/Write` handle.

If everything goes according to plan, once the DOS app is up and running, what it recognizes as DOS `STDIO` will in fact be the Win32 pipes.

It's not enough to merely place handle values into `STARTUPINFO`. Unless explicitly told otherwise, Windows assumes that any data in the handle elements of `STARTUPINFO` is the result of a lack of initialization.

To indicate that the handle elements are in fact valid handles to be connected to the app being launched, the handle inheritance flag in `CreateProcess()` must be set to `TRUE`.

If `CreateProcess()` is successful, the Win32 app should be able to transmit a message to `STDIN` of the DOS app by piping info to the `STDIN/Write` handle using the `WriteFile()` API function.

Likewise, the Win32 app should be able to receive whatever the DOS app writes to `STDOUT` by using the `ReadFile()` API function and the `STDOUT/Read` handle. Thus, it's possible to establish a complete

Listing 1a—The MS-BASIC source code for a simple DOS reflector application redirects `STDIN` to `STDOUT`. **b**—Here's the same code in C.

```

a) DEFINT A-Z
OPEN "CONS:" FOR OUTPUT AS #1
PRINT #1, "*****"
PRINT #1, "*** DOS Reflector On-Line"
PRINT #1, "*** Pipe a tilde (~) to Exit."
PRINT #1, "*****"
PRINT #1,
DO
  A$=INPUT$(1)
  PRINT #1,A$;
  LOOP UNTIL A$="~"
PRINT #1, "Goodbye!"
CLOSE
END

b) #INCLUDE <STUDIO.H>
INT CH;

VOID MAIN()
{
  PRINTF("*****\n");
  PRINTF("*** DOS REFLECTOR ON-LINE\n");
  PRINTF("*** PIPE A TILDE (~) TO EXIT.\n");
  PRINTF("*****\n\n");
  DO {
    CH = GETCHAR();
    PUTCHAR(CH);
  }
  WHILE(CH != '~');
  PRINTF("GOODBYE!\n");
}

```


two-way communication link between DOS and Windows.

Once they're finished communicating, the apps must be shut down in reverse order. The last to start must be the first to end. In this case, it's the DOS app.

Why? Windows won't release an anonymous pipe if it's connected to a live, external application. Also, Windows won't properly terminate a Win32 application that owns a pipe which hasn't been properly released.

These factors combine to dictate an orderly shutdown procedure controlled by the Win32 side of things. The DOS app must be closed first, then the pipes, and finally the Win32 app.

That's it—four easy steps. Admittedly, I haven't addressed all the details and minutiae of using the necessary API function. However, you can find that in the many available books on Windows programming.

PERFORMANCE

I haven't conducted any definitive, quantitative tests on the communication speed of a Win/DOS pipeline like what I just described. However, simple observation and experience suggests that its performance is quite adequate for a user interface and simple interapplication data transfer. I'd hesitate to recommend using such a pipe for time-sensitive network data transfer, but for many lesser tasks, it performs quite well.

As with all areas of programming, poor design can lead to poor performance. So to avoid performance problems, take into account these design considerations.

When writing to a pipe, `WriteFile()` may not finish if the pipe buffer is full. In this case, `WriteFile()` waits for a read operation to remove data from the buffer and make more space available.

Obviously, for maximum speed, you want to avoid this situation. Make sure the pipe buffer appropriately reflects the size of the data packets being used. And, it should go without saying that the applications need to be in sync with regard to packet transfer.

When working with pipes, the overhead of the `ReadFile()` and `WriteFile()` API calls may be significant. The best way to minimize overhead is by using larger data packets.

Also, if possible, avoid reading data from a pipe in a piecemeal manner. Always try to remove all data currently

Listing 2—This Object Pascal source code for a Win32 application demonstrates two-way communication with a hidden DOS reflector program using pipes.

```
unit pipe1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, HyperStr;
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Edit1: TEdit;
    Timer1: TTimer;
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Timer1Timer(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure CloseDOSApp;
  private
  public
  end;
var
  Form1: TForm1;
  S,T:AnsiString;
  Flg:Boolean=False;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var
  I:Integer;
begin
  FlashSplash(Application.Icon,'Welcome');
  I:=GetTickCount+2000;
  try
    PipeExec('dospipe.exe',sw_Hide);
    SetLength(S,1024);
    FillStr(S,1,#32);
  except
    KillSplash;
    ShowMessage('Error opening pipe!');
    Close;
  end;
  Flg:=True;
  repeat until GetTickCount>I;
  KillSplash;
  KillIdle;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key=#13 then begin
    T:=Edit1.Text;
    Edit1.Clear;
    if CompareText(T,'Exit')=0 then
      CloseDOSApp
    else if Flg then begin
      Flg:=ScanF(T,'~',1)=0; //check for manual shutdown, set Flg
      WritePipe(T+#13+#10);
    end else SpeakerBeep;
    Key:=#0; //swallow the enter key
  end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
{Take steps to insure DOS app closes first; otherwise, the Win
 app will not close properly.}
begin

```

(continued)

Listing 2—continued

```
    if Flg then begin
        Action:=caNone;           //abort the close temporarily
        CloseDOSApp;             //shut down DOS app
    end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
{For demo purposes, a timer event checks the pipe once a second.
 A real app would undoubtedly use a different approach.}
var
    I,J,K,N:Integer;
begin
    I:=ReadPipe(S);              //check the pipe
    if I>0 then begin            //data available ?
        J:=DeleteC(S,#10);       //get rid of any line feeds
        if J<Length(S) then I:=I+J-Length(S);
        if I>0 then begin
            with ListBox1 do begin //add the text to listbox
                K:=1;
                if Items.Count=0 then Items.Add('');
                repeat
                    N:=Items.Count-1;
                    J:=ScanF(S,#13,K);
                    if (J>0) and (J<=I) then begin
                        Items.Strings[N]:=Items.Strings[N]+CStr(S,K,J-K);
                        Items.Add('');
                    end else begin
                        J:=I+1;
                        Items.Strings[N]:=Items.Strings[N]+CStr(S,K,J-K);
                    end;
                    K:=J+1;
                until J>=I;
            end;
        end;
        Form1.Edit1.SetFocus;
    end;

procedure TForm1.FormActivate(Sender: TObject);
begin
    BringToFront;
    Form1.SetFocus;
end;

procedure TForm1.CloseDOSApp;
{Do a quick and dirty shutdown of the DOS app to close the pipes.
 A more proper shutdown would wait for confirmation.}
begin
    WritePipe('~');              //send shutdown command
    Sleep(1000);                 //give it time to take effect
    Flg:=False;                  //clear flag
    Close;                       //manually retrigger the close
end;
end.
```

available from the pipe before attempting to sort things out.

DEMO

You can download the source and executables for a demo application that illustrates a working Win/DOS link. The demo consists of a Win32 app that simulates a simple console screen with a command line at the bottom (see Photo 1). A real-world application would probably

have a more sophisticated interface, but the underlying communication aspects would be pretty much the same.

At startup, the Win32 app builds the necessary pipes and connects them to a simple DOS app that is launched in a hidden DOS VM (virtual machine) window. The DOS window could easily be made visible.

For the demo, however, I chose to keep the DOS app out of sight and create the illusion of a Win32-only app. With a well-

designed application, the average user would probably be none the wiser.

I refer to the provided DOS app as a “reflector” because it merely reads data from STDIN and reflects it back to STDOUT. When the user presses Enter, whatever has been typed at the simulated command line in the Win32 app is taken and piped to the DOS program’s STDIN.

In turn, the DOS app reflects the data back to STDOUT. It is then received back by the Win32 app and displayed on the console screen. This effectively demonstrates full-circle communication from Win32 to DOS and back.

Strictly for illustration purposes, the hidden DOS app can be manually shut down prior to the Win32 app by piping in a tilde (~) and receiving back a “Goodbye!” message. Once the DOS app is shut down, any further attempts at communicating are met with a simple beep response.

In a more typical application, the DOS app would terminate as part of the standard Win32 shutdown, which was initiated in response to a normal user close

request. The demo includes the necessary code to address this situation as well.

When working with the demo, response time may occasionally seem a little sluggish. This situation is more a reflection of the simplistic nature of the demo rather than the speed of the communication link.

In the interest of simplicity and convenience, the demo uses a simple timer to check the pipe for incoming data once every second. Occasionally, this setup may produce a small but noticeable delay from the time a message is sent to the DOS app until it is retrieved again from the incoming pipe and redisplayed.

A real-world application probably requires a different approach to pipe management. One simple approach is to only check the pipe when data is expected. The main execution thread simply enters a loop where the pipe is continuously checked until the expected data arrives or a failsafe timeout occurs, indicating some sort of problem.

A more sophisticated approach, which may or may not be warranted, is to use the `WaitForSingleObject()` API function in combination with a separate background execution thread to continuously

monitor the pipe and generate a custom Windows message whenever data is available. An associated message handler then retrieves the data from the pipe.

SOURCE CODE

I made the source code to both the DOS and Win32 demo applications available to provide a first-hand look at the construction of a working Win/DOS pipeline.

The DOS reflector app in Listing 1a was compiled using MS BASIC V.7. The source code is simple, brief (just over a dozen lines), and generic. Listing 1b presents the same code in C.

My preferred Win32 development system is Borland’s Delphi, so I compiled the Win32 app using Delphi V.3 (see Listing 2) and my HyperString library. In addition to providing a wealth of string manipulation functions, HyperString offers a set of three simple functions—`PipeExec()`, `ReadPipe()`, and `WritePipe()`. These functions fully encapsulate all the details of building and using a Win/DOS communication link.

THE TOOLS AT HAND

The Win32 API is vast, broad, and a little overwhelming at first. It has lots of functionality—much more so than DOS.

However, the documentation can be a little sparse, particularly where interfacing with DOS is concerned. You’re often left to discover the best way to apply the available resources to the problem at hand.

Nevertheless, my preferred approach is always to use native functionality if possible. Only when no satisfactory solution can be found should a more drastic and invasive approach be explored. [EPC](#)

Ernie Deel owns and operates EFD Systems, a software design and development firm located in Marietta, GA. You can reach him at efd@mindspring.com.

SOFTWARE

The source and executable code (PIPEIT.ZIP) for this article can be downloaded from the Circuit Cellar Web site.

REFERENCE

B. Ezzell and J. Blaney, *NT4/Windows 95 Developer’s Handbook*, Sybex, Alameda, CA, 1997.

SOURCES

Borland Delphi

Inprise Corp.
(800) 457-9527
(408) 431-1000
www.inprise.com

HyperString

EFD Systems
efd@mindspring.com
www.mindspring.com/~efd

FAT32

File System for Data-Intensive Applications

When used with contiguous files, the FAT32 file system enables you to read and write high-speed datastreams to disk. Edward demonstrates this with multichannel audio input which he needs to edit and play back in real time.

Writing multiple streams of high-speed data to a disk can be a frustrating programming experience. But, you can reduce some potential loss of data due to head seek times by using double buffering and pipes.

However, you have to deal with complex programming and rely on either a high-level kernel or your expertise to handle the data. An easier method is to use the Windows 95 FAT32 file system along with a contiguous file system to write these high-speed streams of data to a disk.

In this article, I show you how this task is accomplished and then discuss how the data can be presented to a Windows 95 application for presentation or manipulation. The application records a multichannel audio input stream to a disk for later use by a Windows 95 application.

It demonstrates the use of contiguous block allocation for embedded systems. It also demonstrates the convenience of using standard file-system API calls to manipulate the files.

In this application, n (here, $n = 10$) audio channels are multiplexed in the time domain. A DSP front-end digitizes the input into discrete 512-byte packets, which are then written to a contiguous section of the disk.

This section of the disk is assigned to n files that are interleaved in a cyclic pattern, so each block is assigned to a separate file

representing the channel. Every n th block is owned by a file assigned to channel n .

Once the data is collected, it must be demultiplexed and each channel must be streamed through an audio player. Due to the high data rates involved, it isn't possible to perform disk seeks during the record or the playback sessions. The multiplexed data must be stored contiguously during collection and then demultiplexed to contiguous per-channel audio streams so the sound files can be played back.

Both the record and playback sessions require real-time response. During the transition from a record to playback session, you have time to move the data around. The resulting sound files must be accessible by a sound-editor application running under Windows 95.

I use programmed I/O to demonstrate the technique. In your application, the record and playback routines could use DMA to transfer the data to and from the contiguous regions of the

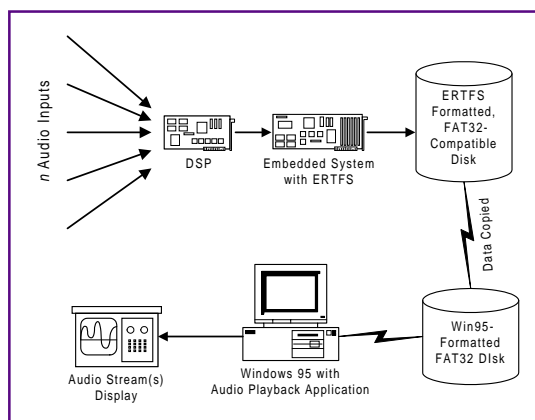


Figure 1—This system collects high-speed streaming audio data and efficiently stores the data to disk.

Listing 1—The MAIN routine includes the audio streaming definitions, an interface to the hypothetical DSP front end, and some error-message print routines.

```
#define DRIVENUMBER 0      /* Assume drive #0. We only have one drive*/
#define FREELISTSIZE 200 /* Assume max. 200 discontinuous free
                          segments */
#define NUM_CHANNELS 10   /* Collecting 10 audio channels */
#define NUM_BLOCKSPER 1000 /* 512 KB per channel is collected */
#define SAMPLESIZE 100   /* 100 blocks per DSP sample */

FREELISTINFO free_list[FREELISTSIZE];
/* Global array to hold free segment list */
unsigned char big_buffer[10240]; /* Buffer for moving data */

char *infile_names[NUM_CHANNELS] = {
    "input_file_0.snd", "input_file_1.snd",
    "input_file_2.snd", "input_file_3.snd",
    "input_file_4.snd", "input_file_5.snd",
    "input_file_6.snd", "input_file_7.snd",
    "input_file_8.snd", "input_file_9.snd"};

char *outfile_names[NUM_CHANNELS] = {
    "output_file_0.snd", "output_file_1.snd",
    "output_file_2.snd", "output_file_3.snd",
    "output_file_4.snd", "output_file_5.snd",
    "output_file_6.snd", "output_file_7.snd",
    "output_file_8.snd", "output_file_9.snd"};

/* Hypothetical DSP subsystem */
extern void dsp_start(int n_samples, int samplesize);
extern char *dsp_get_sample();
extern char *dsp_get_play_buffer(int samplesize);
extern void dsp_play_play_buffer(char *pdata);

/* Capture and play n channels of multiplexed audio.
   Returns 0 on success and -1 on failure.*/
int record_and_play(){
    int return_value;

    /* Check space then load coordinates into global array free_list */
    if (find_session_free_space() == -1){
        printf("Not enough contiguous disk space for session\n");
        return(-1);}
    return_value = -1; /* If we break out before completion report error */

    /* Create n interleaved files for incoming multiplexed datastream */
    if (create_input_data_files() == -1)
        printf("Failed to create input files\n");

    /* Create n contiguous files for outgoing audio streams */
    else if (create_output_data_channels() == -1)
        printf("Failed to create output files\n");

    /* Collect multiplexed data to the interleaved files */
    else if (collect_data() == -1)
        printf("Failed while collecting data\n");

    /* Demultiplex the data by copying it from the interleaved files
       to the contiguous files */
    else if (copy_input_to_output() == -1)
        printf("Failed copying input to contiguous output files\n");

    /* Stream demultiplexed data from contiguous files to audio system
       for playback */
    else if (copy_output_to_audio() == -1)
        printf("Failed while playing back audio\n");

    /* If no subsystems failed report success */
    else
        return_value = 0;
    delete_all_files(); /* Clean up */
    return(return_value); /* And leave */
}
```

disk, and the file-system code could be executed by the DSP front end.

To accomplish these goals simultaneously, I use an embedded real-time file system, the ERTFS V.1.0 from EBS. This embedded file system is Win95 compatible, has contiguous file support, and has direct block manipulation routines.

Many embedded applications and kernels have support for contiguous files and may have routines to directly manipulate disk blocks. ERTFS has all this plus Win95 FAT32 support, and the functionality can be added to either a stand-alone application or to an existing kernel.

Because the data sets are quite large and my blocking requirements are small (512 bytes per block), this application uses a 2.1-GB hard disk formatted with the FAT32 file system. This format provides a cluster size (minimum allocation unit) of one block per cluster.

FAT32 FILE SYSTEM

The FAT16 file system—the file system of the MS-DOS, Windows 3.1, and most versions of Windows 95 operating systems—is 21 years old. It was first developed for floppy disks.

Over the years, the FAT (file allocation table) has been modified to accommodate ever larger disks. It finally reached its limit with the 2-GB drives.

The FAT32 file system is an enhancement of the FAT16 file system and now supports larger hard drives with improved disk space efficiency. What makes the space usage more efficient is the smaller clusters used on the larger disks.

For disks up to 8 GB, the default cluster size is a modest 4 Kb compared to the 32-Kb cluster size for a 2-GB drive using the FAT16 format. For this application, I formatted a disk with ERTFS to specify a 512-byte cluster size.

There are some drawbacks to using the FAT32 format. Windows NT doesn't use this format, and converters aren't available for Windows NT 3.5 and 4.0 systems. Windows NT 5.0 systems are supposed to have a conversion utility to convert from FAT32 format to NTFS (the native NT file system).

Another drawback is that there are some compatibility problems with existing application programming interfaces (APIs) and older MS-DOS utilities. The cluster

values for FAT32 now use four bytes as compared to two bytes in the FAT16 system. The Win32 APIs aren't affected, and all disk utilities bundled with Windows 95 have been updated.

It is this four-byte cluster value that makes the FAT32 format so relevant to high-speed contiguous files. You can define clusters that are small enough to be equal to the size of the data blocks being collected and written to disk.

The older FAT16 cluster size value couldn't accommodate a large number of clusters, so as the disk size increased, it had to use ever-larger cluster sizes. This kept the value of the cluster size small enough to fit into the two bytes provided for the value.

CONTIGUOUS FILES

The normal disk structure has data written in the first available space. When that space is filled, a link is created to the next available disk space. When writing to the disk, the disk head is constantly going back and forth between the FAT and the area on the disk where the data is being written.

