CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#72 JULY 1996

DEBUGGING TECHNIQUES

Alternative Debugging Techniques

Motorola's New Single-Wire Development Interface

Flash Memory Emulation

Hard Facts about Soft Ferrites

$3.95 U.S.
$4.95 Canada

TASK MANAGER

An Exterminator's Dream

This issue was a particularly fun and interesting one for me to edit. Given the amount of embedded programming and debugging I do, this month's feature articles were especially relevant to my daily work.

As I read through the articles, I found myself feeling justified that many of the tried-and-true techniques I've relied on for years were suggested by multiple authors as valuable tools. I also found some new ideas to try on my next bug-hunting foray.

Even if you're a veteran entomologist, take a look at what we offer in the features. You're bound to learn something new.

We start off the feature section with an article on alternative debugging techniques. Since many of today's processors make it increasingly difficult to access their insides, Jerry Merrill offers some inexpensive, processor-independent, hardware-assisted debugging techniques.

Next, Graham Moss and his partners start a three-part article on in-circuit emulators. In this first part, they take a look at some low-cost options, including EPROM emulators. Next month, they continue with a discussion of monitor-based debuggers and true in-circuit emulators.

When you're using an embedded microcontroller with internal memory and I/O, your hardware-assisted debugging options become severely limited. An application that uses most of the I/O lines makes matters even worse. Motorola has included a built-in single-wire debugging interface on their latest microcontroller (the MC68HC12) that promises to revolutionize low-cost embedded development. Motorola engineer Jim Sibigtroth gives us some of the background into how the single-wire interface was designed.

These days, when an application needs external memory, flash memory is used more and more. Flash memory emulation then becomes yet another debugging tool, as Arvind Rana explains in the next article.

In our final feature, Bob Meister digs out an old favorite-the LSI-1 l-and describes how to write a simulator to run on the IBM PC. While the LSI-11 may be old hat to many, the techniques used to write such a simulator aren't, and can be applied to most any processor.

In our columns, Ed continues his examination of 'x86 processor performance, Jeff explores soft ferrites, and Tom looks at the latest 8-bit offerings from Zilog.

editor@circellar.com

CIRCUIT CELLAR®

THE COMPUTER APPLICATIONS JOURNAL

FOUNDER/EDITORIAL DIRECTOR
Steve Ciarcia

EDITOR-IN-CHIEF
Ken Davidson

MANAGING EDITOR
Janice Marinelli

TECHNICAL EDITOR
Elizabeth Laurençot

ENGINEERING STAFF
Jeff Bachiochi & Ed Nisley

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Rick Lehrbaum
Fred Eady

NEW PRODUCTS EDITOR
Harv Weiner

ART DIRECTOR
Lisa Ferry

PRODUCTION STAFF
John Gorsky
James Soussounis

CONTRIBUTORS:
Jon Elson
Tim McDonough
Frank Kuechmann
Pellervo Kaskinen

PUBLISHER
Daniel Rodrigues

PUBLISHER'S ASSISTANT
Sue Hodge

CIRCULATION MANAGER
Rose Mansella

CIRCULATION ASSISTANT
Barbara Maleski

CIRCULATION CONSULTANT
Gregory Spitzfaden

BUSINESS MANAGER
Jeannette Walters

ADVERTISING COORDINATOR
Dan Gorsky

CIRCUIT CELLAR INK'? THE COMPUTER APPLICATIONS JOURNAL (ISSN0896-8985) is published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066(860)875-2751. Second class postage paid at Vernon, CT and additional offices. One-year (12 issues) subscription rate U.S.A. and possessions $21.95, Canada/Mexico $31.95, all other countries $49.95. All subscription orders payable in U.S. funds only, via international postal money order or checkdrawn 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: Please send address changes to Circuit Cellar INK, Circulation Dept., P.O. Box 698, Holmes, PA 19043.9613.

Cover photography by Barbara Swenson
PRINTED IN THE UNITED STATES

Circuit Cellar BBS—24 Hrs. 300/1200/2400/9600/14.4k bps, 8 bits, no parity, 1 stop bit, (860) 871-1988; 2400/9600 bps Courier HST, (860) 871-0549. World Wide Web: http://www.circellar.com/

INSIDE ISSUE 72

READER I/O

A NEW FAN

I am a new subscriber to *Circuit Cellar INK.* I just wanted to tell you how impressed I am with the quality of the magazine. Although I smiled a little at the proposition that business is currently suffering the slings and arrows of outrageous media bias ("Not Just Ferengi Values," *INK* 69), I enjoy Steve's "Priority Interrupt."

And, you have a great BBS.

Jeff Jacobs
adhoc@celestat.com

YET ANOTHER WAY

I enjoyed Mike Smith's article "The Evaluation Board Saga Continues" *(INK 70)* on inexpensive M68*xxx* development tools. I faced a similar need for inexpensive high-quality tools, though for a completely different reason.

However, I solved the problem differently. I built the entire suite of the GNU tools (assembler, linker, librarian, C compiler, C++ compiler, and even a remote debugger) using the free patches from Cygnus support. It takes several hours to generate all the tools, but the process is straightforward.

A UNIX box is required to build the tools, but it is easy to produce MS-DOS binaries as the final result. The GNU sources are available from **ftp://prep.ai.mit.edu/** pub/gnu/ and the Cygnus patches (and docs) are available from **ftp://ftp.cygnus.com/pub/embedded/crossgcc/**. You'll probably want to start by grabbing the FAQ and giving it a quick read.

Jeff Francis
Denver, CO
jfrancis@mail.frii.com

SOLDERHEAVEN

Jeff's "Handcrafting Design Ideas" *[INK 70)* provided a very good review of the different prototyping techniques.

I also like to test a design with a prototype and recognize that a handcrafted board can be faster than laying out a PCB if only one unit is required or if the circuit is simple.

For those times when I wanted to build a permanent prototype, I used wire-wrap wire. But, one must measure, cut, strip, and solder. In seeking alternatives, I even tried enameled wires (also called magnet wire), but without success.

I then discovered a product called VeroWire made by BICC-Vero Electronics. It uses a self-fluxing polyurethane insulation. When you touch the wire with a 300°C–400°C iron, the insulation melts and leaves the wire fluxed. You apply solder directly to it. (It comes in 40-m spools of 34-AWG, 100-mA max colored wire. It's $20 for four spools of differently colored wires and $20 for a wiring pen and one spool.)

This method saves a lot of time. You need only to put the wire on the pad and solder it directly. You can then use the pen to string the wire from pad to pad, soldering it each time.

If you want to use the "channel" technique described by Jeff, Vero even makes comb-type strips that fit in the 0.1" on center holes of the board. You can segregate and route the wires without resorting to hot glue or strips of wires.

To my knowledge, it's the most elegant, durable point-to-point system there is. And, it's fast and cheap.

Thank you for publishing the excellent *Circuit Cellar INK.* I have enjoyed each issue since *1991.*

Michel Clavette
clavette.michel@ic.gc.ca

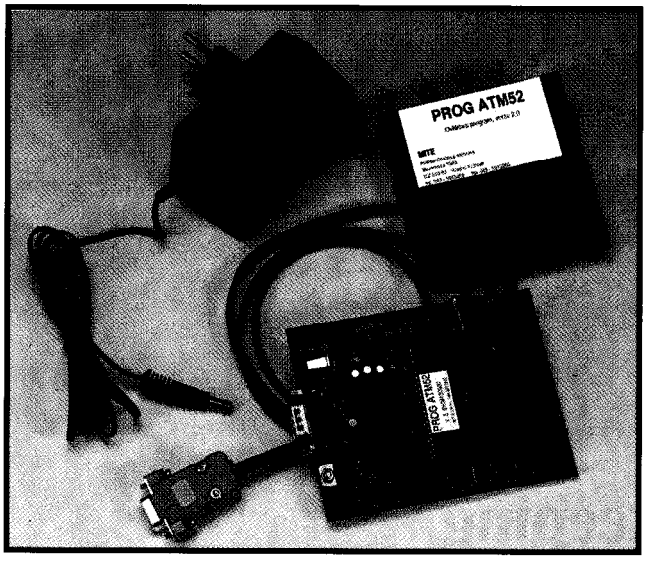
Contacting Circuit Cellar

We at *Circuit Cellar INK* encourage communication between our readers and our staff, so we have made every effort to make contacting us easy. We prefer electronic communications, but feel free to use any of the following:

Mail: Letters to the Editor may be sent to: Editor, Circuit Cellar INK, 4 Park St., Vernon, CT 06066.
Phone: Direct all subscription inquiries to (800) 269-6301. Contact our editorial offices at (860) 875-2199.
Fax: All faxes may be sent to (860) 872-2204.
BBS: All of our editors and regular authors frequent the Circuit Cellar BBS and are available to answer questions. Call (860) 871-1988 with your modem (300–14.4k bps, 8N1).
Internet: Letters to the editor may be sent to **editor@circellar.com**. Send new subscription orders, renewals, and address changes to **subscribe@circellar.com**. Be sure to include your complete mailing address and return E-mail address in all correspondence. Author E-mail addresses (when available) may be found at the end of each article. For more information, send E-mail to **info@circellar.com**.
WWW: Point your browser to **http://www.circellar.com/**.
FTP: Files are available at **ftp://ftp.circellar.com/pub/circellar/**.