If a program could know beforehand that the entire file could be written in a

Listing 2—This code calls ERTFS and asks for a list of free segments. The first argument is the drive number, second is the size of the free_list array, third is the free list, and fourth is minimum size of contiguous regions to report.

```
int find_session_free_space(){
    int freelist_size;
    int i;

    /* Try to get all space in one chunk */
    freelist_size = pc_get_free_list(DRIVENUMBER, FREELISTSIZE,
        &free_list[0],(2 * (NUM_BLOCKSPER*NUM_CHANNELS)));
    if (freelist_size >= 1)
        return(0);          /* Got it */
    /* Couldn't get it in one chunk. Try it in two */
    freelist_size = pc_get_free_list(DRIVENUMBER, FREELISTSIZE,
        &free_list[0],(NUM_BLOCKSPER*NUM_CHANNELS));
    if (freelist_size >= 2)
        return(0);          /* Got it now */

    /* Not enough contiguous space. Dump free list to console.
       Return failure */
    /* Threshold of one returns all free regions */
    freelist_size = pc_get_free_list(DRIVENUMBER,FREELISTSIZE,
        &free_list[0], 1);

    printf("Free List\n");
    printf("CLUSTER    LENGTH\n");
    for (i = 0; i < freelist_size; i++){
        printf("%8ul    %8ul\n", free_list[i].cluster,
            free_list[i].nclusters);
    }
    printf("Storage allocation failed\n");
    return(-1);
}
```

**Listing 3a—This routine creates *n* interleaved files over a contiguous segment of the disk.
 b—This code creates 10 contiguous 512-KB files which store the output to stream to the playback application.**

```

a) int create_input_data_files(){
    int n, j;
    PCFD fd;
    long cluster;
    for (n = 0; n < NUM_CHANNELS; n++){ /* Open channel n */
        fd = po_open(infile_names[n], PO_BINARY|PO_RDWR|PO_CREAT,
                    PS_IWRITE|PS_IREAD);
        if (fd < 0){
            printf("File creation error \n");
            return(-1);
        }
        /* Cluster offset for files is 0,1,2,3,... for channels 0,1,2,... */
        cluster = free_list[0].cluster + n;
        /* Allocate one block every nth block for data channel */
        for (j=0; j < NUM_BLOCKSPER; j++){
            if (po_extend_file(fd, 512, cluster, PC_FIXED_FIT, FALSE) < 0){
                printf("File extend error \n");
                po_close(fd);
                return(-1);
            }
            cluster += NUM_CHANNELS;
        }
        po_close(fd); /* Close this file and do another */
    }
    /* Created 10 512-KB input files. Blocks are
       CHANO|CHAN1|CHAN2|CHAN3...|CHANO... */
    return(0);
}

b) int create_output_data_channels(){
    int n;
    PCFD fd;
    long cluster;
    long file_size;
    /* Get cluster in contiguous region.
       If we get here, there are enough free blocks */
    if (free_list[0].nclusters >= (2*NUM_BLOCKSPER*NUM_CHANNELS)){
        /* If input and output files allocated in one segment,
           use second half of segment for allocation. */
        cluster = free_list[0].cluster + (NUM_BLOCKSPER*NUM_CHANNELS);
    }
    else{
        cluster = free_list[1].cluster; /* Use beginning of 2nd segment */
    }
    file_size = (long)NUM_BLOCKSPER; /* Byte size of each data file */
    file_size = (long) (file_size * 512);
    for (n = 0; n < NUM_CHANNELS; n++){ /* Open channel n */

        fd = po_open(outfile_names[n], PO_BINARY|PO_RDWR|PO_CREAT,
                    PS_IWRITE|PS_IREAD);
        if (fd < 0){
            printf("File creation error \n");
            return(-1);
        }
        /* Allocate one contiguous chunk for output channel
           Final argument specifies device driver to preerase data
           p blocks if device supports preerase */
        if (po_extend_file(fd, file_size, cluster, PC_FIXED_FIT, TRUE) < 0){
            printf("Output File extend error \n");
            po_close(fd);
            return(-1);
        }
        cluster += NUM_BLOCKSPER; /* Next file offset by NUM_BLOCKSPER */
        po_close(fd); /* Close file; do another */
    }
    return(0); /* Created 10 512-KB output files, contiguous */
}

```

contiguous portion of the disk and if it knows the address of this space, the program could issue direct read and write commands. This is what the ERTFS, LynxOS, and SMX products provide to the developer. (Note that ERTFS and SMX are MS-DOS compatible, whereas LynxOS is Posix compliant.)

A contiguous file is a section of the disk consisting of sequential physical blocks which, after being allocated, are treated as a high-performance raw device. A contiguous file usually doesn't use the normal buffered file system but is accessed in block units of the file system.

The FAT32-compatible format in this application uses a 512-byte cluster size. Using this cluster size, I can interleave the 10 channels in a contiguous region of the disk. Because the DSP collects data in 512-byte blocks per channel, this design is as efficient as possible in disk writes of one cluster per write.

The ERTFS has functions to read and write up to 128 blocks directly to and from a disk. With these functions, you specify the starting block number and the number of blocks to transfer.

Although DOS APIs may happen to create contiguous files, there is no way to specifically request a contiguous file without contiguous or sequential file support. Any contiguous file created, however, can be read by DOS-compatible utilities.

PROGRAM STEPS

The algorithm implemented for the n -channel audio datastreaming application illustrated in Figure 1 requires these seven steps:

- allocate a contiguous segment of the disk to store the incoming datastream
- interleave the blocks in the contiguous segment, so that every n th block is associated with the same file. n is the number of input channels (in this case, 10)
- for the output files, allocate another contiguous segment of the disk to store a copy of the input stream in n contiguous files (one for each channel)
- collect the data and store interleaved in the contiguous segment of the disk created for the input data
- demultiplex the data by copying the data from the interleaved input file to the contiguous output files
- play back each channel
- clean up the disk

Listing 4—This routine enables you to collect the audio data.

```
int collect_data(){
    PCFD fd;
    long blockno;
    char *pdata;
    FILESEGINFO fileinfo;
    int n_samples;

    /* Open channel 0 and get block number of first block in file.
       Beginning of contiguous region allocated */
    fd = po_open(infile_names[0], PO_BINARY|PO_RDWR,0);
    if (fd < 0){
        printf("File open error\n");
        return(-1);
    }

    /* Ask for list of block extents that make up file. Only need
       first block. raw_io flag is false since partition mapping
       included when we write file */
    if (pc_get_file_extents(fd, 1, &fileinfo, FALSE) < 0){
        po_close(fd);
        printf("Error getting file extents\n");
        return(-1);
    }

    /* Close file. */
    po_close(fd);
    blockno = fileinfo.block; /* Here it is */

    /* How many samples to collect (total # of blocks/SAMPLESIZE) */
    n_samples = (NUM_CHANNELS*NUM_BLOCKSPER/SAMPLESIZE);
    /* Tell DSP to Collect n_samples of SAMPLESIZE */
    dsp_start(n_samples, SAMPLESIZE);
    /* Loop. Wait for samples and write to disk */
    while (n_samples--){
        pdata = dsp_get_sample(); /* Wait for sample */
        /* Write SAMPLESIZE blocks from pdata to the block at blockno.
           raw_io argument is false because we want partition mapping */
        if (pc_raw_write(DRIVENUMBER, pdata, blockno, SAMPLESIZE,
            FALSE) < 0){
            printf("Error writing to disk\n");
            return(-1);
        }
        blockno += SAMPLESIZE; /* Wrote SAMPLESIZE blocks. Increment
                               block pointer. */
    }
    return(1);
}
```

The main routine and the definitions are in Listing 1. The code in Listing 2 scans the disk drive for enough free space to run the application. If it finds the space, the size will be returned in the `free_list` array. If it does not find enough space, it analyzes the disk and prints the free map for informational purposes.

Two contiguous regions on the disk are needed to hold the data collection of size $(\text{NUM_CHANNELS} * \text{NUM_BLOCKSPER})$ blocks, which is 512,000 blocks. The routine first tries to allocate all the data from one region. If that does not work, it tries to allocate the data in two segments. All of the routines return 0 on success and -1 on failure.

Listing 3a creates n interleaved files over a single contiguous segment of the disk. Here, my routine creates space for 10 interleaved files, each containing 1000 blocks of data.

The data is laid out so that every n th block belongs to a specific input channel (see Figure 2). In other words, CH0 is a member of `INPUT_FILE_1`, CH1 is a member of `INPUT_FILE_2`, and so on.

In Listing 3b, the routine creates the n contiguous files to hold data to be streamed through an audio playback system. These files are used by the embedded system or copied to the Windows 95 application system.

All blocks within a file should be contiguous to minimize disk head movement and

Listing 5—Once the data is demultiplexed into separate contiguous files, the input data may be streamed to the audio player.

```
int copy_input_to_output(){
    int n,i;
    PCFD in_fd;
    PCFD out_fd;
    for (n = 0; n < NUM_CHANNELS; n++){ /* Open channel n */
        in_fd = po_open(infile_names[n], PO_BINARY|PO_RDONLY,0);
        out_fd = po_open(outfile_names[n], PO_BINARY|PO_WRONLY,0);
        if ((in_fd < 0) || (out_fd < 0)){
            if (in_fd >= 0) po_close(in_fd);
            if (out_fd >= 0) po_close(out_fd);
            printf("File open error \n");
            return(-1);
        }
        /* Read from input and write to output */
        /* Work 20 blocks at a time, given 10,240-byte buffer */
        for (i = 0; i < NUM_BLOCKSPER; i += 20){
            if (!(po_read(in_fd, big_buffer, 10240) == 10240) &&
                (po_write(out_fd, big_buffer, 10240) == 10240)){
                po_close(in_fd);
                po_close(out_fd);
                printf("File copy error \n");
                return(-1);
            }
        }
        po_close(in_fd); /* Close files and loop back for next pair */
        po_close(out_fd);
    }
    return(1); /* Copied all files */
}
```

Listing 6—This code enables you to play the audio data back.

```
int copy_output_to_audio(){
    PCFD fd;
    long blockno;
    char *pdata;
    FILESEGINFO fileinfo;
    int i;
    int channel;
    for (channel = 0; channel < NUM_CHANNELS; channel++){
        /* Open channel */
        fd = po_open(outfile_names[channel], PO_BINARY|PO_RDONLY,0);
        if (fd < 0){
            po_close(fd);
            printf("File open error \n");
            return(-1);
        } /* Get starting block of file. We know it is contiguous */
        if (pc_get_file_extents(fd, 1, &fileinfo, FALSE) < 0){
            po_close(fd);
            printf("Error getting file extents\n");
            return(-1);
        }
        else blockno = fileinfo.block; /* Here it is */
        po_close(fd); /* Close file. Play all data in one channel */
        /* How many samples is (total number of blocks/SAMPLESIZE)? */
        for (i = 0; i < (NUM_BLOCKSPER/SAMPLESIZE); i++){
            pdata = dsp_get_play_buffer(SAMPLESIZE); /* Call DSP */
            if (pc_raw_read(DRIVENUMBER, pdata, blockno, SAMPLESIZE,
                FALSE) < 0){ /* Read contiguous blocks from disk */
                printf("Error reading output sample\n");
                return(-1);
            }
            dsp_play_play_buffer(pdata); /* Tell DSP it's loaded */
            blockno += SAMPLESIZE; /* Increment block pointer. */
        }
    }
    return(0);
}
```

reduce disk access times.

This setup is not necessary for the application to function, but it makes it simpler to implement and also eliminates fragmentation and ensures the output data will be streamed smoothly (no breaks while waiting for disk head movement).

The routine in Listing 4 collects the 10 channels of multiplexed audio data into *n* interleaved files. It collects data from a DSP or any other front-end data-collection system and writes the raw blocks to the interleaved data block region. The DSP system provides 100 block buffers of data at a time to the upper layers of the application software.

The buffers are in a ring buffer, so the application calls the DSP software layer to provide a buffer. When it returns a buffer, the data is written to disk. Once the write is completed, the buffer is given back to the DSP layer. The DSP system and the application run asynchronously.

In Listing 5, the contents of the input data files are copied to the output files. The data is automatically demultiplexed, because the input files are interleaved and the output files are contiguous.

This process doesn't happen in real time because the disk must seek as it reads the data blocks from the input files. The interesting thing about this routine is that a simple file copy using standard API calls demultiplexes the data into contiguous output files.

Next, we finally get to stream the contiguous data to the audio player. This can be either done by the embedded system or through a Windows 95 application.

In Listing 6, the routine assumes that the embedded system is used to play back the audio streams. It reads blocks from output files that are contiguous and commits the blocks to the DSP system to be played as audio.

A final routine (not shown here) deletes all the input files and output files to release the contiguous space. This routine calls `pc_unlink()` for each file that may have been created.

If the algorithm runs to completion, it deletes every file. If it doesn't run to completion but leaves some files on the disk, it deletes only those files. The `unlink` call will fail on the files that we did not create, which doesn't cause any harm.

MAIN PROGRAM

The main program, which you see in Listing 1, makes sure there is enough

contiguous disk space to collect the multiplexed data in a contiguous section and to store the demultiplexed data files in contiguous sections. Then it creates NUM CHANNEL interleaved files so the multiplexed input stream can be collected to contiguous sectors and later demultiplexed.

After it collects the data, it demultiplexes it by copying the data on a per-channel basis from the interleaved input files to the contiguous output files. It then plays back each channel and deletes all the files.

FAT32 AND ERTFS

Using the Windows 95 FAT32 file system with its smaller cluster sizes and your embedded application to allocate contiguous files and to read and write directly to disk blocks can make for a fast data-collection system for continuous datastreams.

The reduced head seek times made possible by using contiguous files mean you will not lose incoming data. The ability then to copy the input file into any number of contiguous output files means the output device can stream the data without breaks due to disk head movement.

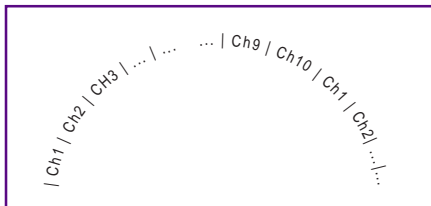


Figure 2—In the 512-KB block/channel layout on the disk, the data from the channels are interleaved every nth block in a single contiguous file.

The ERTFS product has a unique function that may be required when the length of the datastream is not known. If the file size you preallocated was not sufficient to contain the datastream, the ERTFS product has a function to extend the contiguous file.

The ERTFS API provides for calls to get the first available contiguous chain of clusters of sufficient size to contain the extension (fastest method), the chain of clusters that allows for the best fit, or the longest chain of available clusters. The extension may not be contiguous with the first allocated space but might be linked like noncontiguous files.

Some embedded kernels have contiguous (sequential) file capability. The ERTFS file system from EBS can be used with

kernels that do not possess this feature or it can be incorporated easily into embedded products that do not contain a file system. **EPC**

Edward Steinfeld has over 25 years' experience in real-time and embedded computing. He has marketed embedded and real-time products to OEMs and resellers for DEC, VenturCom, and Phar Lap Software. He now heads his own company, Automata International Marketing. You may reach Edward at stein@ma.ultranet.com.

SOURCES

ERTFS V.1.0

EBS, Inc.
(978) 448-9340
Fax: (978) 448-6376
www.etcbn.com

FAT32

Microsoft Corp.
(206) 882-8080
Fax: (206) 936-7329
www.microsoft.com/windows/pr/fat32.htm

LynxOS

Lynx Real-Time Systems, Inc.
(498) 879-3900
Fax: (408) 879-3920
www.lynx.com

SMX

Micro Digital, Inc.
(714) 373-6862
Fax: (714) 891-2363
www.smxinfo.com

Ingo Cyliax

Data Acquisition

Collecting data in real time is tricky. You have to consider the response to real-time inputs as well as the delay introduced by data buffers. Ingo gets us up to speed on possible solutions, including DSP- and FPGA-based PC/104 modules.

When I discussed networking last month, one of the things I mentioned was data-acquisition devices over Ethernet. This time, let's take a look at the business end of that data acquisition.

In particular, I want to talk about PC/104-based data-acquisition solutions. After filling you in on the terminology, I discuss some of the issues that arise when you're dealing with data acquisition in real-time systems.

In the Sources section, I list a number of vendors that make PC/104-based data-acquisition modules. There are a lot! But, that's not surprising. After all, most real-time systems interface with the real world, which is very analog in nature.

ANALOG ACQUISITION

With data acquisition—and analog data acquisition, in particular—you need to know some terminology be-

fore we can look at issues and techniques. It's pretty simple, so let's dive right in.

The A/D sampling rate is the speed at which you sample the analog signal. Typical numbers are 30 kHz or 100 kS/s (kilosamples per second).

The maximum sampling rate depends on the particular A/D chip or technique, and it's related to the resolution of the ADC. In general, the higher the resolution (i.e., the number of bits in the converted digital word), the lower the sampling rate.

The resolution defines how many steps there are in the signal range to be sampled. For example, if the analog signal range is 0–10 V and the resolution is eight bits, the step size is $10/256$ or 0.039 V. If the resolution is 12 bits (a typical size), the step size goes down to 0.0024 V.

The resolution can also be expressed in dynamic range. A 12-bit ADC has a dynamic range of 4096:1 or $20\log(4096)$, which equals 72 dB.

The dynamic range published by a manufacturer for a particular device is a figure of merit. It also has to take into account the converter's analog performance. It's not that uncommon for the dynamic range of a converter to be smaller than the resolution might indicate.

The input bandwidth of a converter, which describes its analog performance, is important as well. To make

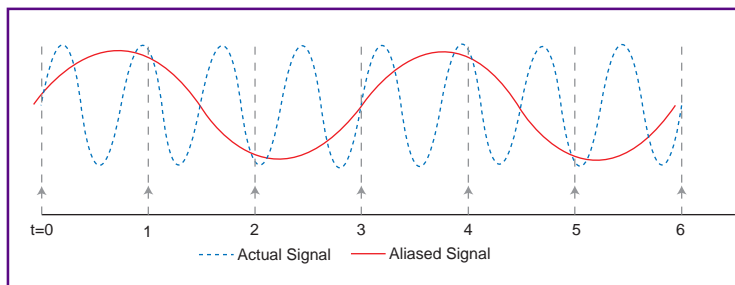
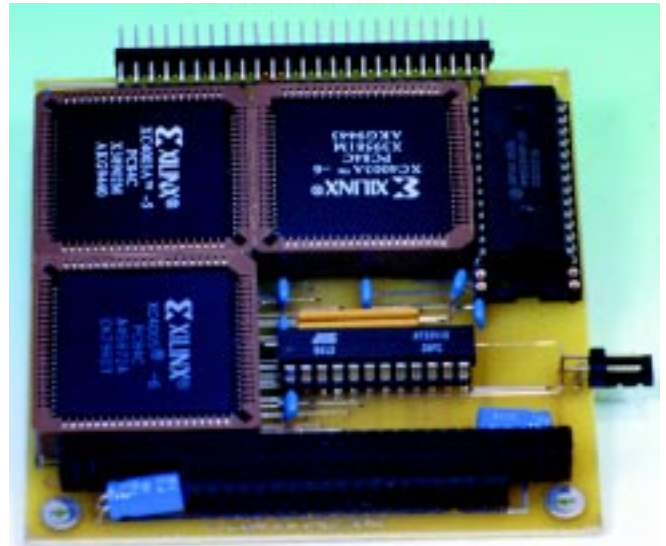


Figure 1—Aliasing occurs when a signal is undersampled. To sample a base-band signal, the signal needs to be band-limited to half the sampling rate. The minimum sampling rate should be twice that of the maximum frequency component present in the signal.

Photo 1—The PF2000 from Derivation Systems can be configured to have up to three Xilinx FPGAs and up to 512 KB of SRAM, EPROM, or flash memory. Up to four boards can be used in one system. Complex algorithms can be implemented in the FPGA on this PC/104 form-factor board to offload the system bus and CPU.



sure that all of the signal is present, the converter's input bandwidth should be larger than the signal bandwidth you're interested in.

In most applications, the input bandwidth to the converter should be less than half the sampling rate. The converter's sampling rate should be the Nyquist rate, which is at least twice that of the highest frequency component you're interested in.

Sampling at less than two times the signal rate introduces aliasing to the signal. Figure 1 gives you an idea of what aliasing looks like.

To measure a signal that has a bandwidth of 0–20 kHz, you need to sample it with at least 40 kS/s. Audio CDs are sampled at 44.1 kS/s to reproduce high-fidelity audio with high-frequency components over 20 kHz.

However, a converter may have much higher input bandwidth than half the sampling rate. For example, a 40-MS/s converter might have an input bandwidth of 100 MHz. That way, you can undersample to capture signals with higher frequency components than the sampling rate.

How is this possible? Digitizing an analog signal or sampling it at discrete time steps is much like heterodyning, or mixing, a signal.