NEW PRODUCT NEWS

Edited by Harv Weiner



ATMEL ATM52 PROGRAMMER

MITE offers a simple, inexpensive way to program the AT89 series of Atmel processors. The **PROG ATM52** consists of a 115- x 89-mm printed-circuit board, three-wire null-modem cable, AC adapter (230 VAC to 12 VDC), and diskette with PC software.

The PCB includes a universal DIP40/DIP20 socket for a programmed processor, two LEDs indicating power and busy states, a DE-9 connector for RS-232 communication with a PC, and an adapter plug.

All operations such as reading, erasing, programming, blank checking, verifying, lock-bits setting, checksum counting, or buffer editing are accomplished by the software. The software requires at least MS-DOS 3.3, 80286 or higher processor, 640-KB RAM, hard drive, and a CGA, EGA, VGA, or HCG monitor. It runs under DOS, or it can be executed from Windows in graphics mode

The user can change program configurations such as COM1 or COM2 and the type of processor. The software supports a mouse, and brief help is included. Two types of files are supported: binary and Intel hex format.

The PROG ATM52 programmer sells for $99 in single quantity ($80 per unit in lo-unit quantities). A complete evaluation kit consists of an 805 1 assembler, 805 1 simulator, PROG ATM52, samples of 89C52 and 89C205 1, and datasheets. The **Starter Pack Atmel 52** costs **$115** apiece ($90 per unit in lo-unit quantities).

MITE Hradec **Králové s.r.o.**
Veverkova 1343 • CZ 500 02 Hradec **Králové** • Czech Republic
+42 49 5813252 • Fax: +42 49 5813260
http://www.mite.anet.cz/

#500

PC TROUBLESHOOTING BOOK

An excellent guide to IBM-compatible equipment and general PC system troubleshooting and enhancement is available from WEKA Publishing. ***How to Maximize Your PC,*** edited by Steven S. Ross, is written in simple terms and gives straightforward information about PC hardware and software.

The book is presented exactly as the buyer needs it, enabling anyone to assess their own computing requirements. The work offers hints and recommendations, including many caveats not found in manufacturers' literature.

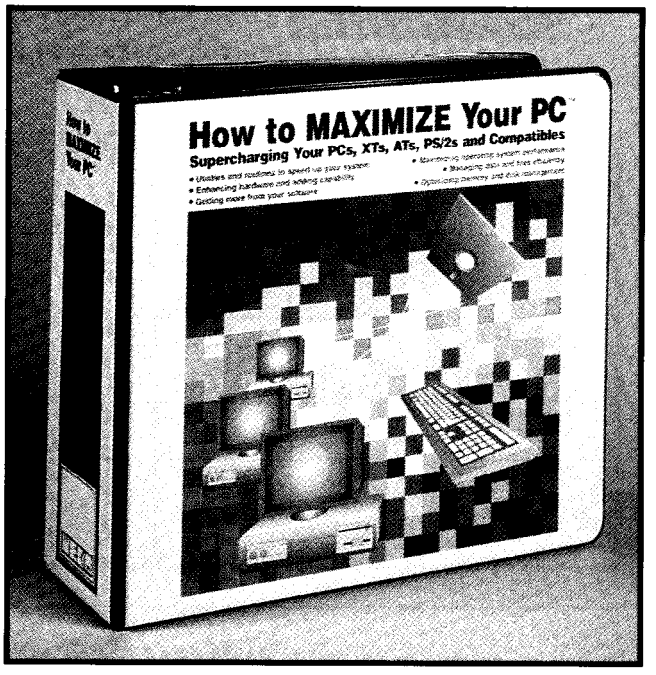
Major topics include IBM standards and compatibility issues, CPUs and coprocessors, operating systems including Windows 95, programming language options, applications software, hardware, peripherals (cables, printers, modems), enhancement guidelines, networks, and software tips.

Each section of the ring-bound volume deals with typical questions, describes common problems, and outlines steps for remedial action. Periodic updates are available direct from the publisher.

Jensen Tools is offering this book at **$59.95**. This price includes a free software disk containing many useful PC utilities. A free copy of the 1996 **Jensen Master Catalog** is also available.

Jensen Tools, Inc.
7815 S. 46th St.
Phoenix, AZ 85044
(602) 968-6231
Fax: (602) 438-I 690

#501



NEW PRODUCT NEWS

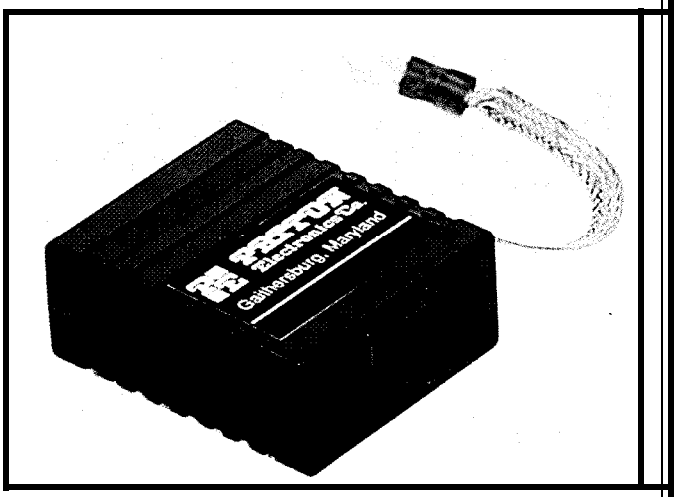
SURGE PROTECTORS FOR 100-MHZ NETWORKS

The Patton Models 570 and 580 Surge Protectors connect directly to Category-5 data lines to protect hosts, workstations, and communications equipment from data loss and damage due to lightning, ESD, EMR, and other surge hazards. The environments protected include Fast Ethernet (100BaseT),100VG-AnyLan, ATM, and RS-422.

The units connect inline to Category-5 cable using modular RJ-45 jacks. Surges are diverted to a chassis ground through a heavy-duty braided metal strap. Both models are EIA/TIA TSB-40A Category-5 certified and also comply with IEC Standards 801.2, 801.4, and 801.5. Near End Crosstalk (NEXT) exceeds -43 dB at 100 MHz, making the units appear to the network as a few inches of Category-5 cable.

The Model 570 installs at the point-of-use, connecting directly to terminal and communications equipment. It has been tested to withstand IEC 801.5 Class-2 surges (1 kV). The Model 580 installs at barriers (e.g., building entrances, between floors, and electrical panes). The 580 withstands IEC 801.5 Class-3 surges (2 kV).

Prices for the Patton Model 570 and 580 are $99 and $159, respectively.



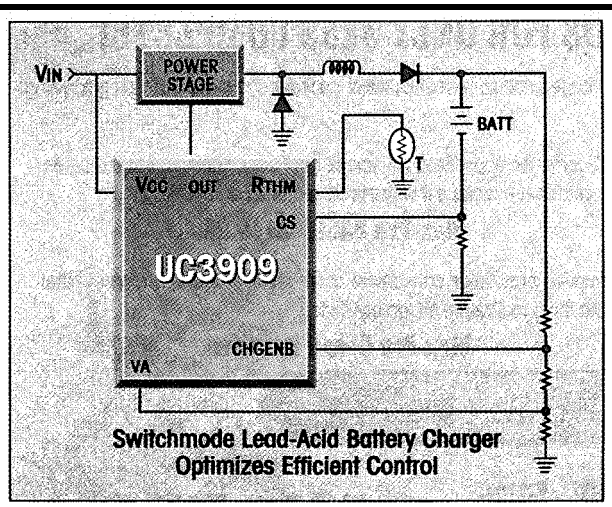
Patton Electronics Co.
7622 Rickenbacker Dr.
Gaithersburg, MD 20879
(301) 975-I 000
Fax: (301) 869-9293
http://www.patton.com/

#502

LEAD-ACID BATTERY CHARGER

Unitrode's new UC3909 family of switch-mode lead-acid battery chargers combines charge-state logic with average-current PWM-control circuitry for greater efficiency in high-current charge applications. The UC3909 provides accurate and efficient battery charging for emergency lighting, security systems, telecommunications, lawn mowers, UPS backup power, and the automotive industry.

Featuring a thermistor interface which tracks battery requirements over temperature, the device provides precise charge control for greater charge acceptance and longer battery life. The UC3909's voltage-control loop with average-current limiting accurately controls charge rate from trickle to overcharge.



Switchmode Lead-Acid Battery Charger Optimizes Efficient Control

Four internal charge states with status bits report accurate charge information at all times. The chip uses undervoltage lockout to monitor V_{CC} and V_{ref}, ensuring sufficient supply voltage before output switching begins.

Additional circuit blocks include a differential-current sense amplifier, a 1.5 % voltage reference, and a –3.9-mV/°C thermistor linearization circuit. Voltage and current error amplifiers, a PWM oscillator, comparator and latch, charge-state decode bits, and a 100-mA open-collector output driver are also included.

The UC3909 is priced at $4.27 in quantity.