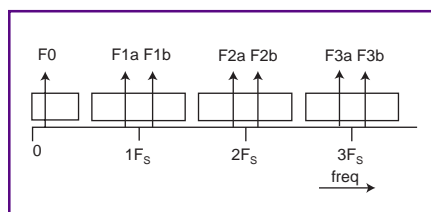


Figure 2—Undersampling can be used to acquire signal in frequency bands that are higher than the sampling rate. The aliasing works in my favor here, down-converting the desired frequency band into a base-band signal. For example, the images of F2a and F2b can be found in the base band F0.

That's what the aliasing effect is, and you can use it to your advantage. For example, sampling a signal at 20 MS/s will downconvert a 25- or 45-MHz signal to a 5-MHz signal.

To select the signal band you wish to convert, you first need a band-pass filter to select the desired band you want imaged in the sampled signal. Figure 2 shows what this looks like in the frequency domain. Filtering the base-band signal at half the sampling rate to get the actual signal without aliasing is just a special case [1].

Delay or latency is the time it takes to convert the analog signal and have the converted digital value of the signal ready. The latency may be longer than the sampling rate because some converters have pipelines and FIFOs. Therefore, the data may appear several sample times later.

You must account for the latency in real-time-based systems because it affects the system's response to an external stimulus. Be sure to add the converter latency to your total system-latency budget.

Some converters are able to convert several signal sources with one ADC. This task is achieved via an analog input multiplexer (see Figure 3). This device selects one of the inputs as a source for the ADC input. In this case, you sacrifice sampling rate for the number of inputs.

If you want to sample eight channels at 40 kS/s each, you need an ADC that's capable of sampling at 8×40 kS/s (or 320 kS/s) because it has to service each channel in order. Another way to look at the situation is that a 40-kS/s ADC can only sample 16 channels at 40 kS/s divided by 16 (or 2.5 kS/s).

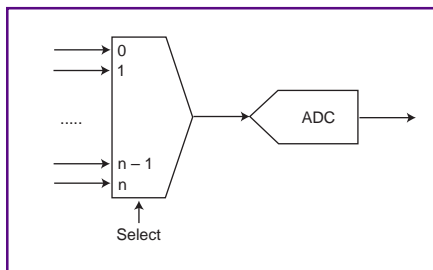


Figure 3—An analog multiplexer increases the number of inputs to a converter. This multiplexer is typically incorporated on the converter chip.

ISSUES

Latency usually isn't a big problem with nonreal-time applications, like test and measurement, where we want to collect some data and then store it for later analysis. In real-time systems, however, you typically want to control something based on the signals acquired, so latency becomes a critical issue.

A flight simulator is a good example of a demanding data-acquisition application. It measures pilot inputs to the system (e.g., pedal and yoke positions) and computes the response to the control inputs.

Typically, the computation models the aerodynamics of the aircraft to be simulated and then updates the hydraulic actuators that move simulator's cockpit so that the pilots feel like they're flying the aircraft. Also, the simulator has to update the displays to be realistic facsimiles.

It turns out that if the update rate is too sluggish, the pilots in the simulator get sick. In this case, then, there's an upper bound on how much system latency can occur.

If the acquisition system has too much latency, it isn't possible to meet this end-to-end latency requirement, even if there is enough processing power to handle all the computation required.

A flight simulator is just one example of a control loop that illustrates the effect of latency in acquisition systems. Other control examples (e.g., factory process controllers or robot actuators) have even higher latency requirements.

Another issue that arises in data acquisition is the data throughput based on sensor inputs. In a real-time system, the device has to respond to inputs in a bounded amount of time. Therefore, it has to acquire all the data it needs for the computation at the required sampling rate.

For example, if you have to sample data at 500 kS/s, the system must be able

to handle that amount of data. At 500 kS/s with a resolution of 16 bits, your system has to read and process 1 MBps of data.

This situation can be problematic for a PC/104 system. Such a system might be able to just handle this data rate, but there wouldn't be any excess bandwidth to do anything else on the PC/104 bus (e.g., drive a graphics board).

Obviously, one way to increase performance is to use a faster bus. PC/104+, which runs at 33 MHz, allows burst transfers of up to 132 MBps. That certainly helps. Even if you don't have to transfer

huge amounts of data, PC/104+ permits much higher bus utilization because the device is on the bus for much shorter periods of time.

Another approach to the data-bandwidth problem is to preprocess the collected data. In many cases, the raw data must be sampled at high rates so that information (e.g., the derivative or frequency spectrum of the signal) could be extracted. Or, perhaps you want to recover an interesting signal by filtering out noise instead.

Some of the processing can be done near the converter. Rather than converting the data and letting the system CPU process all the data, try using a DSP-based board that interfaces to the ADC.

The DSP can then preprocess the data and extract the essential information. In this case, the data is reduced before being transferred over the system bus.

A related approach is to use FPGAs for signal preprocessing. In addition to implementing DSP function in FPGAs, FPGAs can also be used for customized I/O functions and acquisition architectures.

Two vendors have FPGA modules in PC/104 format. The module shown in Photo 1 is one I built, and it's available from Derivation Systems Inc.

Nova Engineering also offers a PC/104-based FPGA module built around the Altera FPGA. The Nova's FPGA module is available in a variety of options, including low-voltage FPGA. Both Altera and Xilinx provide libraries of DSP core functions that can be embedded in an FPGA.

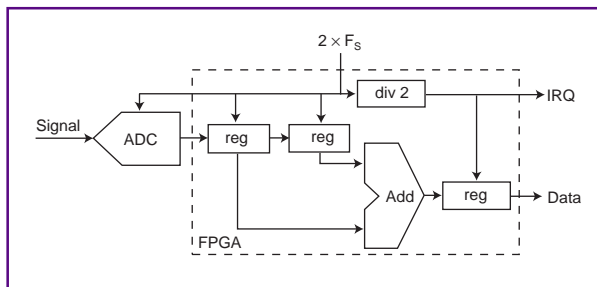


Figure 4—An FPGA can be used to preprocess data acquired before passing it on to the CPU. In this example, I oversampled a signal and the FPGA averaged it to increase the resolution and reduce traffic over the system bus.

You might wonder what kind of preprocessing you can do in a FPGA- or DSP-based acquisition system. Recall that I mentioned using undersampling to downconvert signals at higher frequencies. You can also try oversampling.

In oversampling, you sample the signal at a higher rate than the Nyquist rate, which provides more samples than you need to reconstruct the signal. It also gives an improved dynamic range—that is, it increases the effective resolution.

The sigma-delta converter is an extreme case of this technique. It uses a one-bit ADC and oversamples to get the desired

resolution. If you oversample the signal with a 2x sampling rate, you must reduce the signal by averaging the two samples you get for every one sample you're interested in:

$$y(t) = \frac{x(t1) + x(t2)}{2}$$

The resolution increase comes from the add operation. Since you're dealing with integers, you don't perform the divide-by-two operation, which would truncate the result. The add operation introduces a carry bit when the result overflows the original sample size.

If the averaging operation is performed in the CPU, the higher sampling rate increases the system's bandwidth requirement. To lower this requirement, you can use FPGAs for this operation. Figure 4 shows a system that downconverts and averages the signal. You can also use a DSP for this.

The FPGA interfaces to the ADC and the PC/104 bus. In this case, you can also derive the ADC clock and divide it by two to get an interrupt signal (IRQ) for the CPU.

The data from the ADC is shifted into two holding registers—one holds the current value, and the other the previous value. An adder then adds these two registers and stores the result in a hold register. The divide-by-two clock interrupts the CPU, which reads the averaged result.

Perhaps this example isn't so drastic, but you can see how this technique works well if you oversample by 8 or perhaps 16 times to increase system resolution, while keeping the system bus bandwidth down. One advantage to using an FPGA is that you can use nonstandard word sizes, like nine-bit results from adding two eight-bit samples, and perform operations, like a four- or eight-way adds in parallel.

Oversampling relies on the fact that the signal contains noise. By oversampling, you time-average the signal and the noise. The noise then gets averaged out, since it has a zero mean.

A sigma-delta converter typically adds noise to the input signal to dither it, increasing the resolution and decreasing the overall converter signal-to-noise ratio [2,3].

WHAT ELSE IS OUT THERE?

DSP- and FPGA-based PC/104 modules can be used to offload the bus by preprocessing data. But, you can also find conventional acquisition modules.

Acquisition boards are abundant for PC/104 module format. Some range from 8 to 16 bits and even 18 bits at rates from 20 to over 100 kS/s. Some have up to 16 input channels. And, some high-speed acquisition modules from Chase Scientific go as high as 200–250 MS/s.

The wide selection in I/O devices makes it attractive to use PC architectures for embedded real-time applications.

Naturally, there's a lot more to real-time data acquisition than I can possibly cover here. But, I'm sure the vendors can fill you in on what might work best for your application. [RPC.EPC](#)

Ingo Cyliax has been writing for INK for two years 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. Before joining DSI, Ingo worked for over 12 years as a system and research engineer

for several universities and as an independent consultant. You may reach him at cyliax@derivation.com.

REFERENCES

- [1] E.C. Ifeachor and B.W. Jervis, *Digital Signal Processing*, Addison-Wesley, New York, NY, 1993, 14–28.
- [2] E.C. Ifeachor and B.W. Jervis, *Digital Signal Processing*, Addison-Wesley, New York, NY, 1993, 492–495.
- [3] J. Watkinson, *Art of Digital Audio*, Focal Press, Oxford, UK, 1994, 120–130.

RESOURCES

P. Horowitz and W. Hill, *The Art of Electronics*, Cambridge University Press, New York, NY, 1989.
PC/104 Consortium, *PC/104 Consortium Resource Guide*, Mountain View, CA, 1997.

SOURCES

PC/104 DSP Modules

Dalanco Spry
(716) 473-3610
Fax: (716) 271-8380
www.vivanet.com/~dalanco

Lila Fabriken AB
+46 8-287286
Fax: +46 8-288802
www.lillfab.com

Traquair Data Systems, Inc.
(607) 266-6000
Fax: (607) 266-8221
www.traquair.com

PC/104 FPGA Modules

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

Nova Engineering, Inc.
(513) 860-3456
Fax: (513) 860-3535
www.nova-eng.com

PC/104 high-speed acquisition cards

Chase Scientific Co.
(408) 464-2584
Fax: (408) 479-8572
www.chase2000.com

PC/104 acquisition modules

Aaeon Electronics, Inc.
(732) 203-9300
Fax: (732) 203-9311
www.aaeon.com

Ajeco Oy
+358 9.7003.9200
Fax: +358 9.7003.9209
www.ajeco.fi

Analogic Corp.
(978) 977-3000
Fax: (617) 245-1274
www.analogic.com

Arcom Control Systems, Inc.
(888) 941-2224
(816) 941-7025
Fax: (816) 941-0343
www.arcomcontrols.com

Axiom Technology, Inc.
(888) GO-AXIOM
(909) 464-1881
Fax: (909) 464-1882
www.axiomtek.com

14-channel 18-bit A/D module

Computer Dynamics
(803) 627-8800
Fax: (803) 675-0106

Real Time Devices USA, Inc.
(814) 234-8087
Fax: (814) 234-5218
www.rtdusa.com

Applied PCs

Fred Eady

Debugging & the Net186

Sure, for basic debugging, you can fix this baby with AMD's E86MON. But, if what you need to do is more challenging, you might want to check out a full-fledged debugging package like Paradigm's. Listen up while Fred walks you through it.

I don't get to watch a bunch of television, so when I do get the chance, the show had better be really good. Just the other night while I was mentally preparing to write this piece, I was taking in some "this is how the Pharaoh did it" on The Learning Channel.

It dawned on me. We all know what an ancient Egyptian is, but do we really know as much as we think we do about them?

The light got brighter. We all know what debugging is, but do we really know as much as we think we do about it? For instance, there are at least two ways to implement debugging techniques on the AMD Net186 board, shown in Photo 1. Can you name them?

NETTING THE Net186

Now that the questions have been presented, let's proceed with finding out how much debug energy we can throw at the Net186. (You can go to

The Learning Channel later for the ancient Egyptian answers.)

Before diving into the software, I want to talk some about the Net186 hardware. The AMD Net186 is really a demo board measuring 3.5 in². This

little board is fascinating. There are only ten circuit packages onboard.

The Am186ES microcontroller runs at 40 MHz and is accompanied by AMD's Am79C961A PCnet-ISA II Ethernet controller. There's Am29F400 flash memory and 512 KB of SRAM, too. Throw in a couple of MAX232 RS-232 ICs and a PALCD22V10, mix in some E86MON monitor software, and there it is.

Notice that I didn't mention any glue parts. Good reason. There aren't any. The PAL provides what little glue functionality is needed because the Ethernet controller IC doesn't require bunches of interface parts. A simplified block diagram of the Net186 is shown in Figure 1.

The Am186ES micro is much like its Intel cousins. This part combines twelve 16-bit memory chip-select controllers, two async serial controllers, three timers, 32 programmable I/Os, an interrupt controller, and a watchdog timer. The part is self-adhesive, supporting a glue-



Photo 1—Not bad for an embedded application not much bigger than a deck of cards.

less connection to SRAM, flash memory, and EEPROM. Serial ports and Ethernet capability point the Net186 demo board at developers thinking of integrating the Am186ES and supporting cast into what we all have come to know as 'Net appliances.

With that thought, the Net186 board comes loaded with a Web-server application that uses a TCP/IP stack and application provided by US Software. I found this to be of great interest until I discovered that the source code for the application was not furnished because of licensing.

Of course, that put a damper on things, but the application was good and really showed off the muscle the Net186 board possesses. The Web-server application enables the Net186 to respond to HTTP requests over the Ethernet network and return Web pages to a Netscape browser running on a separate machine on the network.

I didn't have any trouble getting the home site to appear. The instructions were clear and even included some refreshing humor. I threw in some IP addresses, added TCP/IP to my Win95 test machine, and bound it to my Ethernet card, and it all appeared in my Win95 HyperTerminal window.

Some time ago, I wrote some text on emWare ("Interfaces and GUI-Building Packages," /NK 88 and 89). emWare is a software product that lets the programmer simulate real hardware interfaces via an Internet connection.

The Net186 Web-server application has a piece that is very close to that concept. A page is set aside that emulates the LED array on the Net186 board.

By clicking on an LED in the Netscape window, you can turn the physical LED on

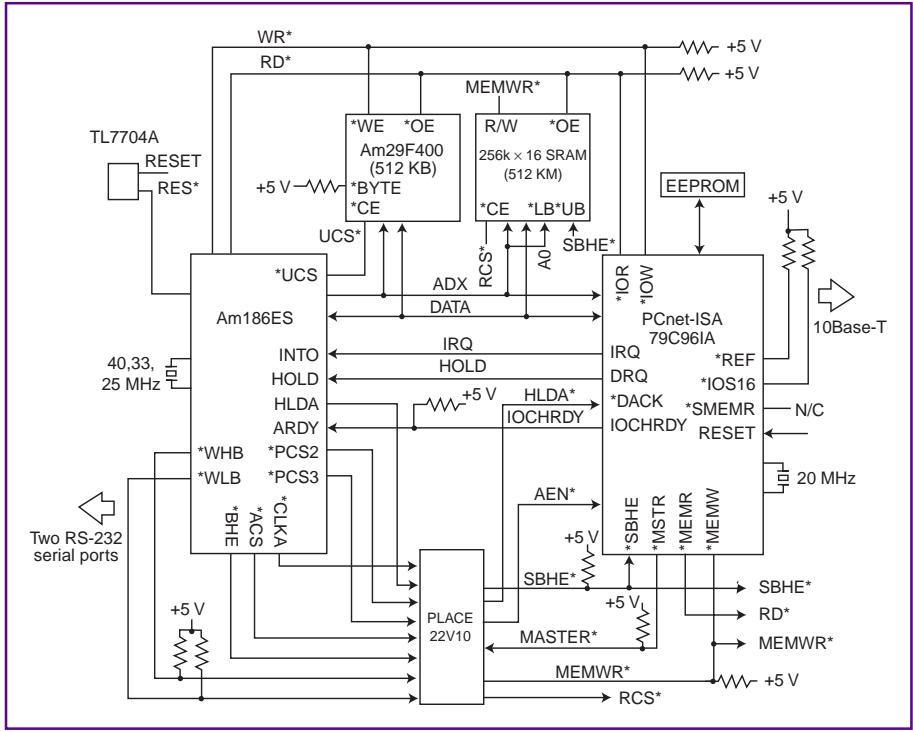


Figure 1—Lots of stuff is left out, but the important thing to note is the lack of glue.

or off on the Net186 board. The Net186 application does this a little differently than the emWare implementation.

Instead of using objects, the Net186 app sends a request with an attached parameter. A CGI (Common Gateway Interface) is present at the Net186 and accepts the parameter.

The state of the selected LED (which is attached directly to the Am186ES programmable I/O pins) is determined and toggled. A new Web page is then built around the toggled state and returned to the browser. The idea here is to convey the ability to control anything over Ether-or-Internet.

Access to the Ethernet can be accomplished via telnet and the serial ports or by

using its onboard Ethernet capability. The serial process is as usual.

On the other hand, the Ethernet implementation is rather unique in that the PCnet-ISA II Ethernet controller uses DMA and the processor local bus to transfer packets directly into SRAM.

The Ethernet controller can be memory mapped or I/O mapped. The Net186 board allows either. The four LEDs indicate the status of the PC-ISA II Ethernet controller interface.

If you decide to I/O map the Ethernet controller, you must sacrifice two pins on the second async port. The Ethernet controller is mapped in I/O space 0x200–0x21F on the Net186 card. Figure 2 is a representation of the PCnet-ISA II Ethernet controller.

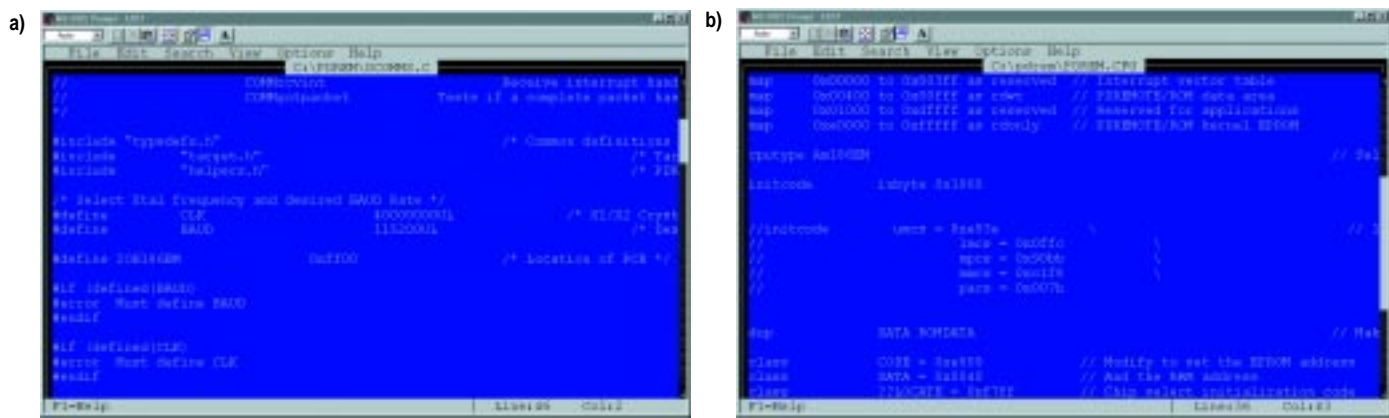
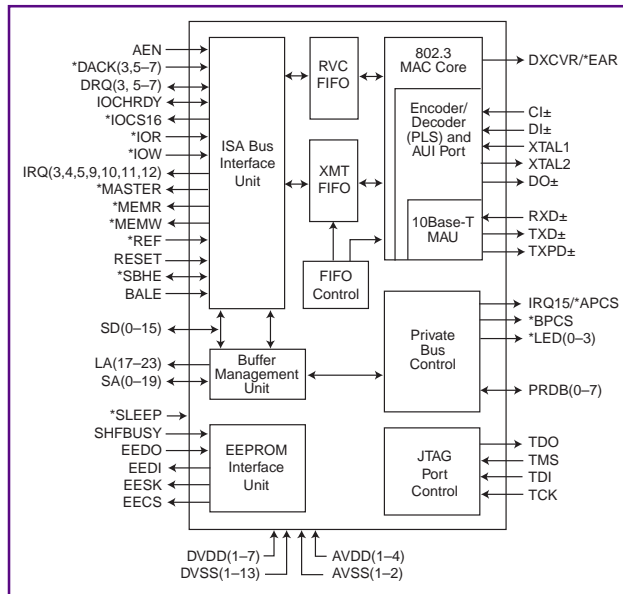


Photo 2a—Setting the baud rate is a piece of cake. b—Notice the *init* code is commented out. The Net186 start-up code does this initialization.

Figure 2—This is too neat. I gonna come back to this part when I have more pages.



E86MON

For those developers wishing to evaluate the AM186ES and its peripherals, E86MON was written just for you. E86MON is a software utility included with the Net186 system that allows the downloading and execution of application code.

E86MON also can assist in basic debugging. Applications can be downloaded to SRAM or flash memory. There's even a limited DOS emulator that can execute small .EXE files written in your favorite language.

I could take up a lot of space describing the detail of every E86MON command, but that would absolutely bore most of you.

There's nothing special about its command structure. E86MON has the usual DUMP and DISPLAY commands that all good monitors should have.

One interesting quirk—if you want to call it that—is that Intel hex files are used exclusively by E86MON. The trick is to compile your file to a .EXE and use the EXE-to-hex utility to do the conversion.

E86MON sees the colon that begins all Intel hex files and knows to go into download mode.

CHANGING THE PARADIGM

While E86MON is fun to play with, a serious developer doing real work would have to get things cooking another way. Fortunately, the Net186 folks realized this and allowed third-party vendors to apply their wares to the Net186 architecture.

As I was perusing through the box full of docs that came with the Net186, I came across a familiar name—Paradigm. Wait a minute. I think I have this stuff in the Circuit Cellar Florida Room library.

I read through the requirements and sure enough. Paradigm Locate, check. Paradigm PDREM, check. Paradigm Debug/RT-186, check. Microsoft C/C++ V.1.5, check. Microsoft Assembler 6.1, check. Net186 demo board, check. Let's go!

The first order of business was to load Bill's C compiler and his assembler. No problem. One CD-ROM and five diskettes later, C and Assembler are done deals.

Next, I installed Paradigm Locate. While the disk is spinning, you probably want to know what Locate is and what it does for us. As its name implies, it's a locator.

Locate gets its input from a relocatable .EXE file that results from the compile and link process and assigns memory addresses defined in the .CFG file. This action produces what is termed an .AXE file that has embedded written all over it. Figure 3 shows you what I just said.

OK, the diskette drive light is out, so back to the Locate install. The first thing I was asked to input to the install procedure was what compiler if any I was using. C/C++ was not an option, but C++ V.8 was.

Well, as it turns out, C/C++ 1.5 is the same thing as C++ V.8. Go figure.

The next step was to define a library directory. This directory is used to house the ROMable run-time libraries. The libraries in this particular directory are scanned during linker time to build the ROM object file.

If you set up your pathing and environment variables correctly, these libraries are built during Locate install time by Bill's C/C++ compiler. If you don't, like I didn't, you can make them after the fact with a batch file included with Locate.

I ran into one problem here. Seems that some of the LIB files didn't transfer during the C/C++ installation. The Locate installer program bombed out with an error.

So, after dropping a couple of bombs, I simply inserted the C/C++ CD and copied all of the LIB files over. Problem solved, Locate installed.

PDREM, short for PDREMOTE, is next on the install list. PDREM is Paradigm's target system compliment to Debug/RT-186. PDREM serves as the eyes and ears of Debug.

The first thing PDREM wants to know is what processor will be used. It was really good to be able to select the exact piece of hardware you are actually using. As you know, sometimes it's a guessing game between the software switches and what they really mean and really do.

Because PDREM is the communications channel, the data rate for communications between the debugger and the target system is set in this install. This is done by setting a parm within the dcomms.c file in the PDREM directory as shown in Photo 2a. Now to install the debugger, Debug/RT-186.

After setting the baud rate and COM port to match the dcomms.c configuration, Debug/RT-186 installed before I could finish this paragraph. At this point, I made sure everything was correctly inserted in the path statement of the AUTOEXEC.BAT file.

OK. Everything is installed. It's time to build and download PDREM. I will ultimately end up with a pdrem.hex file that

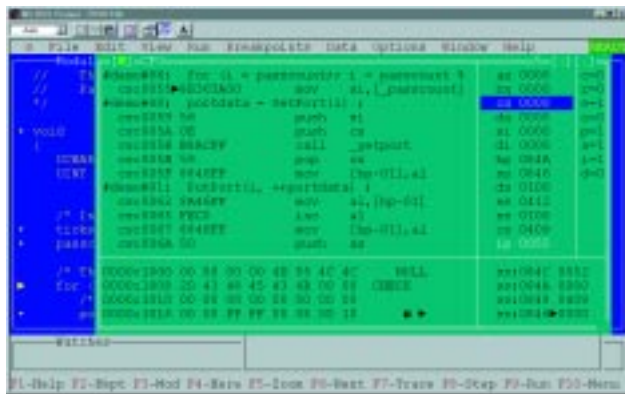


Photo 3—Thank you, Rick! Bananas on the banana tree.

is downloaded to the target flash. This will provide a high-level source and data debugging environment for the Paradigm Debug/RT-186.

The first modification is done on the target.h file in the PDREM directory to define interrupt-driven I/O. This was done by changing the COMMUNIT parm to 0x14 to define the interrupt vector. The next file modification was done on the PDREM.cfg file.

Basically, all of the Net186 memory mapping was defined here as you can see in Photo 2b. After correcting a minor mistake, (taking the underscore out of MASM_611 in the path statement) I was able to realize a pdrem.hex file in the PDREM directory. This is good.

The next step in the process is to download the pdrem.hex file to the Net186 flash memory. As the Web-server demo is residing in the flash application area, it would be a good idea to remove it before I download the new PDREM code.

A connect with HyperTerminal and a quick X A command does the trick. Time to download the pdrem.hex file and fire up some debugging screens. Not!

Well, its been a couple of hours and more compiles, tests, and downloads then I want to talk about. All I get is Remote Link Timeout. OK. Been here before. Time to

call Paradigm support. Maybe I'm missing something.

Rick at Paradigm says, "Hmm... Fred, never heard of that problem. What's your E-mail address? I'll send you some code."

OK! I'll be back in a minute after I check my mail. Stay put.

First of all, I received a package of code from Rick at Paradigm that contained a different pdrem.hex file. Seems that this pdrem file uses the other async port and flashes the LEDs violently to let you know something is going on inside the silicon. The pdrem.hex I built didn't indicate that anyone was home.

Rick also included a load program to put the pdrem.hex file on the Net186 board. No difference there.

NETTING IT OUT

Not only have you had kid's day at work with Dad, you have been introduced to a new hardware tool and a couple of equally nifty software debugging tools for the new toy.

Photo 3 says it all to any of you who've ever written code. The Net186 package also includes all of the necessary engineering drawings to implement your own product using the AMD parts as a reference. And as you saw, technical support for the products is very good.

I don't know which bird (Mom or Dad) teaches the new ones to fly, but I'm gonna push you out of the nest. Hopefully, my experiences will give you a head start on developing your AMD186 appliance.

You know, I tried my best to complicate this. But in the end, it wasn't complicated and it is embedded. **APCEPC**

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

Net186
AMD
(800) 222-9323
(512) 385-8542
Fax: (408) 749-4753
www.amd.com

Locate, PDREM, DEBUG/RT-186

Paradigm Systems
(607) 748-5966
Fax: (607) 748-5968
www.paradigm.com

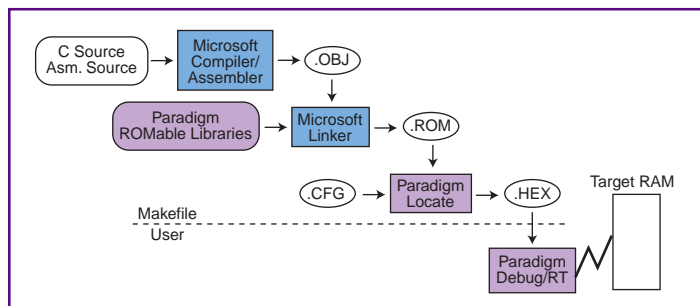


Figure 3—As Spock would say, "Logical."

FEATURE ARTICLE

Bobby Crouch

Designing for Smart Cards

Part 2: Practical Implementation

The difficulty in smart-card design always hinges on security. After showing us how to implement transactions, Bobby walks us through the memory authentication procedures that keep the cards shipshape and secure.



A typical smart-card session involves some form of transaction, such as a purchase, information transfer, or perhaps authorization for entry or access.

In Part 1, Carol Fancher presented smart-card basics and the history of the industry. This month, I consider the software, command structure, and security concerns of a smart-card developer.

I first discuss some of the software coding considerations involved in smart-card transactions. Later sections detail some common ISO 7816 commands used in smart-card transactions.

Most technical information from each smart-card silicon vendor is proprietary, and the considerations pertinent to secure technical information focus on hardware security features. Although the CPU cores are the same as their nonsecure counterparts (e.g., 'HC05 or 80C51 microcontrollers), the other silicon implementations that improve security must remain secret.

The software development tools are also secretive in nature, although vendors do give out memory size (RAM, ROM, and EEPROM), operating voltage, and, of course, pricing information. As another obstacle, the U.S. government requires export licenses for smart-card devices using cryptog-

raphy (these devices fall under the control of armament laws).

PHYSICAL CONSIDERATIONS

To implement a smart-card system, first consider the size of the pertinent changing data, which will be in the EEPROM, and the size of the application software or OS, which is implemented in ROM. Both are commonly touted features of smart-card vendors.

The price of the silicon is proportional to the EEPROM array, but like other devices, smart cards are evolving into larger EEPROM array sizes and cheaper prices. Most vendors are currently using 0.6–0.8- μm geometries and die with areas less than 25 mm².

The physical resources of the silicon should be carefully considered when developing the functions of the OS or application software. Specifically, think about the RAM.

How many levels deep will subroutines or interrupts go into the stack? If software doesn't use the entire stack, those free RAM bytes are available to the OS.

Cryptographic algorithms use large amounts of RAM, affecting the selection of silicon (and the algorithm), as well as the number and size of pertinent RAM variables. The I/O buffer of received and soon-to-be transmitted data is likely to consume the most RAM.

SOFTWARE DEVELOPMENT

Once the nonvolatile memory size is agreed on and a vendor chosen, you can begin software development. Over the past 10–15 years, OSs in smart cards have evolved from simple, single-application cards to current GSM SIM cards with multiple applications (see Figure 1).

The goal of the OS is to ensure a simple application that can be used anywhere. The software must be essentially error free because it is commonly implemented in ROM and, thus, is expensive and timely to repair.

The OS needs to provide communications with card readers, execute only predefined ISO commands (as pertinent to the card's application), manage data and authorization, and ensure execution of encryption algorithms.

SYSTEM SECURITY

Obviously, access to the smart card should be limited only to authorized users. But, that's only the first level of security for the total system.

The software should ensure that external, logical addresses aren't the same as the card's physical address. This interpretation of logical and physical addresses should be coded into the OS or application software, so an intruder can't breach the first layer of card security, issue an ISO read command, and access sensitive information.

MEMORY MANAGEMENT

Another way to provide file security is via a memory manager, which limits access to other memory areas by a running application or software. Various vendors call these features by different names (MMU, CCMS, etc.).

These features enable you to select areas that an application may reside and operate in, as well as exclude other areas from accessing that particular application. Essentially, an application assigned a certain memory space is also prevented from reading or modifying another application's space or data.

With cryptographic keys, for example, the card issuer can protect or prohibit access of this particular area of memory by any software or application other than a unique, authorized file or program. Enhanced memory-management features may permit dynamic allocation or updating of memory areas and their respective applications.

Memory management units (MMUs) are essential for multiapplication OSs, where dynamic loading of new tasks in Java or other interpreted languages is possible. In these environments, memory and address management ensures that applications from unrelated tasks (e.g., loyalty and electronic purse) don't interfere with each other, even if bugs exist in the task or OS code. It also limits ways that hackers can inject uncontrolled behavior into a smart-card device.

Memory management developed in sophisticated 32-bit environments, where operating systems such as Unix and Windows demand a way for hardware to isolate tasks. But in these environments, issues like silicon area, power,

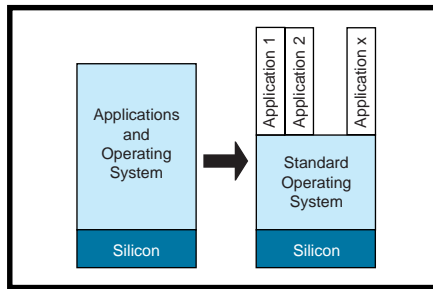


Figure 1—The memory needs of multiapplication cards are much greater than for single-application cards, and so are the needs to isolate each application from the others.

and execution time are less important than in the smart-card environment.

Smart-card memory management must combine the best concepts of existing MMU strategies within a stringent silicon, performance, and power budget if such solutions are to be practical in the high volumes and low costs demanded by distributors.

Another way to limit access to secure data is by introducing complex sequences to enable access to certain memory areas. Control register bit manipulation and other operation-sequence-dependent methods make a hacker's job much more difficult and time consuming. They also prevent rogue code from accessing data it shouldn't, even if it's trusted code like an operating system.

Currently, several system providers offer smart cards with Java operating systems onboard. The security of Java on a smart card is enforced by virtue of the Java Virtual Machine (JVM), which prohibits program access to anything the JVM hasn't allocated.

These cards load the JVM in ROM and permit Java applets to be loaded and run in EEPROM. This type of data security enables reuse of the card for different applications.

An added benefit to Java-based smart cards is the ability to load applications on smart cards from various vendors. No longer are applets tied to a specific manufacturer or architecture.

The I/O channel is a prime target for hackers, so be careful when selecting what data to transmit across the channel. It would be simple to solder a wire, record the transmitted data, and examine it for clues.

Also, design the data so that an attacker with access to the I/O channel would not be able to obtain any valuable

or secret information. In an electronic purse, for example, the balance of an account or the transaction amount aren't secret, but account numbers, keys, and passwords shouldn't be transmitted (in the clear) across the I/O channel.

The card's software should allow only necessary commands. Commands that aren't in the card's instruction set should return the appropriate ISO error code and reset the card after several attempts by a card reader to issue invalid commands.

Basically, that's the default procedure for any illegal action—just reset the card.

ISO COMMANDS

The ISO commands are explained in ISO 7816-4, which is similar to other data-transmission specifications like those for OSI 802.2 and 802.3. Due to the relatively simple environment of smart cards, 7816 is easy to understand.

The types of commands contained in the standard include file selection, file reading and writing, file searching, file operations, identification, authentication, file management, program execution, and special instructions only implemented in an individual application. Obviously, not all commands are necessary for every application.

Careful thought should be given to the transaction process, as well as a system of checks that the process transpired properly. Transactions concern user identification, verification of the user's account or access privileges, the actual transaction, verification that the transaction was successful, and verification that a history entry was made of the transaction.

One important point is that EEPROM bytes are cleared to their erased state prior to being written (this may be improved by only writing or erasing bits that need changing). Intruders shouldn't be allowed to remove power in the middle of an EEPROM write sequence and thus alter error counters or other sensitive variables. So, any EEPROM updates shouldn't be signaled to the reader until after they complete.

SYSTEM TYPES

There are two types of smart-card systems—open and closed. In an open system, the card may be used for transactions between many operators. In

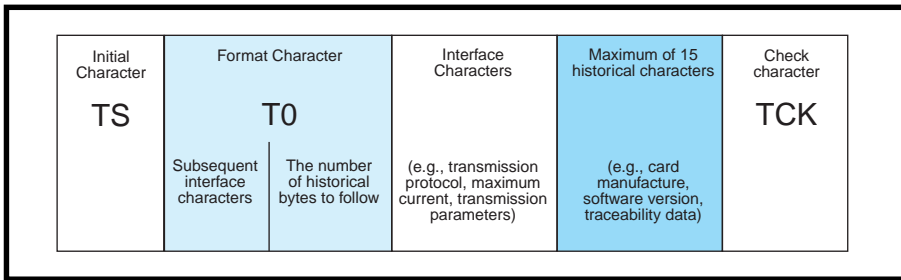


Figure 2—The ATR string, which has a maximum of 33 bytes, is broken down into five main sections.

closed systems, the cards are used with one operator (usually the issuer).

Examples of an open system include phone cards and electronic purses. In those cases, the card establishes a unique identity and eligible services for use with a phone system or for making debits at various point-of-sale terminals.

The simplest case—a closed smart-card system—involves one type of card and one type of reader (e.g., a card used with one mass transit carrier). And, there is a known identification procedure between them, which can be implemented in the answer-to-reset (ATR) string.

The ATR is the first message in a smart-card session with a reader and is sent by the card immediately after powerup (provided that a clock is supplied along with the power). The ATR message contains data that defines relevant transmission protocols and other information.

ISO defines the ATR to have a maximum length of 33 bytes, although this length is seldom reached. As you see in Figure 2, the ATR consists of:

- an initial character, TS
- a format character, T0
- several interface characters
- a maximum of 15 historical characters
- a check character, TCK

TS defines the conventions to code data bytes in all subsequent transmissions. There are two conventions—direct (\$3B) and inverse (\$3F). All terminals can read both, and most card manufacturers are capable of both.

The format character consists of two parts. One part indicates the subsequent interface characters, and the other indicates the number of historical bytes to follow (1–15).

Information transmitted via the interface characters includes the transmission protocol (e.g., T=0 or T=1), the maximum current the reader should supply to the card, and any other transmission parameters pertinent to communicating with the card.

The historical characters may contain any information, such as the manufacturer of the card, software version on the chip, or traceability data. The number of historical bytes is indicated in the least significant half of the interface character.

The last byte of the ATR string is the check character. This byte is defined by ISO 7816-4 to be “a value such that the exclusive ORing of all bytes from T0 to TCK, included is null.”

SMART-CARD SESSION

After the identification of the card and reader is completed and the proper transmission protocol established, the session begins. The initial procedures should include authentication of the reader and card, the granting of access privileges to data areas, and the actual transaction.

Authentication is the act of validating the reader (or user) and granting file access. File security can be as simple as assigning file-access authority at card personalization or as complex as the more comprehensive memory-management schemes.

File-access rights can consist of defining the file structure and authorization policies. This can be performed explicitly at the vendor’s factory or at the card issuer’s location.

Data access may be contingent to the reader asking for, and passing, a challenge. A common challenge scenario is for the reader to issue the command `get challenge` to the card, and the card then sends a ran-

dom number to the reader for manipulation.

Manipulation may consist of encrypting or decrypting the data with an algorithm that uses a fixed key, set of fixed keys, or changing keys that are coordinated with keys on the card. Once the manipulation is performed, the result is sent to the card for comparison.