Unitrode Corp.
7 Continental Blvd. • Merrimack, NH 03054
(603) 424-2410 • Fax: (603) 424-3460

#503

NEW PRODUCT NEWS

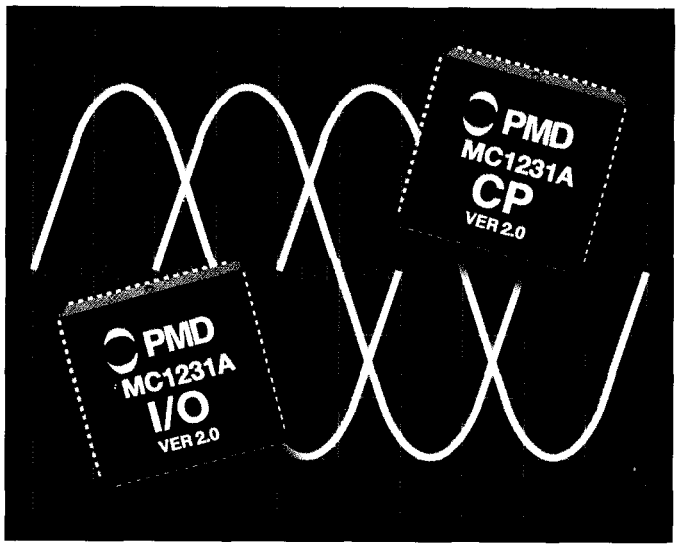
BRUSHLESS MOTOR CHIPSET

The **MC1231A** directly performs sinusoidal commutation of brushless motors using the motor's encoder signal to determine the phasing angle. Up to two independent motor axes can be controlled, and both 2- and 3-phase motors are supported.

The "A" version adds the ability to detect commutation errors on-the-fly for increased reliability. As well, it supports a more sophisticated servo filter with feedforward and a DC-bias offset. The ability to perform torque limiting under software control has also been added.

Standard features of the chipset include four user-selectable profiling modes: S-curve, trapezoidal, velocity contouring, and electronic gearing. Other features include servo-loop closure, two directional limit switches per axis, 16-bit DAC output, lo-bit PWM output, and separate index and home signals.

The MC 123 1A is available in one- and two-axis configurations. The chipset is packaged in two 68-pin PLCCs and sells for $85 in quantity.



Performance Motion Devices, Inc.
97 Lowell Rd. • Concord, MA 01742
(508) **369-3302** • Fax: (508) 369-3819

#504

NEW PRODUCT NEWS

CALLER-ID INTERFACE

Compared to high-priced modems and units costing five to ten times as much, the **ITU Caller-ID-to-PC Interface** is a price and performance breakthrough. The interface measures only 2.25" x 3" and can be plugged directly into any spare PC serial port.

DOS- and Windows-based software with source code is included to transform a PC into an advanced Caller-ID box and to integrate Caller ID into other applications. The Windows-software source code is supplied in Visual Basic, and the DOS-software source code is in C.

The ITU Caller-ID–to–PC Interface kit is available for $39. Each unit comes with a 30-day money-back guarantee and 90-day warranty.

ITU Technologies
3477 **Westport** Ct.
Cincinnati, OH 45248-3026
(513) 574-7523
Fax: (513) 574-4245
sales@itutech.com
http://www.itutech.com/

#505

SWITCHING REGULATOR OPERATES OVER 3.5-36-V INPUT

Linear Technology has introduced the LTC1435 low-noise DC/DC voltage converter. The LTC143.5 is based on a fixed-frequency current-mode architecture that achieves efficiencies up to 95% over an input voltage range of 3.5-36 V. This switching regulator is particularly useful in portable designs that use a variety of multicell battery packs, such as lithium-ion, nickel-metal-hydride, and nickel-cadmium batteries, where the voltage can vary from 4 V for discharged nickel types to over 30 V for wall adapters.

The LTC1435 drives N-channel MOSFETs instead of more expensive P-channel MOSFETs. These N-channel MOSFETs have lower on-resistance, enabling conversion efficiencies to approach 95%, even at load currents in excess of 5 A. Their lower on-resistance results in reduced I^2R losses, greater efficiency, and cooler operation than P-channel MOSFETs. Switching frequencies can be programmed up to 400 kHz, so inductors as low as 10 µH can be used to save board space.

The LTC1435 is available in 16-lead SOIC and SSOP and costs $4 in 1000 quantity.

Linear Technology Corp.
1630 McCarthy Blvd.
Milpitas, CA 95035-7417
(408) 432-I 900
Fax: (408) 434-0507

#506

VERSATILE RTC MODULES

Module products that incorporate a low-power real-time clock (RTC) and nonvolatile SRAM have been announced by Benchmarq. The **bq4830 Real-Time Clock module** directly replaces industry standard 28-pin DIP SRAMs and also fits into many EPROM and EEPROM sockets.

The bq4830 provides nonvolatile SRAM by combining an internal lithium battery with a 32 K x 8 CMOS SRAM, a quartz crystal, clock, and power-fail chip. The chip provides IO-year minimum data retention and unlimited write cycles.

The bq4832 and bq4842 provide full CPU supervision along with the same features as the bq4830. In a 32-pin DIP module, the bq4832 provides a watchdog timer, power-on reset, alarm and periodic interrupt, power-fail and battery-low warning, along with the low-power RTC and a 32 K x 8 nonvolatile SRAM. The bq4842 can replace the bq4832 to provide a 128 K x 8 nonvolatile SRAM.

The three modules also contain a power-fail detect circuit that deselects the device whenever V_{cc} falls below tolerance, thus providing a high degree of data security. All modules are shipped with the battery electrically isolated to provide maximum battery capacity. The battery remains disconnected until the first application of VW

Quantity prices are $18.75 for the bq4830, $20 for the bq4832, and $42.50 for the bq4842.

Benchmarq Microelectronics, Inc.
17919 Water-view Pkwy.
Dallas, TX 75252
(214) 437-9195
Fax: (214) 437-9198
http://www.benchmarq.com/

#507

NEW PRODUCT NEWS

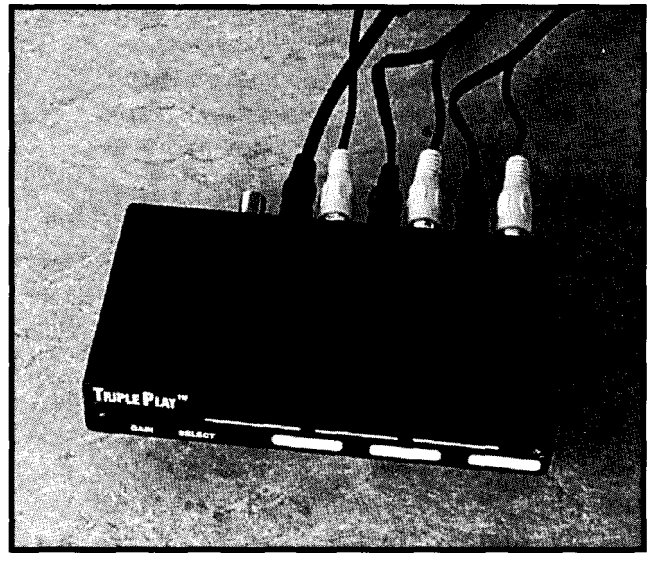
MICROPROCESSOR-CONTROLLED VIDEO MODULATOR

A three-channel, microprocessor-controlled, programmable digital video modulator is available from NetMedia. The **Triple Play** has three modulators which cover video channels UHF 14-69 or cable 70-94 and 100–125 with precise, digital control.

The Triple Play comes with its own 12-VDC, 300-mA power supply. It takes standard video and audio input from home video cameras, security cameras, video recorders, laser discs, and video games to provide closed-circuit television channels for distribution throughout the home or business.

The Triple Play features a built-in video amplifier with external gain control, internally generated test bars to allow individual-channel setup without a video source, and a built-in video sequencer to display multiple video sources one after the other on the same channel.

For example, the Triple Play can be used in a sequencer mode in a security application to view the front door, then side yard, then back door. The unit can be used in sequencer mode alone, with more than one Triple Play, or with a single-channel modulator MMOD70. Either **MS-DOS setup software** or the **Serial Link** setup-no software needed-is required to program the video-sequencer mode.



The Triple Play sells for $149. The MS-DOS software and cable for programming the unit and/or the MMOD70 modulator is $49. The Serial Link setup is also available for $49.

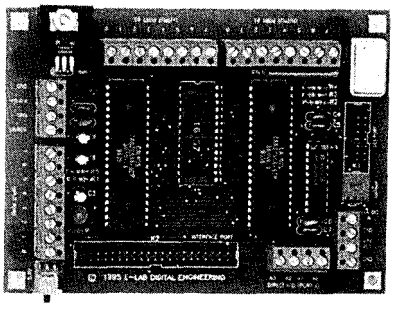
NetMedia, Inc.
10940 N. Stallard Pl.
Tucson, AZ 85737
(520) 544-4587
Fax: (520) 544-0800
sales8 homeautomation.com
http://www.homeautomation.com/

#508

PIC MICROCONTROLLER

The PICPlus microcontroller board uses a RISC-architecture microcontroller (PIC16C57) running at up to 20 MHz. It is ideal for a wide variety of applications such as robotics and motion control, front-end control panels, alarm systems, and embedded-control systems.