The card compares the reader’s result with its own and, if they match, grants further communication to the card. Certain predefined files (whose access rights were defined at personalization) may necessitate further authentication (e.g., user PIN or password) for access. Once identification and authentication are satisfied, a transaction commences.

As an example, consider a debit-card application. The basic transaction information—identity and authorization—are the first steps. A transaction amount is selected, followed by some form of signature (e.g., signature algorithm or other authentication) from the requesting device.

Once the desired transaction type and amount are selected, funds are debited or credited appropriately. This task involves writing to EEPROM to record the transaction in a history file and modifying the correct account.

If the requested transaction is a purchase from a reader, the crediting message should also include a signature. The last step is to confirm the transaction.

In this example, the signature authenticates each critical step of the transaction process. The complete data set is digitally signed, so it cannot be altered during transmission.

You should consider implementing various checks for each step of the transaction. These checks may include control bits or bytes of EEPROM variables that verify each step of the transaction process (because of the critical nature of the EEPROM write sequence mentioned earlier).

Also, think about possible transaction failures or hacking attempts. What happens if the card is withdrawn from the reader during the transaction? What happens if the reader fails to supply the proper power voltage to the card?

What happens if the reader’s encryption of the random number is

incorrect? How many times should the card accept authorization attempts, and what action should be taken when the allowed number of attempts is violated?

How does the card or reader know if a transaction completes properly? Also, what action should be taken, and by what part of the system, if there is a transaction failure?

A hacker may try to cheat the system by withdrawing the smart card after a transaction but before it could be recorded onto the card. Put simply, authorizing any transaction should be the last procedure of a smart-card session. Recording procedures onto the smart card should be completed prior to external verification of a transaction.

One possible method of monitoring a transaction's progress is to define a transaction status byte in EEPROM. Different values indicate transaction in progress, failed transaction, successful transaction, last successful transaction, last unsuccessful transaction, and place of failure of the last transaction. This information may require more than one EEPROM byte.

Monitoring this transaction control byte should be the first and last acts of a smart-card session. Certain states then reflect successful completion of various stages of a transaction.

Most silicon vendors implement a manual or automatic programming of EEPROM. Possible hacker attacks include reducing the power level (V_{dd}) to the card during critical (EEPROM programming) stages of a transaction.

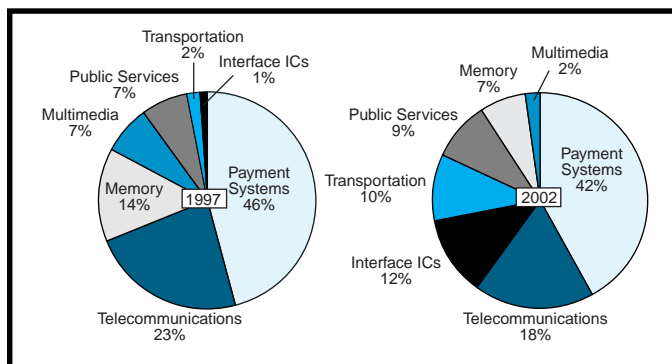
The manual or automatic programming of EEPROM is acknowledged by some event that the smart card can check. It should be included in every EEPROM programming procedure.

Implementing this check in the EEPROM procedure helps defeat low-voltage attacks. The transaction control byte is the smart card's check on every transaction and should be included in the software.

Possible reactions to a fraudulent reader—one that fails the authentication procedure—should include a reset of the card. There should be a limit to the number of times a smart card fails authentication. This would likely be a RAM variable since no transaction takes place prior to authentication.

Of course, there should be allowances for one or two failed attempts, possibly

Figure 3—These are the current and projected market needs for smart-card applications. Clearly, purse applications are the dominant need.



due to reader software failure. However, at some point, the smart card should shut down and refuse further communication.

Some smart-card features may not require authentication—for example, the wallet account of an electronic-purse application. Small transactions are normally paid for with pocket change or small-denomination currency can be completed much faster with a smart card.

The wallet account of a smart card can be limited to a small, fixed amount, which is similar to the amount of cash you're comfortable carrying with you. The risk to this account, just like the risk to your wallet, is that if you lose it, you'll probably lose all your cash.

However, you can immediately cancel your credit cards and prevent unauthorized use. Similarly, a smart-card wallet could contain an account intended for small purchases or transactions. This account should allow a check of recent transaction (history), as well as the current balance.

More critical and sensitive areas of the smart card (e.g., credit/debit accounts or sensitive personal information) should only allow access after authentication of one or more levels. Such purse accounts are commonly checked by small, key-ring-size readers.

The primary elements of a smart-card transaction are card power-on reset, card/file authentication, transaction type selected and approved, transaction history parameters initialized and checked (for prior incomplete transactions), transaction effected on the card, transaction history parameters modified (to signify a successful transaction), transaction approval sent to the reader, and transaction terminated.

All of these actions may be formulated in pseudocode and then modified to the actual instruction set of the chosen smart-card microcontroller.

GETTING SMARTER

Developing a smart-card system involves all of the points I've discussed—and many more. I didn't even talk about the infrastructure of the

reader network, reader software, the card manufacturer, graphics, or the silicon module.

But, you now have some more ideas about designing solutions into smart-card programs at an early stage of development, when the cost is low.

Smart cards offer incredible security and can greatly minimize fraud and theft, so they're a valuable and cost-effective solution to many applications.

Figure 3 illustrates the shift in use of smart cards that is expected over the next few years. Many of today's single-use applications are likely to be cost effectively incorporated onto future multiapplication smart cards.

▣

Bobby Crouch is a smart-card applications engineer at Motorola. You may reach Bobby at r23001@email.sps.mot.com.

SOURCE

ISO 7816 and other standards

International Standards Organization
www.iso.ch

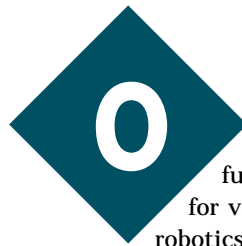
FEATURE ARTICLE

Don Lancaster

Vector-to-Step Conversions

An Introductory Tutorial

Whether in video, graphics, or automation, vector-to-step conversion is a critical computing need. Don takes a look at some of the factors that enable you to keep those steps as fast and smooth as possible.



One of the most fundamental needs for video displays, robotics, or laser printers is vector-to-step conversion. That's where a random line or motion direction gets broken down into the precise number of needed pixels or incremental robotic steps.

As Figure 1 illustrates, a vector can input at any arbitrary angle. The needed output steps get locked into specific x and y positions.

There are several methods for handling vector-to-step conversions. Extensions of these algorithms enable you to do higher dimensions, curved paths, and circles, and they even provide for image or object rotations.

The method you pick depends on your choice of language, programming skills, available system resources, and the speed of operation. In 2D or 3D animated graphics, speed might be of utmost importance, but it wouldn't be such a big deal on a wood router.

A high-level language offers ease of programming and end-user friendliness. A lower level one usually increases speed and reduces costs but it severely limits your use of fancy math or trig functions.

In a low-end PIC robotic application, minimizing memory space or costs might dominate your decision making.

BASIC CONVERSION

It's often simplest to break the 2D vector-to-step conversion process down into the eight cases of Figure 1. Solve one of them, and the rest fall in place.

Consider octant 0, which goes from 0 to 45°, and assume a process that accepts x and y values as inputs and provides discrete pixels or locked mechanical steps as outputs.

In octant 0, the value for x is always positive. Your value for y should also always be positive. And, x always ends up greater than or equal to y .

In this octant, you always step by x . You may step by y if you get less error.

One solution is to always take the next x step. Then, measure your errors of stepping by y and of not stepping by y , and then take the result that gives you the error with the lowest absolute value.

Alternatively, you can step by one half of y and see if you end up above or below the required vector. If you're low, step both x and y . If high, step only by x .

You end this process when you use up all the needed x steps. The other seven octants are similar, except that y might dominate x or negative values may be involved.

Any vector-to-step routine might create positioning and closure errors caused by those discrete steps. Your high-level software should track any fractional pixels, as well as accommodating fancier options, such as grid locking or adjusting optical widths.

BRESENHAM'S ALGORITHM

I was happy with my vector-to-step routines until I found I was clumsily and inefficiently going over well-plowed ground. An often optimum solution is known as Bresenham's Algorithm [1].

By creating a double-sized error value, only simple adds and shifts are needed for calculations. The double-sized error function lets you test for a simple sign rather than for a half-unit change.

In octant 0, first calculate the error value of $2y - 2x$. At each step, test and modify your error value. Then, decide where to go:

```
IF e < 0
  THEN e = e + 2 y
ELSE e = e + 2 y - 2 x
STEP x
```

IF $e \geq 0$
 THEN STEP y

This process continues for your needed number of x steps. As before, the other octants are handled alike, except that the signs and roles of x and y change.

You can find a lot of language-specific examples of this algorithm on the Web, especially for Java, C++, and PICs. One version requires seven machine cycles per x pixel plus 31 overhead cycles.

Thus, a 16-step Bresenham conversion might take 144 machine cycles.

You can also find extensions to Bresenham's Algorithm. One lets you rotate an image without using trig functions by taking each scan line and remapping its position. Another variation draws circles by use of an ancient digital differential analyzer scheme.

AN EXAMPLE

Let's look at a somewhat detailed Bresenham example. Say you decide to travel east by 10.134 pixels and north by 3.65 pixels.

Because you can only work in whole pixels, it's a good idea to shoot 10 over and 4 up in quadrant 0. Save the 0.134 and -0.350 as spare change somewhere to prevent error pileups.

First, calculate and save the initial error value of -12 (i.e., $2 \times 4 - 2 \times 10$). Also calculate and save $2 \times y = 8$. Note that you can multiply by two simply by doing a left shift. Therefore, you need to start with an error value of -12 .

If your error value is negative, add 8. If your error value is 0 or positive, subtract 12. When your new error value is positive or 0, step both x and y . If the new error value is negative, step only by x . Continue for the needed number of x steps. It looks something like:

$\epsilon = -12 + 8 = -4$ $\Delta y = 0$
 $\epsilon = -4 + 8 = 4$ $\Delta y = 1$
 $\epsilon = 4 - 12 = -8$ $\Delta y = 0$
 $\epsilon = -8 + 8 = 0$ $\Delta y = 1$
 $\epsilon = 0 - 12 = -12$ $\Delta y = 0$

$\epsilon = -12 + 8 = -4$ $\Delta y = 0$
 $\epsilon = -4 + 8 = 4$ $\Delta y = 1$
 $\epsilon = 4 - 12 = -8$ $\Delta y = 0$
 $\epsilon = -8 + 8 = 0$ $\Delta y = 1$
 $\epsilon = 0 - 12 = -12$ $\Delta y = 0$

The final x step pattern is 1111111111. The y step pattern is 0101001010, which exactly agrees with my clumsier method.

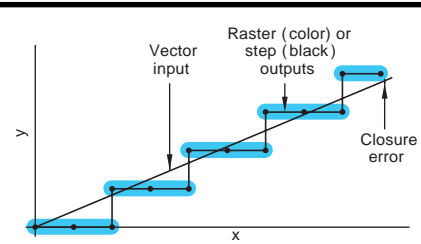
TABLE LOOKUP

Can Bresenham be beaten at his own game? His solutions are fast and extremely compact. Many programmers have spent a lot of time further optimizing them.

In theory, a simple table lookup of the pattern you need for a given x and y might end up faster and simpler. On the other hand, variable length words and additional storage space could be required, as might the overhead of step extraction. The table length depends on the maximum number of pixels or steps to be handled.

Figure 2 is a 2D PostScript array that has all of the needed vector-to-step patterns for x equals 0 up through x equals 16. Longer vectors are done by repeated looping until the input vector is completely used up.

For quadrant 0, enter with x on stack top and y immediately below. The command:



Correct the input vector so it starts at your actual initial position, which prevents closure errors from piling up. Resolve your input vector into x and y components. For a vector of length Z and an angle θ :

$$x = Z \cos(\theta)$$

$$y = Z \sin(\theta)$$

Calculate the slope y/x . Round x and y off to the nearest integer values to get the actual steps needed. Compare the signs of x and y , and the absolute sizes of x and y to find an octant:

$x+$	$y+$	$x>y$	octant 0	(0–45°)
$x+$	$y+$	$x<y$	octant 1	(45–90°)
$x-$	$y+$	$x<y$	octant 2	(90–135°)
$x-$	$y+$	$x>y$	octant 3	(135–180°)
$x-$	$y-$	$x>y$	octant 4	(180–215°)
$x-$	$y-$	$x<y$	octant 5	(215–270°)
$x+$	$y-$	$x<y$	octant 6	(270–315°)
$x+$	$y-$	$x>y$	octant 7	(315–360°)

In octant 0, always step by $+x$ and sometimes step by $+y$. In octant 1, always step by $+y$ and sometimes step by $+x$. In octant 2, always step by $+y$ and sometimes step by $-x$. In octant 3, always step by $-x$ and sometimes step by $+y$. In octant 4, always step by $-x$ and sometimes step by $-y$. In octant 5, always step by $-y$ and sometimes step by $-x$. In octant 6, always step by $-y$ and sometimes step by $+x$. In octant 7, always step by $+x$ and sometimes step by $-y$.

Figure 1—Here are some vector-to-step fundamentals along with a brute-force algorithm.

`vspat exch get exch get`

delivers the ASCII string pattern to the stack top.

As you see, using a PostScript table lookup is simple and fast. The output is a string optimized for robotic flutter-wumper uses.

The size of the lookup tables depends on whether you use pattern bits or ASCII

```

/vspat [ [(0)]
[(0)(1)]
[(00)(10)(11)]
[(000)(010)(101)(111)]
[(0000)(0100)(1010)(1101)(1111)]
[(00000)(00100)(01010)(10101)(11101)(11111)]
[(000000)(001000)(010010)(101010)(110101)(111011)(111111)]
[(0000000)(0001000)(0100010)(1010101)(1101011)(1110111)(1111111)]
[(00000000)(00001000)(00100010)(10101010)(11010110)(11101110)(11111110)]
[(000000000)(0000010000)(0001000010)(1010101010)(1101011010)(1110111010)(1111111010)]
[(0000000000)(000000100000)(000010000100)(010010001010)(101010101010)(110101101010)(111011101010)(111111101010)]
[(00000000000)(00000001000000)(0000000010000)(01000000100010)(10101010001010)(11010101001010)(11101011010101)(11111011010101)(11111110101010)]
[(000000000000)(0000000010000000)(0001000000100000)(0010000000100000)(001000010000010000)(010000010000010000)(1010101000010010)(1101010100010010)(1110101101010101)(1111101101010101)(1111111010101010)]
] def

```

Figure 2—This 2D PostScript vector-to-step table lookup is of order 16.

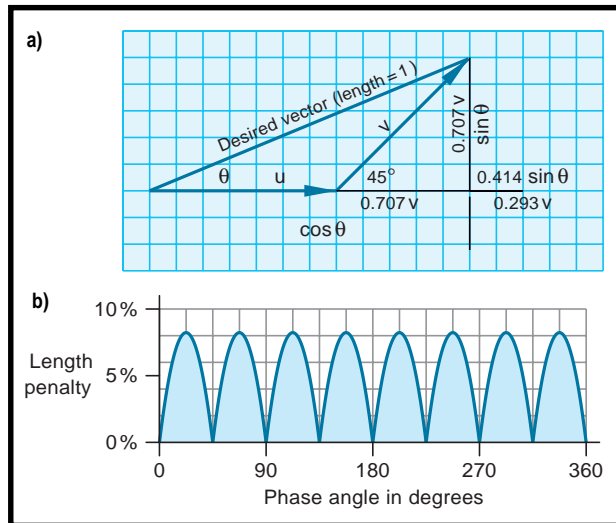


Figure 3—Cardinal moves can sometimes simplify low-end robotics systems. Here is the math behind the minor speed penalties involved. **a**—Assume a quadrant zero vector of length 1 and angle θ . Approximate this with an east move of u and a northeast move of v . The vertical rise will be both $0.707v$ and $\sin\theta$. Thus, $v = 1.414 \sin\theta$. Extend the baseline by $(1 - 0.707)v = 0.293v = 0.4140 \sin\theta$. The baseline will now be $u + v$ long and equal $\cos\theta + 0.4140 \sin\theta$. The excess length (and time penalty) is $\cos\theta + 0.4140 \sin\theta - 1$. **b**—Extending and plotting produces this error curve.

values, as well as on overhead and how efficiently you pack odd pattern lengths. However, tables grow at an n^3 rate.

The minimum bit table size for a 16-pixel lookup is 1632 bits or 204 words of eight bits each. This table fits into the smallest of PIC, but it might be a real challenge to access.

For 32 pixels, allow 11,968 bits or 1496 bytes. For 64 pixels, you need 87,360 bits or 11,977 bytes. Beyond this point, look-up sizes get out of hand.

Breaking a longer vector into short ones may introduce minor placement errors. While nearly all of these end up negligible, it's best to avoid the worst case of a vector one more than a multiple of the table length (17, 33, 49, etc.).

Instead, try a different split. For instance, an 8 then a 9 lookup may give you modestly more positioning accuracy than a 16 then a 1 lookup.

ADDING DIMENSIONS

Vector-to-step conversions are easily extended into the three axes needed for 3D animation rendering. They can even work up to the six or more axes used in fancy robotic moves.

In 3D, there are eight possible x dominant sectors of $x + y + z +$, $x + y + z -$, and so on through $x - y - z -$. Similarly, there are eight possible y -dominant sectors and eight possible z dominants, giving a total of 24 sectors. Each of these 24 sectors can be dealt with in the same way as the eight 2D octants.

For six-axis robotic motions, a two-step (coarse/fine) approach might be useful. Otherwise, use all 384 of the 6D sectors.

CARDINAL MOVES?

Let's wrap up with some fun. When you put pixels on a screen or machine a path, you want a smooth path.

On a simple move, you may only be able to travel in the eight cardinal directions. Such a restriction may simplify your code and shorten your file lengths—at least this is the case in several low-end flutterwumper systems I've worked with recently.

How much time penalty is there in positioning only in directions of E, NE, N, NW, W, SW, S, and SE? As Figure 3 reveals, the penalty is surprisingly small.

The worst-case scenario gives a tad over 8% at 22.5°. The average for all random positions is around 5%, and there is almost no penalty for the usual axis or near-axis moves.

Good luck in your future steps! 🍀

Don Lancaster is the author of 35 books and countless articles. Don maintains a U.S. technical help line at (520) 428-4073 and also offers books, reprints, and consulting services. You may reach him at don@tinaja.com.

SOFTWARE

A complete set of PostScript procedures for vector-to-step conversions are at www.tinaja.com/psutils/flutools.ps.

REFERENCE

- [1] J.E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, **4**, 25–30, 1965.

DEPARTMENTS

70

MicroSeries

78

From the Bench

82

Silicon Update

Build A Serial Port PROM Programmer

MICRO SERIES

Stuart Ball

Hardware Construction

Part
1
of
2

Stuart's low-cost PROM programmer

takes Intel-format and raw ASCII hex data, holds it in an internal 64-KB buffer, and programs it into an EPROM or PIC processor via a serial port. He starts off by focusing on hardware issues.



Do you need to program EPROMs or PIC microcontrollers, but you don't want to pay \$500 for a commercial programmer? Read on.

When I started writing *Debugging Embedded Systems*, I needed a simple embedded system to illustrate certain concepts. The project had to be practical as well, so I built a PROM programmer. My programmer had to:

- use an embedded processor. After all, the book is about embedded systems.
- connect to a PC serial port
- support intelligent programming algorithms
- use a simple command line interface, so it can be used with a standard communication program (e.g., ProComm), without requiring a complicated control program for the PC

The resulting programmer, shown in Photo 1, can program EPROMs from the 2764 through the 27256. It also programs 18- and 28-pin PIC microcontrollers, including the '16C61, '16C620/21/22, '16C62/63, '16C71/710/711/72/73, and the EEPROM-based '16C84. Other devices can be accommodated with appropriate code changes and new adapter modules.

The programmer accepts Intel-format hex data and raw ASCII hex files. Motorola format is not supported



Photo 1—Here's the completed programmer with the EPROM module installed and the PIC module beside it. There are no buttons or knobs. The entire thing is controlled from the serial port of your PC.

but could be added with code changes. The programmer has an internal 64-KB buffer to hold the programming data.

A commercial PROM programmer typically uses a single socket to program many devices. This task is accomplished by using pin drivers, which permit any pin to be driven to any voltage the programmer is capable of.

Pin drivers make a programmer flexible, but they add considerable cost. In the serial programmer, the device to be programmed is installed on a programming module that plugs into a 50-pin connector on the programmer PC board.

There are currently two modules—one for EPROMs and one for PIC devices. These modules connect the various programming signals to the proper pins on the device socket. Using a different module for each family of parts significantly reduces the cost and complexity of the programmer.

COMMAND LINE INTERFACE

The programmer uses a simple command structure, designed so any standard PC communication program can control it. When the programmer is turned on, the command list is sent to the PC. Table 1 lists the commands and their descriptions.

The #T command is followed by a two-digit hex code to select the device type.

The #R command sets an internal 16-bit variable to the specified value. This variable is added to the ROM address for all operations. The addition is limited to 16 bits. If the

result of the addition is greater than 16 bits (64K), it is truncated.

#M works the same as #R, except that the offset is added to the address for placing downloaded data into the internal buffer memory.

#S sets the size of operations on a device. If no #S is specified, operations take the size of the device. A blank check on a 2764, for example, will check the entire 8-KB device.

#R, #M, and #S permit data to be manipulated for special situations. For example, the upper 8 KB of data in a 27256 could be programmed by:

```
#S 1FFF
#R 6000
#P
```

The first line sets the operation size to 2000h (8 KB). The second sets the device offset to 6000h, and the last line programs the device.

To aid in developing new programming algorithms, these debug commands are supported:

```
#XM xxxx—displays 64 bytes of internal
memory starting at address xxxx
#XB xxxx—displays 64 bytes of buffer
memory starting at address xxxx
```

#P	Program the device with data in buffer
#B	Blank check the device
#V	Verify the device against data in buffer
#DI	Download Intel-format hex file to buffer
#DH	Download raw (ASCII) hex file to buffer
#R xxxx	ROM offset = xxxx
#M xxxx	Memory (buffer) offset = xxxx
#S xxxx	Operation size = xxxx
#T xx	Select device type xx
#F aaaa bbbb cc	Fill buffer from aaaa to bbbb with data cc
#L	List the available device types

Table 1—The programmer commands, listed here with their descriptions, are entered from your PC (using a communication program) to control programmer operation.

```
#XF—clears an internal rotating trace
buffer to all zeros
#XV—sets the Vpp and Vcc voltages to
programming levels
#XR—turns off Vpp and Vcc
```

The #Q command enables you to stop a download or a device operation (program, blank check, or verify).

Commands that require data (#F, #T, #S, etc.) fill with leading zeros. So, if you want to enter an address of 005A for the fill command, you can just enter 5A and the programmer fills in the upper eight bits with zeros.

PROGRAMMER HARDWARE

Figures 1–4 show the schematic of the programmer main board. The programmer is based on an 80C188 microprocessor, which is essentially an enhanced 8086 microprocessor core with a number of integrated peripherals, including three timers, a DMA controller, and a programmable chip-select decoder. The processor can access a megabyte of memory.

The 80C188 has a 16-bit CPU core, but the external bus is 8 bits. A 16-bit version—the 80C186—is also available.

For simplicity, I chose the 80C188 over the 80C186. The 8-bit data bus means that only one EPROM and SRAM are needed. The penalty for this choice is throughput, since word operations require two external bus cycles.

The 80C188, marketed for embedded applications, is manufactured by Intel and second-sourced by AMD. The programmer uses a single 27C256 EPROM for program memory, and a 128K × 8 SRAM. Half of the RAM (64 KB) is allocated for variables needed by the program, and half is used for the buffer to hold data to be programmed.

The 80C188 has a multiplexed address/data bus, which is a scheme used by a number of Intel processors. A pair of 74HC373 latches captures the lower eight bits and the upper four bits of the 20-bit processor address, which is presented during the first part of each machine cycle.

The EPROM and RAM are selected using chip selects

from the internal 801C188 chip-select decoder. UCS (Upper Chip Select) selects the EPROM. LCS (Lower Chip Select) selects the RAM.

Although the 27256 EPROM is $32K \times 8$, the firmware programs the chip select for a 64-KB space. Similarly, the chip select for the $128K \times 8$ SRAM is programmed for 256 KB.

The 80188 also has chip selects for peripheral devices, and these select the UART and 82C55 PPI. A 74HC138 completes the I/O selection logic by decoding one of the peripheral chip selects with address lines A3–A5 to produce write strobes for the DAC and the control register. Table 2 shows the memory and I/O maps for the circuit.

The processor uses a 14.746-MHz crystal, which is internally divided by two to produce a 7.37-MHz clock. This signal, available at the CLKOUT pin on the 80C188, also connects to the clock input on the UART.

I chose 7.37 MHz because it is an integer multiple of standard data rates. Higher frequencies provide more throughput, but the maximum 16550 input clock frequency is 8 MHz. I'd need an additional divider if I used a faster processor clock.

The microprocessor is reset by an LM393 comparator. The original 80188 had significant hysteresis on the reset input, so I could use a simple RC circuit. However, the 80C188 has less hysteresis, so an external circuit gives a more reliable reset.

Communication with the host PC is handled with a 16550 UART and a MAX232 TTL-to-RS-232 converter IC. An INS8250 UART also works because the firmware does not use the unique features of the 16550. Some versions of the 80C188 have an internal UART, but I wanted this device to be external.

An 82C55 PPI chip provides 16 bits of address and 8 bits of data to the device being programmed. Four bits of status are returned from the programming module via a 74LS244 and can be read by the CPU.

An eight-bit 74HC374 register controls a bicolor (red/green) LED to indicate status. The two LED leads each connect to one bit of the register, so the software can turn the LED off or display either red or green. The remaining six bits in that register provide additional control functions to the device being programmed.

PROMs require a high voltage (+12 V) for programming, and the intelligent

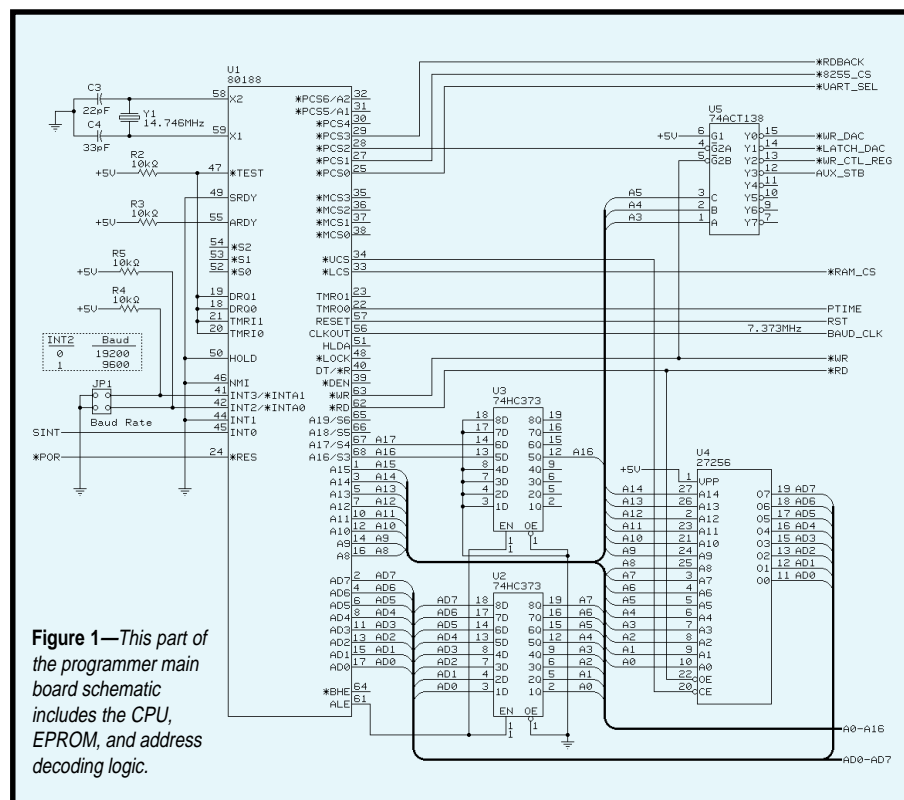


Figure 1—This part of the programmer main board schematic includes the CPU, EPROM, and address decoding logic.



Photo 2—The clear plastic strip to the left on the programmer's main circuit board is a hold down to prevent excessive flexing of the board when programming modules are installed.

programming algorithms require a variable V_{CC} supply as well. To accommodate this, the programmer uses a Maxim MAX505 quad 8-bit DAC.

One DAC output is used to drive a V_{PP} (programming voltage) supply from zero to about 22 V. A second DAC output provides the PROM V_{CC} supply, and can be adjusted over the same range.

The DAC outputs for V_{CC} and V_{PP} swing from zero to about 5 V, so they are amplified and buffered to provide the correct voltages at the programming connector. The remaining two DAC outputs connect to the programming connector for future use.

Each DAC has a holding register, so data for all four DACs can be loaded, one at a time. Then, the data is transferred to all the DACs with a single write command.

The DACs are 8 bits, and the DAC output voltage is amplified by 4.48. Consequently, the output voltage to the V_{CC} or V_{PP} pins on the 50-pin connector is given by:

$$V_{out} = \frac{x}{255} \times 5 \times 4.48$$

where x is the DAC control value.

A DAC value of 44h (68 decimal) produces a voltage of 6 V. The voltages can be set in increments of ~90 mV.

The programming connector is a 50-pin header, which provides all of

the programming signals, as well as +5 and +24 V from the programmer power supply. Each type of device to be programmed uses a programming module, which contains the device socket and plugs into the 50-pin header.

The programming module for the PIC devices has an additional header, consisting of two rows of 12 pins each. A socket plugs into this header to select either 18- or 28-pin PIC devices. This arrangement is necessary because the programming signals, including power and ground, are on different pins of the 18- and 28-pin parts.

INTERRUPTS

The programmer uses one of the 80C188 external interrupts, INT0. This interrupt connects to the interrupt output of the 16550 UART and notifies the processor when the UART is ready to send data as well as when data is received.

Two other 80C188 interrupts, INT2 and INT3, connect to shunt jumpers and determine the serial-interface data rate. INT2 and INT3 do not generate interrupts but are treated as input bits.

PROGRAMMING MODULES

The programming module for the 2764–27256 EPROMs consists of a 28-pin ZIF socket connected to the address and data lines from the pro-

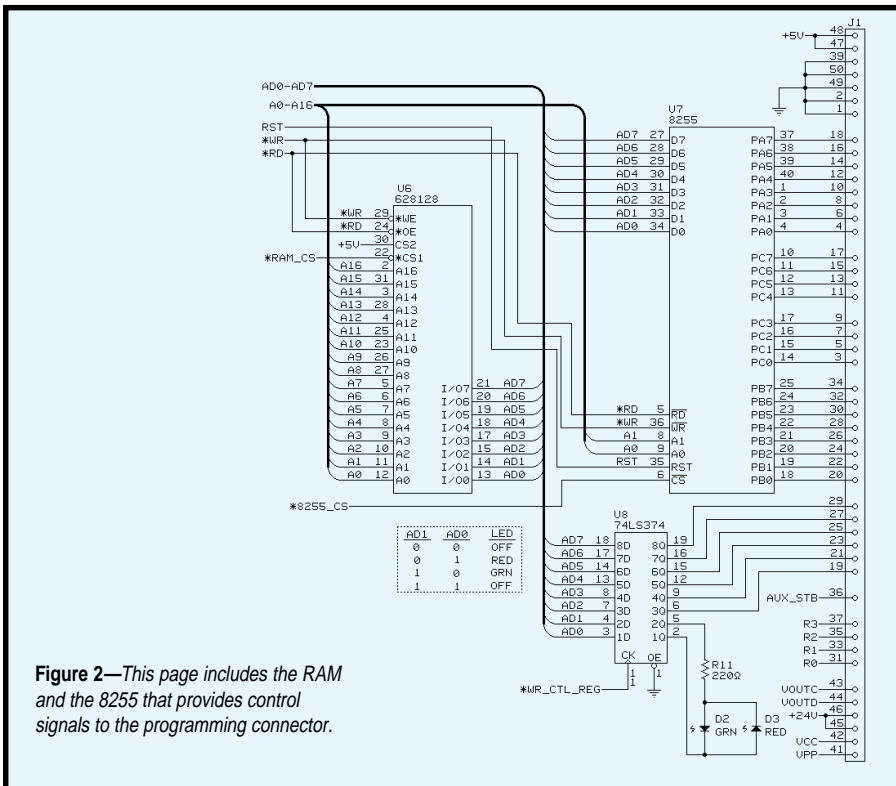


Figure 2—This page includes the RAM and the 8255 that provides control signals to the programming connector.

programming connector. Two LEDs indicate when V_{PP} and V_{CC} are active.

The PIC programmer module also uses a 28-pin ZIF socket. The PIC is programmed by serially loading address and data, so not all the programmer's address and data lines are used. Instead, one address line provides a clock for serial data, and one data line provides bidirectional data to and from the PIC.

Descriptions of the programming algorithms and schematics for the programming modules are coming in Part 2.

COMMUNICATION PROGRAM

The programmer requires a communication program like ProComm. The HyperTerminal program that comes with the communications package of Windows 95 works just fine.

Most communication programs start up in a mode where received data is displayed on the screen. If yours doesn't, put it in that mode.

The communication program should be set up as follows:

- local echo on
- eight data bits, no parity
- no handshake
- same data rate as programmer (9600 or 19,200 bps)

If you're using HyperTerminal, set it for a direct connect to the serial port the programmer is connected to (as opposed to modem communication).

USING THE PROGRAMMER

Let's say you want to program a 2764 EPROM. Start the communication program and turn on the programmer. The main menu should be displayed.

First, select the device type. Since you're doing a 2764, select type 01:

```
#T 1
```

If you forget the device codes, #L lists them. (Of course, I don't have them all memorized myself. Why do you think I included this command?).

Next, fill the memory from 0000 to FFFF with FF so unused locations will program faster:

```
#F 0 FFFF FF
```

Then download the file. Let's say it's an Intel-format hex file:

```
#DH
```

The LED on the programmer goes from green to red, which indicates

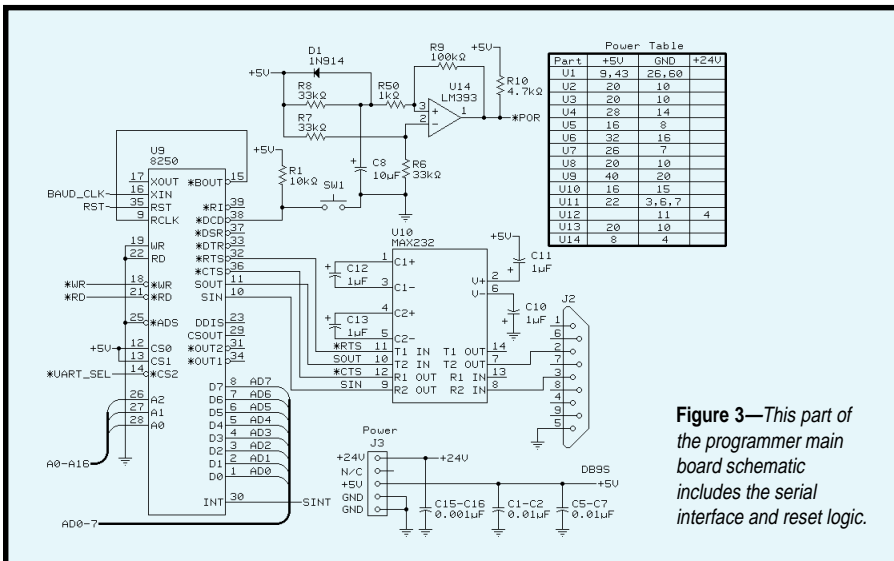


Figure 3—This part of the programmer main board schematic includes the serial interface and reset logic.

that it's waiting for data. Use the communication program to send the hex file to the programmer.

Tell the communication program that you're sending an ASCII file. Don't try to use a modem protocol like XModem. If you are using HyperTerminal, select Send Text File.

As the programmer receives each byte, the LED toggles between red and green. If you are downloading a raw hex file (instead of Intel format), you need to use #Q after the download to tell the programmer that there is no more data.

Install the EPROM to the ZIF socket, and then you can program the device:

#P

The programmer then blank checks, programs, and verifies the part.

While programming, the LED toggles between red and green, giving a (sort of) amber result. If a programming error occurs, a message is displayed.

The programmer doesn't have any editing capabilities, so if you make a mistake when entering a command, just hit Enter and start over.

If there is a certain program sequence that you use regularly (e.g., while developing code for a new project), you can set up a script file that contains the commands and send it from the communication program.

CONSTRUCTION

The prototype was built on perfboard (see Photo 2), but the PC board is recommended. Note the orientation of polarized components (diode, electrolytic capacitors) and ICs.

The 80C188 is installed in a PLCC socket. Be sure to install the socket the right way on the board, and the IC the right way in the socket. Although the 80C188 is keyed with enough force it's possible to deform the socket and install it wrong. If you do that, you'll let all the magic smoke out of the chip when you turn it on.

The bicolor LED and the 50-pin device connector all install on the back (solder) side of the board. The device connector field is laid out for a Hirose 50-pin DIN connector, but you can use any 50-pin connector with pins on 0.1" centers.

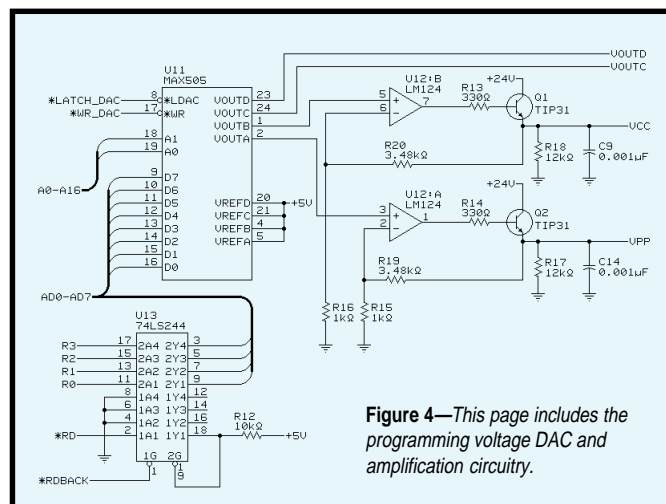


Figure 4—This page includes the programming voltage DAC and amplification circuitry.

The holes for the PLCC socket are smaller than for a standard DIP socket, so some PLCC sockets won't fit. Use one with small pins, or you can install a machined-pin 68-pin PGA socket (which has narrow pins) onto the board and plug your PLCC socket into that. The PLCC socket specified in the parts list (on the Circuit Cellar Web site) has pins that fit onto the board.

If you use the DIN connector, attach it to the board with nuts and bolts before soldering it in. If you solder it first and the connector is not completely seated, you may break the connector when you tighten it down.