The PICPlus board includes an onboard power regulator for battery operation and two high-speed 82C55 parallel interface chips for TTL compatibility. It also has two byte-wide digital output ports, one byte-wide digital input port, one 4-bit bidirectional port, and a 40-pin interface port for connection to expansion circuitry.



A built-in LCD port provides a simple connection to most liquid crystal displays, and an onboard O-5-V analog-to-digital converter directly interfaces to a wide variety of sensors. I/O is greatly simplified due to the low-level routines supplied with the controller. Convenient connection to all ports and power lines is provided by 36 terminal blocks

The board is programmed using either an 805 1 -like assembly language (the PASM assembler is included) or Microchip's original instruction set. With a PIC microcontroller, device programmer, and UV eraser, virtually any embedded control application can be implemented.

The PICPlus microcontroller board starts at $239.

E-LAB Digital Engineering, Inc.
P.O. Box 246
Lawton, IA 51030-0246
(712) 944-5344
Fax: (712) 944-5501
http://www.netins.net/showcase/elab/

#509

FEATURES

Alternative Debugging Techniques

FEATURE ARTICLE

Jerry Merrill

Debugging is not easy. Processors may be too old, new, or fast for ICEs or a project too low profile and short term to warrant the expense. So, try processor-independent hardware-assisted debugging.

We often find ourselves debugging firmware with less effective tools than we'd like. An ICE isn't available because the processor is too new, too old, or too fast. The tools may be available, but the project is too low profile or too short term to warrant expensive, specialized debugging tools.

These problems are caused by the highly CPU-specific nature of traditional hardware-based debugging tools, which become obsolete. Even updated versions of familiar processors often require new debugging hardware.

The trend toward surface-mount processors adds more complexity and expense. High-density, high-pin-count, surface-mount package probes are expensive and finicky.

Designing emulators that interface to a CPU socket is even more difficult. Today's CPUs have shorter cycle times, making the CPU-to-target interface timing much more critical.

Since CPUs incorporate internal caches, prefetch queues, and pipeline architectures, it's increasingly difficult to design hardware that properly emulates the processor at the CPU interface.These factors add up to expensive, nonadaptive development tools.

In this article, I focus on inexpensive, processor-independent, hardware-assisted debugging techniques.

UNIVERSAL HARDWARE INTERFACE

To develop hardware-based development tools tolerant of changes to the CPU speed, package, and bus timing, we have to design to a different hardware interface.

It should allow hardware debugging tools to adapt quickly to new processors or processor derivatives. Ideally, more generalized hardware can be used on a variety of targets, regardless of the make, model, speed, or processor flavor. It should even work for CPU cores buried within an ASIC.

Believe it or not, most embedded systems already have such an interface! The target's EPROM or flash socket is a well-documented, well-understood, and stable interface. Most importantly, its specifications are independent of the target's processor.

Timing at this socket is usually much less critical than at the processor interface. If the processor's cycle times are significantly faster than the memory device's capabilities, the target board incorporates a cache or memory-interleaving techniques to decouple the high-speed CPU interface from the memory-socket interface.

Manufacturers of memory-interface devices can design their products to an EPROM's less stringent and standard mechanical, functional, and timing specifications. These features add up to less expensive, more flexible development tools.

So, here's a processor-independent hardware interface already present on most embedded targets. What can we do with it? Let's discuss some hardware-based tools that interface to the target's memory socket and techniques for using them.

MEMORY-INTERFACE DEVELOPMENT TOOLS

Memory-interface-based development tools are available with a variety of features. Here are some of the major features and debugging techniques each one enables.

The base-line functionality of these tools is simple EPROM emulation. An EPROM emulator is little more than SRAM with a separate download connector. It plugs into a memory socket in place of the EPROM and connects to the development system through a serial or parallel connection.

It enables you to quickly load new code images into its internal SRAM. When the download is complete, it looks like an EPROM to the target.

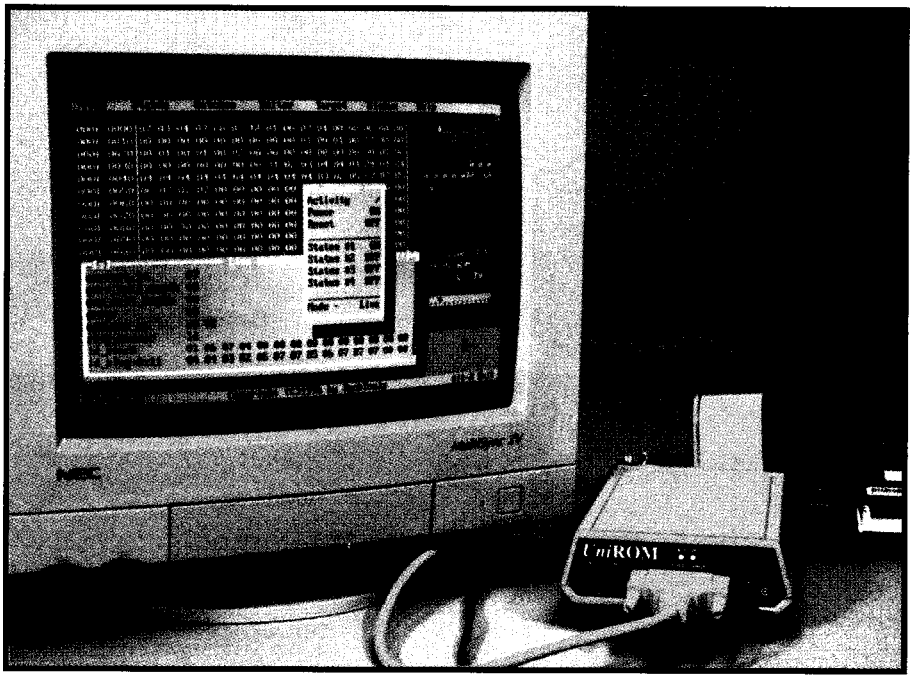


Photo 1-A *live-access* emulator is *shown here tracking down a null-pointer reference. The* data shown in the live *editor is reproduced in Figure 1.*

While this function doesn't actually debug, it speeds up development considerably. The reduced turnaround time permits debugging techniques not feasible before. Even with the best development tools, a basic EPROM emulator easily earns its keep.

The next logical step in memory-interface hardware is an EPROM emulator that accepts target-write cycles. You can then plug into EPROM, flash, and SRAM sockets. More importantly, it provides new debugging techniques since the target can write information into unused memory locations.

Write-back capability enables monitor programs or software-debugger kernels to load applications and write software breakpoints into the target's code space. This feature may be all you need to use these tools on your target.

Some memory emulators have virtual UART options. A virtual UART is a memory-mapped communication port, which adds a serial port to any target with a memory socket. An extra port can be very useful during debug.

These ports can be read/write or read only. The read/write type works like a standard UART. The target application or monitor program reads and writes its status, transmit, and receive registers like any other UART.

The read-only style permits full bidirectional communications without target-write cycles. The target "transmits" by performing a particular sequence of reads from selected memory addresses. The accessed addresses and their order encodes the transmit data.

Read-only ports are useful if the target is incapable of writing to code space. Some Harvard-architecture processors don't have opcode for writing to their code spaces. Other targets specifically exclude write cycles from the EPROM-socket chip selects. Still other targets use single-direction data buffers between the processor and the EPROM socket.

Another read-only-style advantage is the ability to communicate without the target modifying its code space. This feature may be desirable or even required during acceptance testing of some products.

Live access offers the ability to read and write the target's memory while the target is executing from that same memory without imposing any restrictions on the target's accesses. This important feature enables many new debugging techniques, especially for real-time targets that can't tolerate breakpoints or single-stepping.

When looking at emulator specifications, evaluate this one closely. Sev-

eral methods let the host access target memory while the target is running. Side effects vary, depending on how the manufacturer achieves this goal.

Some emulators depend on target arbiters or wait-state hardware for arbitration. This approach may work for particular targets, depending on the availability of these resources, the target documentation available, and the firmware's tolerance of intrusions.

Other emulators impose additional restrictions on the target's timing at the EPROM socket, such as a minimum recovery time between accesses.

A good memory-interface debugging tool provides memory emulation and the support features that fit the mechanical, electrical, and timing characteristics of an EPROM. To achieve this, they cannot place any restrictions on how the target uses them. Note that an EPROM is always ready and has no recovery-time restrictions.

When shopping for an emulator with a live-access feature, look for one that is completely transparent to the target. Verify that the live accesses work in any target designed to accept standard EPROM or flash devices, without imposing additional constraints. For maximum flexibility and compatibility, avoid wait-states or bus-request/grant approaches.

From now on, 1'11 assume your emulator allows live host-side access without any restrictions on the target-side accesses-this is very important.

The target fetches opcodes, operands, and data from the emulator. No processor double reads to verify the data's integrity or checks semaphore flags before starting an opcode fetch. We want to asynchronously access the memory without coordinating with the target application.

Hardware trace is a new, advanced feature for memory emulators. Trace is similar to a logic analyzer, but specifically designed for firmware debugging.

It lets you trigger on single-shot-type events and shows you the events that led up to the trigger condition. Trace allows you to trap things like a target Write to address 0x8000 with data greater than 0x0C.

A trace option is always connected to the target and properly set up to

Listing I--These routines log trace *and status information to data structures in the target's memory space. A live access-type memory emulator can watch these structures to aid debugging.*

```
#include <stdio.h>

#define  OVERFLOW  1
#define UNDERFLOW 2
#define NULL-ENTER 3
#define NULL-EXIT 4
#define FREEZE     while(1);
typedef struct WATCH {
  unsigned char current-id:
  unsigned char curr_call_depth
  unsigned char max_call_depth;
  unsigned char error:
  unsigned int  current_sp;
  unsigned char ring-cursor
  unsigned char id_stack[16
  unsigned int  id_count[16
  unsigned char id_ring[256
};
void live_init(void);          // local prototypes
void push(unsigned char c);
void pop(unsigned char c);

static struct WATCH *live:     // local variables
unsigned int *nptr;

void live_init(void)
{
  unsigned int i;
  nptr = NULL:                 // intentional NULL ptr assignment
  live = (struct WATCH *)0xf0000;  // place log data in unused
                                   // data space
  live->error = 0;             // initialize everyth ng
  live->curr_call_depth = 0;
  live->max_call_depth = 0;
  live->current_id = 0;
  live->current_sp = 0;
  live->ring_cursor = 0;

  for (i=0; i<16; i++)
    live->id_stack[i] = 0;
  for (i=0; i<16; i++)
    live->id_count[i] = 0:
  for (i=0; i<256; i++)
    live->id_ring[i] = 0;
}

void push(unsigned char c)
{
  live->current_id = c;        // update current ID
  live->current_sp = _SP;      // and current SP
  live->id_ring[live->ring_cursor++] = c; // update the call ring
  live->id_stack[live->curr_call_depth++] = c: // update call-
                               // stack tracking
  if (live->curr_call_depth > live->max_call_depth)
    live->max_call_depth = live->curr_call_depth;
  if (*nptr != 0){             // somebody trashed 0
    live->error = NULL-ENTER:
    FREEZE:                    // freeze system to see who did it
  }
  if (live->curr_call_depth >= 16){
    live->error = OVERFLOW;
    FREEZE:                    // catastrophic error. freeze!
  }
  if (live->id_count[c] < 0xffff) // update the count for this ID
    live->id_count[c]++;
  else
    FREEZE:                    // freeze to evaluate
}

void pop(unsigned char c)
{
  live->id_ring[live->ring_cursor++] = (c|0x80);// updatecall
                                                // ring
  if (*nptr != 0){             // somebody trashed 0
    live->error = NULL-EXIT;
    FREEZE:                    // freeze system to see who did it
  }
  if (live->curr_call_depth)
    live->curr_call_depth--;
```

(continued)

Listing 1-continued

```
  else {
    live->error = UNDERFLOW;
    FREEZE;                        // MAJOR PROBLEM, freeze!
  }
}
```

interpret signals in target terms. You make trigger specifications and trace filter criteria from address, data, and control-signal values, ranges, or edges.

Since the trace board connects to the memory socket, it sees the same address, data, and control lines the memory device sees. The address and data signals are usually common to the rest of the circuit. The analyzer has a more global view than you'd think.

Additional trace-option inputs let you trigger on other chip selects or I/O signals. Since these devices usually share the same address and data lines as the EPROM, it's easy for the trace board to trace data, I/O, and code accesses.

You can also connect upper address lines to the extra inputs. This action is rarely necessary after you have the select circuitry debugged.

Finally, some memory emulators let you customize them with your own code. Some manufacturers provide libraries or developers' kits for you to write your own programs. The more advanced the emulator, the more valuable this feature becomes. A live-access emulator with a trace that allows custom program control is the ultimate debugging tool.

DEBUGGING TECHNIQUES

We engineer code to minimize the possibility of errors-but bugs occur every time. When we debug firmware without expensive tools, we usually fall back on several basic techniques.

Smaller incremental steps isolate bugs to specific sections of code. Small test routines reveal how different parts of the target act and interact.

Code patches wiggle spare port pins and flash LEDs. They dump messages out serial ports. Using other techniques, firmware generates nonstandard bus cycles (like writes to EPROM or nonexistent I/O devices) which are benign to the target, but create specific stimuli that a scope, logic analyzer, or other hardware can capture.

These techniques communicate debugging information to the outside world via basic "I made it to here" trace information or an alarm signal like "I received a null pointer."

Memory emulators offer a quicker "spin" on code changes. With the shorter cycle, you can try things you wouldn't if you had to burn EPROMs. It's now feasible to make test patches to your source code and observe the results in little more time than it takes to run a new make.

With a memory emulator, you think about debugging differently. For example, you can move the old "Blink the LED (hardware trace device]" statements around in the code to figure out where the firmware starts thinking on its own.

You can comment out pieces of suspect code, hard code configuration items, or change timing-loop constants. It's possible to freeze the program with **Ha** 1 t instructions at critical or informative locations. Quick code updates let you add tests to the source code instead of patching object code.

Memory emulators enable you to quickly test what-if scenarios. This basic troubleshooting technique is still a major step up from dry-run desk checking.

In addition, with simple and low-cost memory emulators, you can directly test your development system. By eliminating the need to schedule time on the ICE or wait on EPROM erasers or programmers, you can often work entirely from your office.

It is quite possible (but painful) to debug entire projects with nothing more than a memory emulator and an LED. You find yourself saying, "If only I knew what x is at this point" or "Is the FIFO buffer overflowing?"

You find answers to these questions by making patches to the code. You can turn on the LED if x exceeds a specific value or if the FIFO pointer nears its overflow point.

However, it's more useful to see these variables (and many others) to determine just how bad they are or exactly how they got in that shape.

STATIC BREAD-CRUMBING

Memory emulators with target write-back capability add several debugging techniques to our toolbox. First of all, target write back lets the target application write debug messages into unused memory locations.

When the target goes stupid, you can hold it in reset and read the emulator's memory to view those messages. I call this "Static Bread-crumbing."

The target application leaves bread crumbs sprinkled throughout memory to help you determine where it left the road. Other bread crumbs give us clues about why the firmware is in the weeds. Static bread-crumbing is an advanced version of "Flash the LED."

With basic EPROM emulators, we hypothesize the problem's cause based on very few observable symptoms—perhaps only the LED. We plan how to test that hypothesis, patch the code, and observe the results.

After several iterations, we may see that the code froze in `read_e e ()`, for example. So, we shift our focus to that routine. After more iterations, we determine that the code is waiting for the EPROM to set its ready bit.

Eventually, we find we violated a minimum clock width or setup parameter to the EEPROM, so it ignored us. To find the offending code, it takes several well-thought-out tests.

Target write-back capability lets us observe more data on each iteration. With more clues, you can make a more intelligent hypothesis for each test, reducing the number of iterations required to zero in on a bug.

However, a simple trick shows the offending routine immediately. If each procedure writes a unique ID code to a single memory location when it is first entered, then when the target freezes, this location holds the last active pro-

cedure's ID. A quick look at that location tells us which routine locked.

You can expand this concept by having each routine write its ID, the current stack pointer value, and anything else of interest to consecutive memory locations. Now you know what was running and what the last values of these critical items were.

Another simple trick is to assign a different bread crumb to each procedure. Each procedure increments its own bread crumb each time it executes. Postmortem analysis of these counts provides clues to unexpected interaction between modules or excessive use of error-recovery routines.

Call-stack tracking is another useful technique. It creates a call-stack structure that tracks stack activity.

In its simplest form, a macro or procedure call placed at the beginning of each procedure pushes the procedure's ID onto the simulated stack. Another macro before any return statement pops the simulated stack.

This call-stack structure reveals the sequence of calls causing the problem, as well as the maximum call depth encountered. This structure often reveals unexpected interaction between procedures, unplanned recursion, or circular references.

Another useful bread crumb is a call-history buffer, which is simply a circular buffer that holds the IDs of the last *n* called procedures. A single pointer shows the last ID entered.

If the emulator is in the target's RAM socket, the program's variables are there for postmortem inspection. You can also achieve this by changing the target's decode circuitry to map code and data into the emulator.

A target often has unused memory space you can decode into the emulated space. The emulator then provides extra writable memory space for debugging, even if the application fills its normal ROM and RAM space.

DYNAMIC BREAD-CRUMBING

If you're lucky enough to have an emulator that gives truly nonintrusive, live access to the target's memory while the target is running, you then have many powerful debugging techniques.