The DE-9 connector that connects the programmer to your PC is mounted on a ribbon cable and connected to the board with a 10-pin connector. Mounting the DE-9 connector directly to the board, although simpler, complicates mounting the board. Using the ribbon-cable method enables you to mount the DE-9 connector in a convenient place on your chassis.

POWER SUPPLY

The programmer needs 5 V at about 1 A and 24 V at about 200 mA. The best solution is to buy a power supply with the correct voltages. Marlin P. Jones carries suitable surplus supplies.

As an alternative to an off-the-shelf supply, get a single 24-V supply and use a 1-A DC-DC converter to produce 5 V. Another way is to do the reverse: get a 5-V supply at about 2 A and use a DC-DC converter to produce 24 V.

The last alternative is to buy transformers and build a supply. Whichever method you use, be sure to include appropriate fusing on the AC side of the supply.

ADAPTER MODULES

On both adapters, the 50-pin header mounts on the solder side. All other components go on the top side. I recommend adding standoffs to the corner holes to support the module when it is plugged into the programmer.

The PIC module has a circuit board that holds the 12-pin socket to select 18-

Memory Address	Chip Select	Function
00000–1FFFF	*LCS	128-KB SRAM
F8000–FFFF	*UCS	32-KB EPROM
I/O Address	Chip Select	Function
000–007	*PCS0	16550 UART
080–083	*PCS1	82C55 PPI
100–103	*PCS2	Write DAC data registers
108	*PCS2	Latch DAC data into DAC
110	*PCS2	Write control/LED register
118	*PCS2	Spare I/O strobe to device connector
180	*PCS3	Read 4 status bits of device connector

Table 2—The signals that select specific devices are generated by the internal 80188 chip select decoder. The remaining memory decodes (MCS0–MCS3) and I/O decodes (PCS4–PCS6) are unused.

as GPS and single-chip microcontroller designs. He has written two books on embedded-system design, both available from Butterworth-Heinemann (www.bh.com). You may reach Stuart at sball85964@aol.com.

SOFTWARE

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

SOURCES

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

80C188
Arrow Semiconductor
(800) 777-2776
(516) 391-1676
www.arrowsemi.com

Surplus supplies
Marlin P. Jones and Assoc.
(800) 652-6733
(561) 848-8236

or 28-pin devices. The socket mounts on the solder side of the small board.

After building the PIC module, clip pin 7 on both sides of the 12 × 2 header. Then, insert a wire or other plug into pin 7 of the 12-pin socket to provide a keying mechanism to prevent the header from being installed upside down.

Before applying power, check for solder bridges on the bottom of the board, make sure the ICs are installed the right way, and verify the wiring to the power supply to make sure the correct voltages go to the right places.

WRAP UP

That about does it for the hardware construction. By next issue, you're going to have the hardware together and mounted in a case, right?

Next month, I'll check out the hardware, and take a closer look at the programming modules, the programming algorithms, and the programmer software. 📧

Stuart Ball works at Organon Teknika, a manufacturer of medical instruments. He has been a design engineer for 18 years, working on projects as diverse

FROM THE BENCH

Jeff Bachiochi

Transformerless Power Conversion



Power to the project! The question

is: How to get it there? Jeff's tired of wall warts, so instead he checks into AC and shows you how to eliminate the transformer via a Harris voltage regulator.



Although I admire Thomas Edison for the many inventions his idea factory produced in the New Jersey workshops of Menlo Park and West Orange, he wasn't always right.

Thomas's drive to market DC power distribution just wasn't practical. He wouldn't listen to Nikola Tesla's arguments for AC distribution, and he spent big bucks demonstrating how unsafe alternating current was.

Today, we're willing to live with that risk for the benefit of plentiful and inexpensive power. We are constantly reminded to either stay clear of power lines or run the chance of flatlining.

There are, in fact, appliances we use everyday that perform just fine without converting AC, thank you very much. But, the majority of these appliances like to eat DC for breakfast. Is Thomas having the last laugh?

If it weren't for our dismal advances in the battery field, I'd have to chuckle a bit myself. Storage devices, like "green" energy sources, aren't high on the R&D agenda. So, what's left?

WALL WARTS

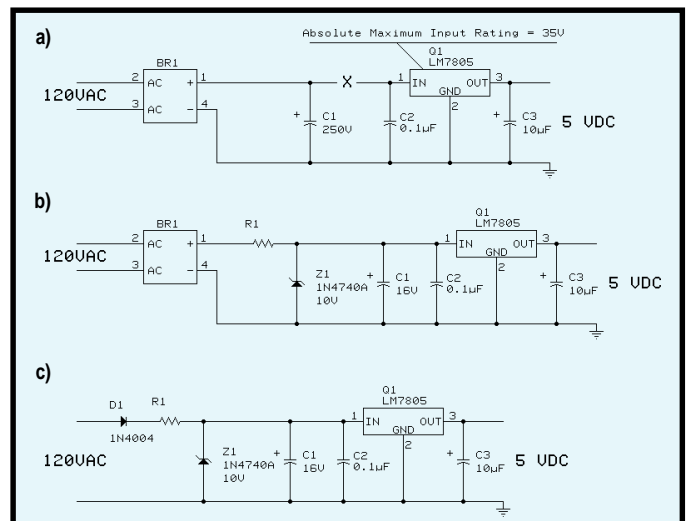
Every time we build a little project, we're forced into tethering it with some AC/DC power supply. You know the wall warts I'm talking about. They come with everything from the cordless phone and answering machine to portable power-tool rechargers and external modems.

That ugly black blob hangs on many of our AC wall outlets and power strips converting the (sometimes) useless AC into low-voltage DC power. I think these things are on the same evolution branch as stockings. They can disappear with the efficiency of a single sock, yet they rise with others to the surface in a pile without any indication as to where their mates are.

The cryptic labeling is often no help in determining where the device came from. It just offers specs on the voltage and polarity output.

Should your appliance become an orphan, many electronics departments offer universal DC power supplies. Some of these come with a slide switch for selecting output voltage and a slew

Figure 1a—Here's what you're not supposed to do—an impractical straight-forward linear design! **b**—A preregulation zener needs a very high wattage series resistor. **c**—Half-wave preregulation reduces wasted current.



of connectors (for matching the socket on your appliance), which you can reverse to change the connectors' polarity.

Whew! The possibility of success for the consumer comes close to your chances of winning the lottery. You'd think somebody would have come up with an alternate approach by now.

TRANSFORMER—NOT

Besides the obvious advantage of isolation, transformers are generally used to reduce the AC down to a level where the ratings of components will not be challenged and where minimal power is wasted.

Eliminating the transformer is more easily said than done. Most linear and switching regulators have maximum input voltages way below line voltages. It's instant death for them to be used without some kind of voltage dropping preregulator.

Let's look at a simple 5-V supply. In Figure 1a, you can see that, following a typical rectifier stage, the capacitors need extremely large ratings, so they would not only be physically large but also expensive.

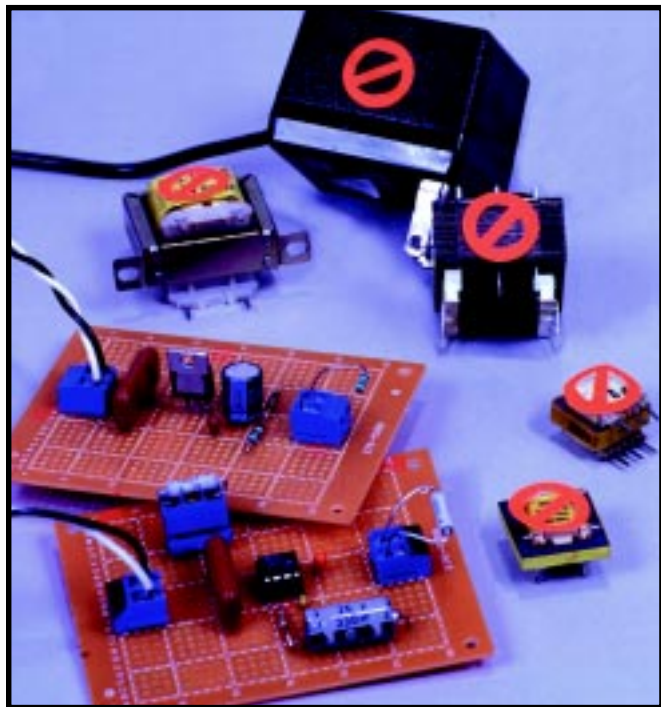
The DC voltage available on the capacitors would also be above what most 5-V regulators could handle as a maximum input voltage. So, other than for special high-voltage supplies, this approach isn't going to hack it.

Figure 1b shows a zener regulation stage that precedes the smoothing capacitor. This approach dumps most of the voltage across the series zener resistor. The capacitor fills in the narrow gaps between the full-wave rectified signal across the zener.

In a situation where the circuit current is fairly constant, the zener remains happy. However, if the circuit's current set by the zener's series resistor isn't needed by the circuit load, the zener has to handle the excess current.

A standard 1-W zener is going to self-destruct rather rapidly as it takes on what the load isn't using. Additional current can also be generated by an increase in the input voltage.

Photo 1—Without bulky transformers, many small circuits can be manufactured with a smaller price tag.



In addition to the balancing act to prevent zener failure, the voltage dropped across the zener's series resistor can add up to very large power dissipation—power dissipation of about a watt for every 5 mA of circuit current. Those are some heavy losses.

You can quickly cut that waste in half by using half-wave rectification, as shown in Figure 1c. Only half cycles of the line voltage get through, so there's only half the waste. Unfortunately, since we're now dealing with a much choppier input, more capacitance is required to fill in the valleys. The costs go up to keep the ripple at an acceptable level.

By adding a preregulation stage, the size (and rating) of the input capacitor is drastically decreased and the input voltage is sufficiently reduced such that a linear or switching regulator can now be used. Heat dissipation can now be shared between the two regulating stages.

You can adjust the preregulation voltage level (zener voltage) to place more of the burden on the first or second stage. This setup also lets you choose the most cost-effective parts based on working voltages.

Getting rid of heat can be a major problem. Small- and medium-wattage resistors generally depend on free air radiation and have no heatsinking (other than the component leads). TO-220 (tabbed) linear regulators can be affixed to a heatsink. The best idea is to raise the efficiency of the supply so there is less heat to worry about.

PREPACKAGED HIGH-VOLTAGE LINEAR REGULATOR

Harris Semiconductor packages much of these features in a TO-220 adjustable output regulator. Similar to the LM317, the HIP5600 has additional circuitry, so it can connect directly to AC line voltages (50–280 VAC). The output voltage is set by a resistor divider, connected between the voltage output, the adjust input, and ground.

In the block diagram in Figure 2, the series rectifier on the input blocks the negative half cycle of the AC line voltage. A high-voltage pass transistor is

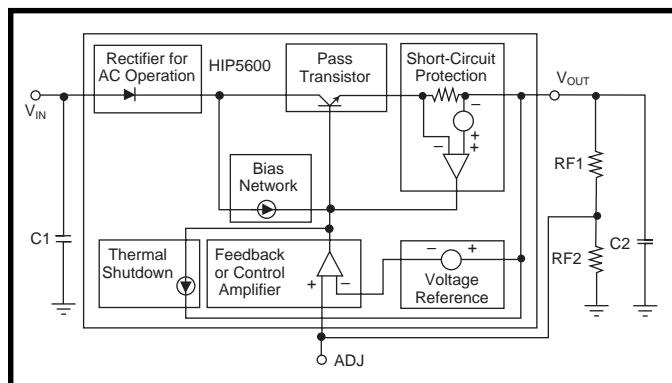
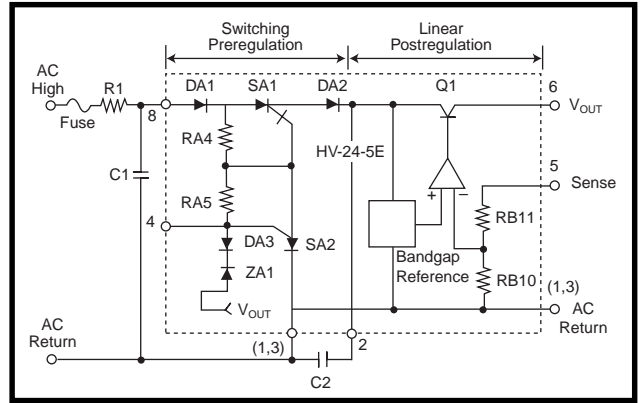


Figure 2—The HIP5600 is a TO-220 regulator with very high voltage input specifications.

Figure 3—The Harris HV-2405E switcher eliminates much of the wasted current by preregulating it to a level in which the post-regulator stays happy.



controlled by a comparator that attempts to maintain an output voltage 1.18 V higher than the adjust-input

voltage. The device has both short-circuit and thermal-shutdown protection.

I prototyped the HIP5600 for use at 120 VAC and found the documentation a bit lacking. Most of the discussion was about regulating high DC voltages. Yes, the rectified AC input was DC or more clearly pulsating DC (half-wave) but all the examples showed a 10- μ F capacitor across V_{out} . That's 100 times too small for a load of 10 mA.

The TO-220 dissipates 0.7 W in free air with no heatsink. Because P equals $(V_{in} - V_{out}) \times I_{out}$, 0.7 W equals:

$$\frac{0.7}{(120 \text{ VAC} \times 0.707) - 5} \sim 8 \text{ mA}$$

By attaching the device to the PCB (as the heatsink), you can get twice the output current. With a maximum output current of 40 mA, the HIP5600 isn't considered a high-current device, even if it's attached to an adequate heatsink.

SINGLE-CHIP POWER SUPPLY

Let's look at another approach, again from Harris Semiconductor. The HV-2405E is an eight-pin DIP, which offers a pre- and postregulator (see Figure 3).

The preregulator is a switcher, which tries to keep the charge on capacitor $C2$ about 5 V above V_{out} . This gives the linear regulator sufficient overhead to operate properly.

Unlike the HIP5600, the HV-2405E needs a series current-limiting resistor. It limits the in-rush current to 2 A on startup and dissipates power. The switching preregulator improves the efficiency, which reduces the power dissipation when compared to the HIP5600.

This time, documentation was clear about the values needed for clean operation with AC line input (15–275 VAC),

although V_{out} is adjustable from 5 to 24 V. By tying V_{out} and the Sense input together, the device regulates to 5 V. Photo 1 shows you the two prototypes.

I found the efficiency of the HV-2405E to be about twice that of the HIP5600. With a 30-mA load, the power dissipation on the series resistor $R1$ was about 1.5 W. Maximum output current for the HV-2405E is 50 mA.

An advantage of the HV-2405E is the minimum input voltage. At a 10-mA load, the device needs only 15-VAC input. This minimum input doubles to 30 VAC at the maximum load.

DANGER, WILL ROBINSON

Getting rid of the bulky transformer means circuits are not isolated from the power line. Exercise extreme caution if devices aren't isolated. Don't use these circuits if the user may be directly or indirectly exposed to any part of the wiring. Safety is always the top priority.

There are plenty of designs where these devices can be effective. A transformerless supply lets you design a smaller, lighter, and ultimately cheaper circuit. But for heaven's sake, don't put your life on the line. ☠

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

SOURCE

HIP5600, HV-2405E
Harris Semiconductor
(800) 4-HARRIS
(407) 729-4984
www.semi.harris.com

SILICON UPDATE

Tom Cantrell

Power Trip



You thought B&B stood for “Bed &

Breakfast.” Not anymore. Tom has redefined the term to mean “Brains & Brawn.” Run this through the chip translator and you get “smart power chips.”



These days, there's an embedded controller available to suit just about any fancy.

Everything from the perennially popular 8-bit chips, now sporting sub-\$1 price tags, to multimillion transistor 32-bit systems on a chip.

Same goes for memory which, with prices ever plummeting, can handle the most gaseous vaporware and then some. All in all, there's enough intelligence to deal with some rather grand challenges.

With so much smart silicon to choose from, there's a shift of emphasis from brains to brawn. In other words, the success of a micro-based product increasingly depends on the pragmatism and utility of the system design, rather than what flavor of chip it uses.

Consider the PDA saga, the Silicon Valley equivalent of a Keystone Kops comedy. Without getting into a religious flame-fest, it's clear that even the first-rate ARM CPU was unable to salvage the Newton. Lesson: It's all too easy to make a weak product with a strong chip.

Subsequently, the PDA market has seen more practical units like the 68k-based Palm Pilot take the lead. Yeah, it may not have the whizziest technology, but it fits in a pocket and the batteries last more than 30 minutes.

In fact, with a dozen chips that can do the job at hand, these days a designer is likely to spend as much time headscratching over power generation and management issues as pondering architectural arcana.

I THINK, THEREFORE I AMP?

Fortunately, the leading linear IC providers are keeping pace with clever solutions that go way beyond yesterday's low-tech discretely.

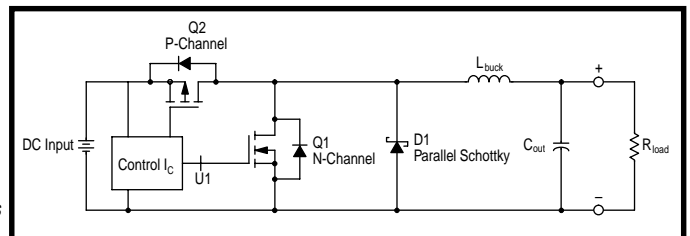
On the desktop, the latest wonderchips' power demands have left behind the days of getting by with a cheap +5-V regulator. Between the CPU and the rest of the system, nearly half a dozen different voltages are required.

The matter is complicated by “green” regimes that call for explicit control of each supply practically down to the individual chip level. At the same time, when the pedal hits the metal, anticipate power surges more akin to starting a car than a chip.

In particular, the newest Pentiums have such finicky demand, they're spawning an entire power niche unto themselves served by parts like Linear Technology's LTC1553, the Maxim MAX1638, and Motorola's MC33470. These chips all work similar to the brains of DC-DC converters that take the plain +5-/±12-V output of silver-box power supplies and step it down to the 1.8–3.5 V required by the latest CPUs.

In fact, Intel has gone to a scheme in which the CPU chip outputs a digital five-bit code specifying the optimal voltage in 50-mV increments. Keep in mind that even though the voltage is reduced, overall power requirements continue to escalate ampere ratings into double digits.

Figure 1—The DC-DC converters that power the latest CPUs include dual MOSFET power transistors, a diode, and an inductor controlled by a synchronous rectification IC.



To deliver the juice, a complete synchronous rectification buck converter combines the controller with a pair of power MOSFETs, an inductor, and a Schottky diode (see Figure 1).

Motorola is offering a kit that includes their MC33470 controller (see Figure 2), the other parts (power transistors, Schottky diode, inductor), and a PCB. The price is certainly right. At the time of this writing, the kit is free and can be ordered via their Web site.

You can refer to previous articles for complete explanations (e.g., "Power Systems for

Autonomous Robots" by Ingo Cyliax in *INK* 92), but the basic buck-converter concept is quite simple, at least on the surface. The controller need merely switch the load between the main (or high side) MOSFET (Q2) and the rectifier (or low side) MOSFET (Q1). The proper output is established by varying the switching duty cycle (i.e., PWM).

When the main (Q2) switch is on, power is delivered to the load and also builds a magnetic field in the inductor. With Q2 off and Q1 on, the inductor magnetic field breaks down, generating current and maintaining voltage across the load. A good analogy is the way a coil (inductor), distributor (transistor), and spark plug (load) work in an automotive ignition.

Actually, the simplest buck-converter designs don't need Q1. However, adding it significantly improves the efficiency (i.e., reduced power loss from input to output) by sharing conduction duties with the diode.