Listing 2—*This* test *program intentionally references a null pointer to demonstrate how a live-access-*capable emulator *helps track bugs. Figure 1 shows the* trace information generated *by this program.*

```
#include <stdio.h>
extern void live_init(void);        // external prototypes
extern void push(unsigned char c);
extern void pop(unsigned char c);
void init(void);                    // local prototypes
void get_ee_byte(void);
void get_ee_bit(void);
void poll_1(void);
void poll_2(void);
void poll_3(void);

#ifdef NDEBUG
#  define LOGIN(id) ((void) 0)
# define LOGOUT(id) ((void) 0)
#else
#  define LOGIN(id) push(id)
#  define LOGOUT(id) pop(id)
#endif

unsigned char *p;                   // local variables

void main()
{
  p = NULL;     // to prove the point, we force p to point to NUL
  live_init(); // initialize data logging
  LOGIN(1);
  init();
  while (1){
    poll_1();
    poll_2();
    poll_1();
  };
  LOGOUT(1);

void init()

  LOGIN(2);
  get_ee_byte();
  LOGOUT(2);
}
void get_ee_byte()

  unsigned char i;
  LOGIN(3);
  for (i=0; i<8; i++)
    get_ee_bit();
  LOGOUT(3);

void get_ee_bit()
{
  LOGIN(4);
  LOGOUT(4);
}
void poll_1(void)
{
  LOGIN(5);
  poll_3();
  LOGOUT(5);
}
void poll_2(void)
{
  LOGIN(6);
  poll_3();
  *p = Oxaa;    // intentional use of a NULL pointer!
  poll_3();
  LOGOUT(6)
}
void poll-3 (void)

  LOGIN(7);
  LOGOUT(7);
```

You can stimulate the target and observe the results in real time! And, you can watch the firmware's data structures while the target is running.

And, you can use these techniques in real time, rather than in postmortem. You can look at the stack-tracking structures and other bread crumbs

without stopping the target. I call this "Dynamic Bread-crumbing."

If the target maps its variables into the emulator's space, you can watch them without affecting its performance. You can watch things like task queues, current task ID, error indicators, and FIFO and ring pointers, without adding code to the application.

It's important to understand what this can and cannot do for you. Any live viewing of a target's memory is a polled view of that memory.

You get to nonintrusively view trends, verify certain variables are staying within bounds, monitor statistics, or watch for error conditions. If you initialize FIFOs and other dynamic data structures to a known value, you can often tell how full they get during operation.

If your live editor lets you freeze the watch screen, you can even take snapshots of the current state of the data you are watching without affecting the target. You can evaluate the health of data changing too fast for analysis.

For example, if you're watching a ring buffer and its associated pointers, the data changes rapidly as the application fills and empties the ring. Freezing the display lets you evaluate the contents of the ring and exactly how full it is at any time.

A live polling of the emulation memory doesn't let you capture single-cycle events or create any type of trigger when the application accesses a specific memory location or writes a specific value to a specific location.

These results are usually obtained from check routines to the firmware that detect the error, write error-message bread crumbs, and optionally halt. The firmware actually traps the event. The live editor shows exactly what event happened and where.

If you're looking for the cause of mysterious changes to a memory location (stray bullets), you need a logic analyzer or a trace-board option with advanced triggering capabilities.

The key point is that live polling and editing of the emulation memory creates a nonintrusive window into the target's operation. Live access is distinctly different from hardware tracing and is a very powerful tool.

Dynamic stimulation makes use of an emulator's live editor to change memory while the-target is executing. This functionality simulates external conditions, devices, or data streams.

You can complete development of interface code without actually having the external equipment on hand. Even if you have the other equipment, this technique creates a more controlled test of the firmware's reaction to the external device.

For example, while debugging one end of a communication link, you can simulate receiving packets from the other end of the link. Simply insert the packet into your receive ring buffer and then adjust the ring pointers and associated flags to indicate you received the packet.

Remember, you can do this while the target is executing-if you're careful. When you set the Full flag, the target should react to the packet. Of course, this example assumes that

your protocol is packet oriented and that you're using large enough ring buffers to hold an entire packet.

By live writing to target memory, you can force error conditions that may be difficult to create in the outside world. In the previous example, you could intentionally create a bad packet and evaluate how the firmware reacts to it.

Live writing can be used to communicate with the target. If you want quick A/B comparisons between different buffer sizes, write code to an agreed-on location to tell the application which buffer size to use.

The application checks this location each time it uses that parameter. This approach is safer than patching the buffer size directly.

It's possible to do direct code patching, but it requires careful planning of the original code. Remember, you can't create critical regions by disabling interrupts. Nothing holds off the live patch!

Virtual UARTs primarily provide a communications path for monitor programs or software debuggers. Software debuggers execute a small target monitor program to gain access to internal CPU resources and external memory and I/O. The monitor communicates with the debugger through a serial port or virtual UART.

The monitor is responsible for reading and writing CPU registers, CPU and target memory, and I/O spaces. It also handles single stepping and patching in software breakpoints.

The debugger uses the host computer's resources to cross-reference symbols and to interface with the user.

A software debugger is useful for single stepping, setting breakpoints, and viewing data structures. Many software debuggers include dynamic modes of operation that let them interrupt the target on a periodic basis and to update watches.

A memory emulator with a virtual UART and target write-back capability is an excellent enhancement to a software debugger. It eliminates two of the software debugger's most intrusive target requirements: the need for a serial port and extra debug RAM.

A live-access emulator and a software debugger are an unbeatable combination. The debugger lets you see high-level data structure and set breakpoints. The emulator minimizes the debugger's target resource requirements.

When the debugger lets the target run at full speed, it becomes invisible. At this point, the emulator's live editor can help with the other real-time techniques I've discussed.

```
current-id          07
curr_call_depth     03
max_call_depth      04
error               03
current_sp          eb02
ring-cursor         1d
id-stack            01 06 07
id-ring             0102030484048404
                    8404840484048404
                    8404848382050787
                    8506078707
```

Figure 1--The *routines in Listing 1 logged this trace and status information when Listing 2 was executed. This dafa enables us to track down the source of the null-pointer reference.*

For example, you set a breakpoint with the software debugger, start the target application, and then switch to the live editor. From the live editor, you force errors or simulate received packets and see if the program hits the breakpoint.

You then use the software debugger to look at variables in their native-language format. This combination gives you the best of both worlds—static and dynamic debugging.

However, some bugs are elusive. A hardware-trace option captures a subtle bug in the act, even if you have no idea which code module is responsible.

For example, it's difficult to determine exactly which line of code uses a null pointer. Without hardware trace, you'd place code at the start of each procedure to check the bottom of data space for corruption.

Most C compilers place a copyright or the word "null" in this location to reserve the space and allow them to check for corruption. Your code sets a bread crumb and freezes the program if it detects a change to this location.

This technique narrows down the suspects to something in the procedure that calls the current one. Of course, this only captures writes that land in writable memory space. Reads won't trigger an error.

Also, this technique only works if the write actually destroys data within the range being checked. Checking an extended range becomes too intrusive to the target's operation.

With a hardware trace, simply trigger on any access to the start of the data area. A hardware trace device captures the events leading up to the illegal access. Check the trace-buffer contents and call-stack bread crumbs to see who is responsible and what call sequence leads up to the problem.

The trigger-out from the trace generates an interrupt to the target when the target accesses any location within the specified range. If you're using a software debugger, the interrupt causes a break.

The debugger references the instruction pointer back to the source code so you see the offending code immediately. Most debuggers also let you look at the call stack at this point.

You can effectively use a hardware-trace option with a software debugger that doesn't necessarily support trace itself. The debugger may not interpret the trace buffer information for you, but the data is still available.

Additionally, you can use the trace board's triggers to implement hardware breakpoints for the debugger.

Even without a software debugger, you can trap the interrupt and bread crumb the stack and register contents for analysis or perform other actions you feel appropriate.

THEORY IN ACTION

Listing 1 offers routines that embody some of these techniques. Of course, you may want to expand or condense them to fit your particular application. Listing 2 is a minimal program designed to create bugs.

You can leave these routines in your source code without affecting the performance of the final production code. I modeled the functionality of the ASSERT macro supported by most C compilers.

You can disable these macros by defining NDEBUG. If you do not define

NDEBUG, the compiler includes the code. This code is in C, but the concepts port to any language.

If you include these macros *as you write* your code, you'll find it's painless to build in this type of debugging. When you start debugging, you can refine exactly what data the push() and pop() routines store and what errors they check.

When enabled, these routines affect the overall performance of your target program. The degree of this impact depends on how much data you log and how often you call the routines. However, logging at full memory bandwidth minimizes the intrusion to an acceptable level for most applications.

Note that in push(), *you* can check a variety of error conditions and react to the errors as you see fit. If you're looking for catastrophic errors (like a stack overflow or null-pointer usage), you may freeze the program to preserve the current state of all memory.

Freezing the program eliminates the possibility of the processor writing stray bullets into memory. When you freeze the application, you may want to disable interrupts so interrupt service routines don't change things. Of course, you can log more or less data as you see fit or check for specific error conditions unique to a particular bug.

Listing 2 shows a simple skeleton of procedure calls. It serves no real purpose, except to demonstrate the use of these debugging techniques.

I intentionally referenced a null pointer in the poll_2() routine to demonstrate bug tracking with a live-access memory emulator. The program freezes when push() detects a corruption to the bottom of data space.

Photo 1 shows the results of running the program on a '486 target while monitoring the debug structures with a live-access memory emulator. Figure 1 offers the logged data.

This example corrupted the bottom of data space in poll_2(). The data structures lead us to the culprit as follows.

The error tells us that we detected a null-pointer corruption on entry to a routine. Current_id tells us that we detected this problem when we entered poll_3().

Id-stack showsusthat poll_3() was called by poll_2(). This information narrows our focus to something that happened after entering poll_2() and before entering poll_3().

In this simple example, we can easily review poll_2() and find the problem. In a more complex routine, we may need to add more trace information to narrow the search.

The id_ring structure provides additional clues. The data shows that we successfully called and returned from poll_3() once while in poll_2() before seeing the null-pointer error.

Since we look for null-pointer usage at the entrance and exit of each routine, we know that the previous call to poll_3() did not cause the problem. Therefore the problem occurred within poll_2(), after the first call to poll_3() and before the second call to poll_3().

The push() and pop() routineslet you find the cause of stack overflows, null-pointer usage, or unexpected recursions. Simple modify these routines to look for clues to any specific bug you are tracking.

These same routines also provide a framework for code profiling. If you have a live-access-type memory emulator with a documented interface library, you can write a simple polling routine to read these data structures at fixed intervals. Statistics gathered from this data tell you the average and maximum call depth and the relative time spent in each routine.

EMULATORS IN YOUR TOOLBOX

I've introduced a number of general-purpose techniques for using memory emulators for both static and real-time debugging. Many of these techniques are simple variations on old themes. Others are only possible with the new live-access emulators.

Regardless of what tools you have, even the simplest EPROM emulator has its place in your toolbox. You can get one for less than $200, and it complements other debugging tools well.

More advanced emulators provide hardware support for software debuggers. You get a development system that more easily adapts to changes in CPU technology.

Top-end emulators provide live watching and editing of the target's memory, enabling debugging techniques previously only available with top-end, highly CPU-specific ICEs. Emulators capable of live editing are available for under $600.

Of course, these devices and techniques are only tools. Their real usefulness depends on your understanding of how to use them. The answers they provide are only as good as the questions you define. ❑

Jerry Merrill is the founder and CEO of TechTools, a company that specializes in memory-emulation devices. Jerry has been involved in electronics for over 20 years. You may reach him at 74014.3223@compuserve.com.

SOURCES

PromICE (EPROM/flash emulator)
Romboy (EPROM emulator)
Grammar Engine, Inc.
921 Eastwind Dr., Ste. 122
Westerville, OH 4308 1
(614) 899-7878
Fax: (614) 899-7888

HT Series (EPROM emulators)
Hobby-Tek
P.O. Box 462085
Garland, TX 75046
(214) 272-0202

Romem (EPROM emulator)
MicroSystems Development
4100 Moorpark Ave., Ste. 104
San Jose, CA 9.5 117
(408) 296-4000
Fax: (408) 296-5877

UniROM (Live-Access EPROM/flash emulator)
EconoROM (EPROM emulator)
TechTools
P.O. Box 462101
Garland, TX 75046
(214) 272-9392
Fax: (214) 494-5814

I R S

401 Very Useful
402 Moderately Useful
403 Not Useful

In-Circuit Emulators

Part 1: Development Tool Options

Debugging code for an embedded controller can be a real nightmare. So, when the sweat is really breaking across your brow, it's handy to have the latest on what types of debugging tools are available.

FEATURE ARTICLE

Graham Moss, Ross McMillan & Ken Mardle

These days, we assume that even the most mundane electronic equipment contains a microcontroller executing some sort of control program.

Users of such equipment expect increasingly sophisticated features, which in turn require increasingly capable microcontrollers executing increasingly complex code. To make matters worse, commercial considerations dictate ever shorter development cycles.

Fortunately, the increased processing power of modern microcontrollers and parallel advances in software-generation technology typically allow control programs for microcontrollers to be implemented with standard high-level programming languages like C, Pascal, or Modula-2. Large, complex control programs are now designed and coded in relatively short periods.

Unfortunately, as any seasoned development engineer knows, improvements in code-generation tools are only half the story-there remains the human element. Developing code for an embedded microcontroller can be a real nightmare.

Developers still make mistakes and omissions both in terms of design and implementation. The number and complexity of such errors roughly follow the magnitude and complexity of the overall control program. They are exacerbated by the complexity of the target microcontroller.

Indeed, the benefits high-level-language code-generation tools bring can be their greatest liability. They provide an efficient way to generate large amounts of complex buggy code.

In the past, time in design and coding dominated the overall development cycle. Now, more time is spent testing, debugging, and qualifying. So, it's critical that the overall development effort be similarly well supported with efficient debugging tools.

When searching for that last elusive bug, your debugging tools can either help or hinder depending on their quality, flexibility, functionality, and suitability of purpose. In this series of articles, we'll describe the types of debugging tools available and compare their strengths and weaknesses using the yardstick of transparency.

Afterwards, you'll be in a good position to see beyond the advertising hype and to sensibly evaluate the genuine strengths and weaknesses of the myriad debugging tools available.

The series concludes with a description of the design of the hardware and software components of PDS51-a popular low-cost in-circuit emulator (ICE) system designed by Philips to support development using their wide range of 80C51-based microcontrollers.

WHAT'S NEEDED

Debugging tools should be as transparent as possible. They shouldn't have side effects which mask a problem or introduce new ones. Users of debugging tools should expect them to place as few hurdles as possible between themselves and the solution of their problem.

A wide range of debugging tools is available today. Each vendor's solution represents a different set of design goals and compromises intended to address different types of debugging problems.

Debugging systems differ in terms of the scope and power of the control and monitoring functions they offer. And, just as importantly, they differ in terms of the microcontroller resources they "steal" to provide their control and monitoring functions, and the extent to which they interface with the real world beyond the pins of the microcontroller.

Costs vary from less than a hundred dollars to the tens of thousands. While cost is important in any purchase decision, it should be kept in perspective.

A cheap debugging tool may hinder more than help. It can quickly end up gathering dust.

At best, you waste the purchase price of the unit. But, true costs may be much higher in terms of wasted effort (new equipment has a learning curve), missed deadlines, or-horror of horrors-bugs in production firmware (just imagine the thousands of masked ROM devices).

Cost alone may also not be a reliable guide to the performance of a particular vendor's offering. Advances in hardware and software technology, combined with low-cost avenues for marketing in journals and over the Internet, now allow debugging systems around $1000 to provide much of the capability and performance of older designs costing ten times as much.

YOUR WORST DEBUGGING NIGHTMARE?

An embedded system has a microcontroller at its core, but its presence is not necessarily obvious to the user. Unlike general-purpose computers, embedded systems are often characterized by having input and output systems which do not lend themselves to control by or communication with a human operator.

To make matters worse, the microcontroller itself does not often present program-address information or execution data to the outside world-all you get are the I/O port pins. Without special-purpose debugging tools, you are forced to infer the behavior of the control program from its response to state changes applied to input pins and the effect on the states of output pins.

All this assumes, of course, that you can get your control program into the program memory of your target microcontroller in the first place.

Development-type microcontrollers based on EPROM [both OTP and erasable) or flash-memory technologies allow this, but are by no means universally available for all families and variants of microcontrollers. In some cases, you are faced with developing a control program for a microcontroller, which is only available in production-masked ROM form.

Development and debugging using only user-programmable microcontrollers is at best extremely tedious and impractical for complex applications.

EMULATION AS THE SOLUTION

The key to efficiently debugging embedded systems is to use some means to externally control and/or monitor the activity of the microcontroller core. Usually, the resources of a general-purpose computer provide the necessary human interface.

At the simplest level, the developer submits a control program for execution by the microcontroller and that program is executed from the normal power-on reset condition. More sophisticated systems extend this concept to allow greater levels of monitoring and control.

Control features may include the ability to execute the control program instruction-by-instruction (single-stepping), halt execution at specific addresses (breakpoints), resume execution from an arbitrary address, and set registers or memory locations to specific values.

Monitoring features may include displaying the value of registers or memory locations or observing the order and timing of instruction execution (tracing).

Since the microcontroller in the target system does none of these things, we have to replace it for debugging purposes with something that performs all the normal functions of the replaced microcontroller and provides the various control and monitoring functions we require.

This technique is generically called ***emulation,*** although the term emulator now has a more specific meaning. We'll describe this definition in next month's article (*INK* 73).

CLASSES OF EMULATION

Although there can be significant overlap between various approaches to emulation, it's useful to divide implementations of the general technique into four broad classes of emulation devices:

- simulators
- memory (ROM) emulators
- monitor-based debuggers
- in-circuit emulator (ICE) systems

We'll discuss each of these in turn with an emphasis on their inherent strengths and weaknesses, rather than on the technical details of how each class is implemented. The strengths and weaknesses described are typical and may or may not be present in any particular implementation.