The diode handles the initial high current but suffers excessive losses from forward voltage drop. The transistor takes over as current falls off, with less power loss because of low resistance (mere milliohms) through the switch.

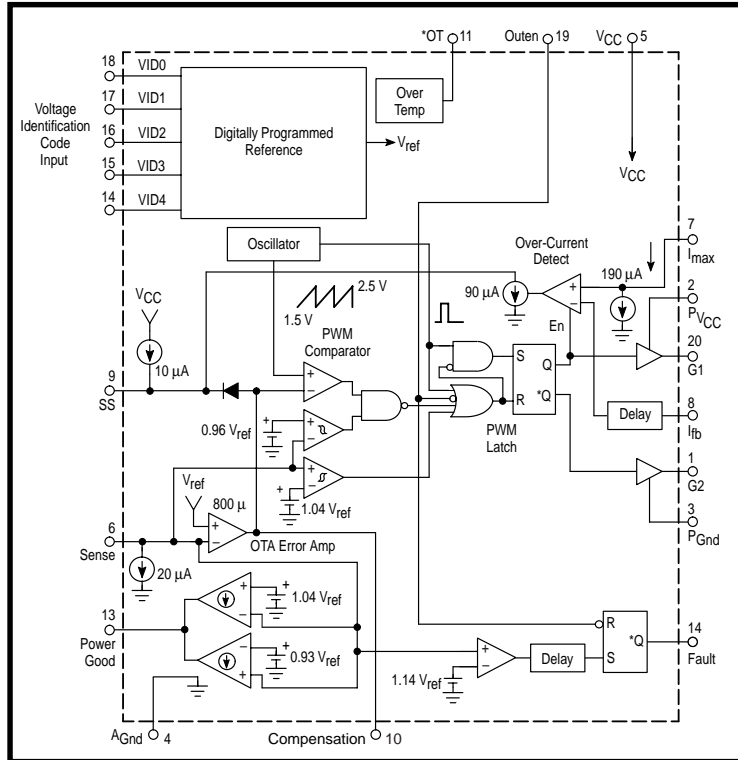


Figure 2—A DC-DC controller IC like the Motorola MC33470 handles the basics of voltage conversion and also deals with practical issues like thermal and overcurrent limiting, on/off control, and soft start.

HARSH ENVIRONMENT

Yes, it's possible to whip up a quick and dirty controller for a low-ball power supply with little more than a voltage ramp and comparator, but that won't cut it for fancy chips or finesse the fine points. For example, the PWM needs some dead time (about 100 ns) when switching between the transistors lest they fight each other.

Maintaining output regulation in response to radically (as in 0–10+ A) changing load isn't trivial. This isn't just a power-up or power-management issue but an ongoing concern. Power demand fluctuates significantly in normal operation, particularly as activity moves on (cache hit) and off (cache miss) chip.

Like traditional designs, the controller senses the voltage level to servo the output (i.e., vary PWM duty cycle). However, a control loop that achieves good stability can't respond quickly enough to the most extreme surges. Thus, the controller includes a special override circuit that steps in to expedite recovery when the output falls more than ±4% out of range.

Powerup is a special case with gotchas for the unwary. First, the MOSFET

gate drives must be held in check until input power ramps up and stabilizes.

Soft start is a power-friendly characteristic that minimizes the extreme power-up surge by slowly ramping the PWM. It's something akin to getting woken by soothing music versus being doused with a bucket of ice water.

Current limiting is important when you're dealing with chips that think they're a kitchen appliance. After all, your chip will certainly be toast if something goes wrong and the silver box dumps a 100+ W onto your pricey sliver of silicon.

Therefore, you also need to monitor the current draw. In the case of the '33470, you do this by

measuring voltage drop across the main switching transistor and comparing against a limit set by external resistor. If the current limit is exceeded, the chip shuts down and then tries to soft restart.

Besides automatic over-current shutdown, the '33470 includes an output enable pin (OUTEN) that enables external logic to flick the on/off chip. It could be under the control of a dedicated power-monitoring and -management chip or even (at least for turnoff) the CPU itself.

With or without such supervision, another safeguard against catastrophic failure (e.g., short circuit) is to connect a negative tempco thermistor monitoring the main switch transistor to OUTEN. When the OUTEN pin drops below 2 V, the controller drives an OT (over temperature) output as a warning. If below 1.7 V for more than 50 µs, the device shuts down.

DC IN A DIP

The same buck-converter design that handles hot desktop chips is downsized by National to serve middle-of-the-road MCU apps. The LM2825's big claim to fame is packaging. It

comes in a 24-pin DIP (see Photo 1) rather than the unwieldy modules, boards, or boxes typically used.

Power capability is 1 A, which isn't a lot by desktop standards but more than enough for simpler single-board designs. Fixed 3.3-, 5-, and 12-V versions are offered as are two adjustable (via external resistor) versions—one covering 2–8 V and another covering 7–15 V.

Despite lesser power pretensions, the LM2825 shares some of its desktop counterpart's bells and whistles, including external shutdown pin, automatic thermal shutdown, automatic (but nonadjustable) current limiting, and soft start.

Electrical specs are easy to work with. First, the input voltage can vary anywhere from about +2 V over the output (e.g., 7 V for a 5-V output) all the way up to 40 V. Nice to have a single unit that can work handheld (9 V), in a car (10–18 V), and in a telecom environment (24 V), or with practically any wall mount.

Output accuracy is adequate at $\pm 5\%$ guaranteed across a wide -40°C to $+85^{\circ}\text{C}$ ambient operating temperature range since most chips only require $\pm 10\%$ tolerance these days. Load regulation is excellent with less than 10-mV output glitch in response to a 0.1–1-A load step.

The unit's conversion efficiency is up there at 75–85% and beyond, depending on the input/output differential. To top it all off, National says the

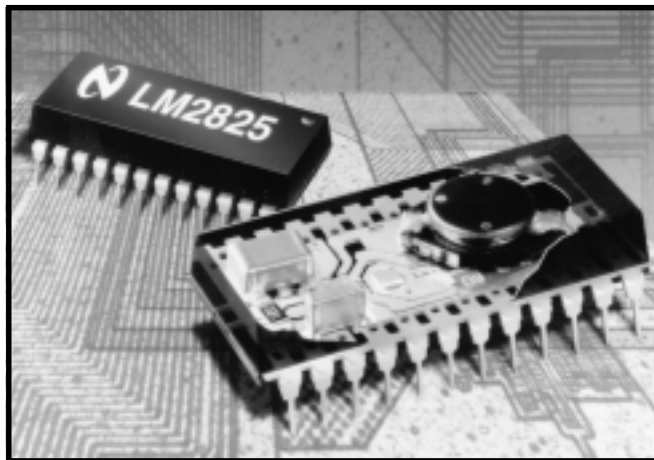


Photo 1—The National LM2825 moves DC-DC converters from bulky, nonstandard modules into a tiny and easy-to-use 24-pin DIP package.

units' MTBF is over 20 million hours, and they even meet rigorous Class-B EMI specs.

When you're talking 35-W/in³ power density, thermal issues are a concern and in fact impose some operating limits. To avoid adding heatsinks or fans, using a lot of copper on the PCB is an effective way to dissipate heat via the pins (see Figure 3).

Even so, with the junction temperature maximum specified at 125°C and a thermal resistance of 30°C per watt, the 1-A output spec must be derated (by as much as 50% in the worst case) for temperature and input/output differential extremes.

Along with the tiny package, the LM2825 offers a similarly shrunk price (around \$10 in hundreds). It's quite competitive with the run-of-the-mill modules, which are not only bulkier but usually lack the built-in features (shutdown, soft start, etc.).

IRREGULATORS

The highfalutin switchers may be grabbing all the headlines, but the good old three-terminal linear regulator is still the best choice for the simplest and lowest-cost apps.

The basic design hasn't changed in decades. Slowly but surely, the specs and usability continue to improve.

Most obvious is dropout voltage (the minimum required input/output differential), which, formerly a few volts, is now down to fractions of a volt. Cutting the voltage headroom means more efficiency and longer battery life (remember, excess input

voltage is simply wasted generating heat).

Another big plus is the shift from funky three-pin through-hole packages to remarkably tiny surface mounts, which are typically eight-pin. The so-called SOT23 handles 100-mA class loads in a package only about 1/8" on a side and the slightly larger SOIC up to 0.5 A. The extra pins are put to good use with the addition of shutdown inputs and low-battery detect outputs.

Extending battery life is job one, so quiescent and shutdown power consumption are both targets. The former refers to the power consumed by the regulator itself, independent of the load. Obviously, the less power consumed by the regulator, the more available for useful work.

Yes, during normal operation, the overhead is only a small fraction of the total (e.g., tens of microamps regulator supply current for a hundreds of milliamps regulator). However, these days, most MCUs offer low-power sleep and standby modes, which cuts their own power consumption to tens of microamps. At that point, the regulator's own demands loom large, perhaps consuming more power than the chip it's connected to.

In fact, unless there's a reason not to, it's best to take advantage of the shutdown feature. Shutdown cuts power by another order of magnitude.

Tiny regulators with such on/off control lend themselves to distributed power schemes for multichip designs. Few applications actually require all the chips to be doing something all of the time. Why not partition the design based on power activity and give each subset an insert switchable regulator?

Good idea. In fact, it's so good the chip suppliers are already way ahead of me. Maxim offers the MAX8865 with dual 100-mA LDO (low dropout, only 110 mV) regulators, each with independent on/off control, in 2.8-V, 2-V, and adjustable versions.

Motorola takes the concept further with the five-output MC33765. Oper-

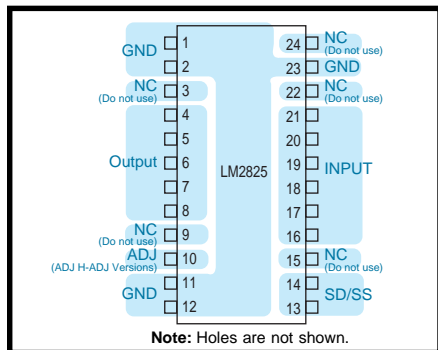


Figure 3—Packing an entire multiwatt DC-DC converter in a 24-pin DIP does raise thermal concerns. The first line of defense is to use the pins as a thermal conduit to copper on the PCB.

ating from a single 3–5.3-V input, each channel delivers a tightly regulated (2.7 V min., 2.85 V max.), low-noise ($40 \mu\text{V}_{\text{RMS}}$) output with independent enable, supplemented by a global on/off control.

Each output has its own current rating (30, 40, 50 60, and 150 mA) and features thermal and overcurrent protection. As usual, the total combined output is limited by thermal concerns (e.g., 250 mA at 25°C, 130 mA at 85°C).

DUALING BATTERIES

Combine ever-lower power chips with improving battery technology (e.g., NiMh, Li), and many designs are candidates to go totally wireless. Just run the gizmo on batteries and throw 'em away (or recharge externally) when they're tapped out. No need for charger electronics, a connector, or a power cord.

The only problem with such a scenario comes when it's time to change batteries. Unless the design is nonvolatile, it's going to take some deft shuffling to get the old batteries out and new ones in before the bits evaporate.

Linear Tech has come up with a novel solution in the LT1579, which is shown in Figure 4. It's a lot like the previously described linears—a 300-mA LDO (400-mV dropout) with adjustable output (1.5–20 V) and shutdown in small 8-pin (SO) and 16-pin (SSOP) packages.

The big difference is that the '1579 accommodates dual battery inputs,

automatically switching between them as appropriate. The 16-pin unit goes further with an explicit battery select input (SS), an output flag (BACKUP) that indicates which battery is currently in use, and pairs of low-battery in (LBI) and low-battery out (LBO) pins.

The LBI and LBO scheme uses an on-chip 1.5-V reference feeding one side of a comparator. An external resistor divider connected to the LBI pin sets the trip voltage, with hysteresis built in to prevent chattering. Once tripped, the corresponding LBO is driven to give the system early warning.

The low-battery-detect feature is especially useful in conjunction with the previously mentioned battery select (SS) to protect batteries from deep discharge. Set the threshold to a safe limit and switch using SS before sucking the battery totally dry, and you won't have to worry about shortening battery life.

POWER TO THE PEOPLE

It's rather clear that power generation and management have come a long way. Nevertheless, I expect we'll see even more progress.

Maxim provides a hint of things to come with their preannounced complete power-management solution. Even without the detailed specs in hand, it's easy to see from Figure 5 that this 28-pin gadget pushes the limits with digitally programmed output step-up (MAX847) or step-up/down (MAX769) DC-DC converters combined with

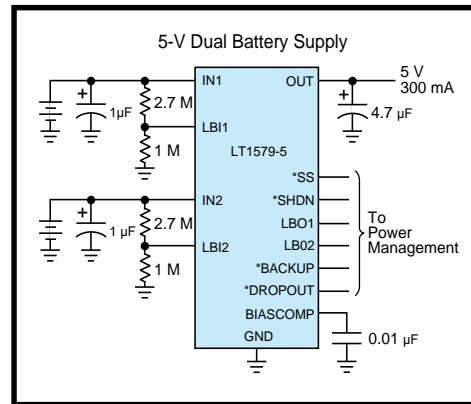


Figure 4—For battery-powered-only apps, the dual-input LT1579 provides a novel solution to the problem of changing batteries on-the-fly.

multiple linear regulators, charger control, automatic battery switching, shutdown (RUN/COAST), power-on reset, and even an optional ADC.

Just remember, even the fanciest design isn't worth a hoot without an efficient, reliable, low-cost source of power. Yeah, a luxu-chip may make for a nice ride, but not if it's always stuck on "E." ☹

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for more than ten years. You may reach him by E-mail at tom.cantrell@circuitcellar.com, by telephone at (510) 657-0264, or by fax at (510) 657-5441.

SOURCES

LT1579, LTC1553

Linear Technology
(408) 432-1900
Fax: (408) 434-0507
www.linear-tech.com

MAX847, MAX769, MAX1638

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

MC33765, MC33470

Motorola SPS
(800) 441-2447
www.motorola.com/wireless-semi
www.mot-sps.com/ppd/html/syncrec.html

LM2825

National Semiconductor
(800) 737-7018
(408) 721-5000
www.national.com

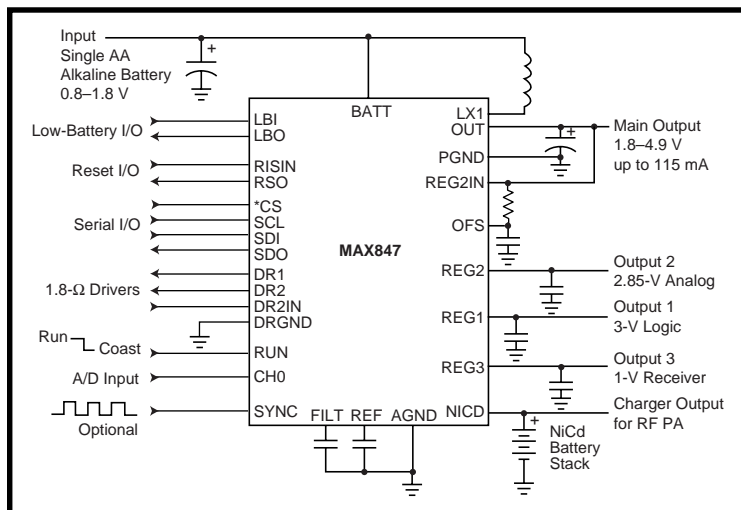


Figure 5—If the forthcoming MAX847 is any indication, ever-fancier and more highly integrated power solutions are just around the corner.

PRIORITY INTERRUPT

When Boilerplates Won't Do



W

hen a niche magazine like *Circuit Cellar INK* promotes a message, it's called an eccentricity. When a major trade magazine promotes a message, it's called an industry trend. It is often only in hindsight that a cause championed by the minority gains due respect. Let me explain.

Like *Circuit Cellar INK*, many technical magazines have Web sites that offer supplemental materials. It's no secret. The object is to create traffic and sell advertising. The difference is, trade magazines are now discovering something we've known all along.

All magazines base their advertising charges on circulation and audience demographics. In the case of the electronic trade magazines, large companies pony up \$12,000 per advertising page to supposedly reach 100,000 engineering managers, department heads, and CEOs (everyone a purchasing-decision maker, mind you). Given that most of us who fall in one of these categories have piles of unread trade magazines on the corner of our desk, I suspect that real readership is considerably less. Of course, since circulation is based on the number sent and not the number actually read, the mailbox continues to overflow.

The digital realities of Web advertising throw a major wrench in the works, however. The statistical packages from most ISPs track the physical number of visitors to a site, a page, and even to specific advertisements. The potential circulation might be infinite (the whole Internet), but the real circulation is only the record of who actually comes to the site or a page.

With banner ads on some of these trade magazine sites going for \$6000 per month, they have a vested interest in getting you to their site. But, that's the rub! Searching the Internet for relevant technical information is not the typical activity of engineering managers, department heads, and CEOs. Most of the Web surfers I meet aren't delegating the tasks; they're performing the tasks. They're most attracted to sites that offer application information that simplifies that task.

It takes guts to buck tradition. A specific example is the online news and technical resource publication, *EDTN* (www.edtn.com). *EDTN* is a joint venture between Aspect Development Inc. and CMP Media Inc. You're probably more familiar with CMP. In addition to their recent purchase of *BYTE*, they publish *EE Times*, *Electronic Buyers News*, *Semiconductor Business*, and about 40 other trade magazines. In the process of building their Web site, *EDTN* came to the inevitable conclusion that the typical trade-magazine boilerplate won't satisfy this crowd. They have to avoid the traditional trade-magazine-formula approach to editorial in the all-important design sections. That's when our name came up.

Beginning this month, *Circuit Cellar INK* will be providing the editorial content for the embedded-systems section of *EDTN*'s *EE Design Online*. I applaud *EDTN*'s boldness in the face of obvious politics. Ultimately, the marriage of their wide audience and our real-world content should benefit both of us. It also demonstrates what others, like Hamilton-Hallmark which posts my editorial each month, have known all along: *Circuit Cellar INK* offers high-quality technical information.

Electronic media changes don't stop there, however. It has always been frustrating to me that we have considerably more editorial than we have pages available in the magazine. This month we inaugurate a new section on our Web site called Design Forum. It's a subscriber bonus section that contains new monthly articles, feature columns, and design projects.

Each month will contain an additional Silicon Update from Tom Cantrell. We've always felt that one column a month hardly covered his West Coast investigations, so now there are two. Lessons From The Trenches is a new column by George Martin. George and I have worked together on many projects over the years. He's the guy I call when I need help. Lessons From the Trenches documents some of the design lessons he's learned the hard way. Design Forum will also contain design hints and new feature articles. Because we aren't limited for space, these projects will typically contain more example listings and illustrations.

Finally, Design Forum solves a real problem for me. When we conduct a contest with the overwhelming success of Design98, it results in a lot of publishable projects. We print the winning projects in the magazine, but there are dozens of others of equal value. As only one of the judges, my top picks weren't necessarily always the winners. As the publisher, however, I get a way to show my top choices to you. PIC Abstractions is a selection of PIC projects from our Design98 contest.

One of the first projects is a personal favorite of mine. It is for a 128 x 240 LCD graphing weather monitor that displays a 48-h moving graph of pressure and temperature. Even if you don't build one yourself, stop by and take a look at the pictures. You'll be amazed at the sophistication and performance of this little device.

So, is anything really different? Our message has remained the same through the years. The people who build all the electronic gadgets we take for granted want a source of reliable design information, and we have provided it. Some may call our mission an eccentricity, I'd rather think of it as an industry trend.

steve.ciarcia@circuitcellar.com