DEBUGGING VIA SIMULATION

For the purposes of developing microcontroller-based embedded systems, a simulator is a program which runs on a general-purpose computer and implements a virtual microcontroller which can be controlled and monitored using the keyboard and display of the host computer. Listing 1 gives a

Listing 1—*This 80x86 code fragment is* from an *8051 simulator. On today's high-end PCs, a simulator can execute faster than the microcontroller being simulated, but it cannot interact with the real world.*

```
; Simulate the action of the setting of the parity flag, which
; always reflects the parity state of the accumulator.

; Executed any time the accumulator is modified.

; Note that the definitions are opposite for the 8086 and 8051.
; The 8086 uses even parity.
; The 8051 uses odd parity.

; In the 8051, the parity flag occupies the LSB of the PSW

; Execution time is 140 ns on a '486DX2-66

update_parity:
  and   [psw], 11111110b; assume ACC has even parity
  mov   al, [acc]       ; get simulated ACC to AL
  or    al al           ;set 8086 parity flag from AL
  jpe   parity-exit     ; skip if parity even
  or    [psw], 00000001b; parity was odd-set bit
parity_exit:
```

code fragment from a simulator for an 8051-family microcontroller which runs on an 80x86-based computer.

Most simulators run within an integrated environment which provides ready access to other software-based development tools. These in combination allow entry and editing of the control program source and its translation into an executable form.

An important advantage of simulation is that debugging is not dependent on the availability of target system hardware. This advantage enables software to be well developed long before prototype hardware is available.

It also enables several members of a development team to work in parallel on different parts of a project where only a single target hardware platform is available for reasons of cost, bulk, or accessibility.

Simulation is especially useful in developing small, self-contained software components not highly dependent on the target system's specific hardware configuration. Typical examples are development of a random-number generator (RNG) routine or implementation of a DSP algorithm.

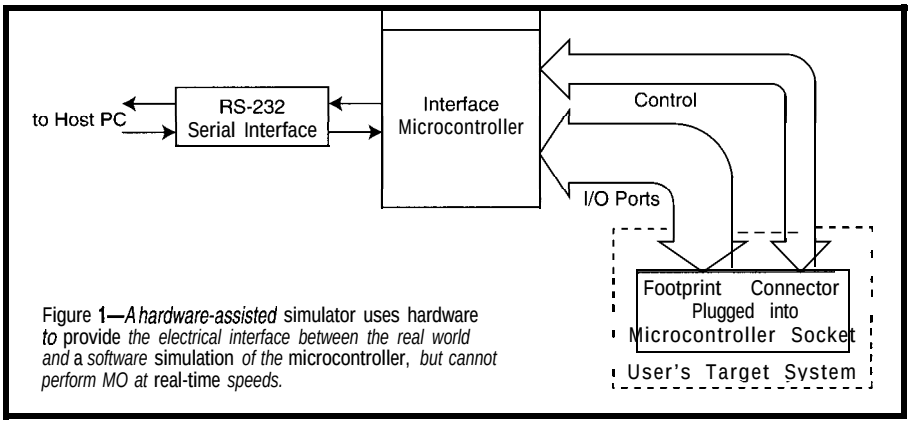


Figure 1—*A hardware-assisted* simulator uses hardware *to* provide *the electrical interface between the real world and a software* simulation *of the* microcontroller, *but cannot perform MO at* real-time *speeds.*

In these cases, the additional possibility of using the general-purpose computer to create test input data (e.g., to evaluate the performance of a digital filter) or to analyze the output data (e.g., to determine the randomness of the RNG) may be a real bonus.

The major problem with simulation as a debugging tool is that, by its very nature, it is divorced from the real world. It cannot readily be coupled with the taget system's hardware.

To offset this limitation, some simulators construct stimulus files. These files synchronize changes in input conditions as the simulation executes, generating log files to record the states of the simulated outputs and other conditions.

In simple situations, this is sufficient. But, it's largely useless when outputs can directly or indirectly affect inputs (e.g., when developing a driver to write and verify flash memory or routines to support CAN-bus network communications in automotive applications). In these cases, it's generally impossible to sufficiently simulate the external hardware environment.

This problem extends to the peripheral hardware resources of the microcontroller itself. Simulators are often limited to the basic instruction set, core registers, and internal memory of the target microcontroller.

Simulation of other subsystems—like timers, communications interfaces, and ADCs—is generally not supported. Many also ignore timing issues greatly limiting their usefulness in time-critical applications like engine management or AC-induction motor control.

In the past, speed of execution significantly limited the usefulness of software-based simulators, but this is no longer necessarily the case. On a Pentium PC, you can simulate the core instruction set of many microcontrollers faster than the real device.

HYBRID SIMULATORS

Some limitations of simulators interfacing with real-world hardware can be overcome with hybrid architecture. Here, the software simulator

combines with a hardware input and output interface unit, which plugs into the target hardware system in place of the microcontroller to be emulated (see Figure 1).

Communication between the host computer and the interface unit takes place via a standard serial or parallel port or a custom interface board in an expansion slot.

The control program to be debugged executes more or less entirely within the simulator on the host computer. The interface mainly provides the I/O pins of the emulated device. To ensure that the electrical characteristics of the emulated microcontroller closely match those of the real device, the interface unit is usually based on a microcontroller from the same family as the emulated device.

The interface microcontroller contains firmware that communicates with the host PC and executes I/O functions difficult or inefficient to simulate, such as analog-to-digital conversion, PWM waveform generation, and serial communications.

Although some peripheral hardware resources are successfully emulated with this hybrid approach, it doesn't work for all subsystems and functions. It's less than satisfactory for real-time I/O-intensive applications, such as timers (especially those with real-time capture and compare functions) and subsystems with interrupts.

Also, a hybrid simulator using a serial communications link to the host computer often steals the communication-interface hardware resources of the target microcontroller. So, it's difficult to debug a communications-oriented application.

A second problem with hybrid simulators is execution speed. While a lot of the core instruction set is simulated at real-time speeds, as soon as an instruction requires an I/O operation, speed dramatically reduces. The necessary communications transactions between the simulation program and the interface unit slow the simulator significantly.

Because they require no special hardware, simulators are traditionally one of the lowest-cost debugging tools. In the circumstances described above, and educational roles, simulators are very useful. Hence, they represent good value for the money.

Many vendors of more expensive hardware-based debugging tools recognize the special place of simulators. They often provide a means to use the software component of their products in a simulation mode.

But, it's less certain whether the improvements offered by hybrid simulators are worth the extra expense of the special interface hardware.

ROM EMULATORS

A ROM emulator is effectively a dual-port memory device. You see in Figure 2 that the target microcontroller has read access to the memory contents via one port and a host computer has write access via the other.

The host computer connects to the ROM emulator with a standard serial or parallel interface. It typically downloads program code for debugging in a few seconds. The ROM emulator nor-

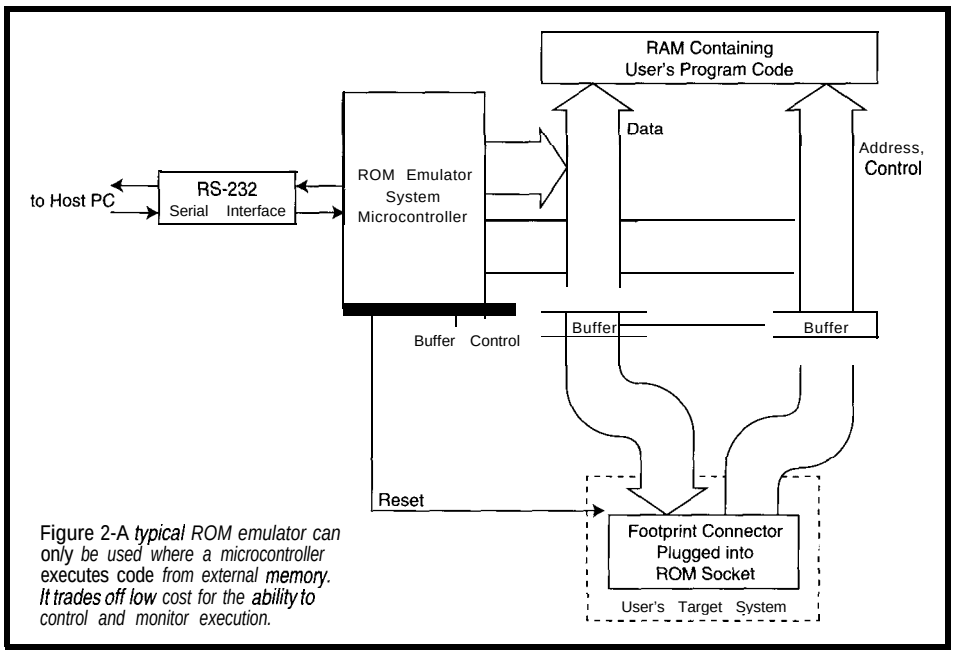


Figure 2-A *typical ROM emulator can only be used where a microcontroller executes code from external memory. It trades off low cost for the ability to control and monitor execution.*

mally controls the microcontroller's reset pin to prevent it executing meaningless code until the download process is complete.

Rather than emulating the whole microcontroller in the target system, the ROM emulator just emulates the memory device from which the microcontroller fetches its control program instructions. ROM emulators are therefore simple and relatively inexpensive-modern manufacturing technology produces ROM emulators no larger than standard memory devices that cost $100–200—but the savings is achieved at the expense of two major shortcomings.

First, ROM emulators require the target to fetch its control program from an external memory device.

Some microcontrollers operate in a single-chip mode, where the program and data memory is internal to the device itself and the majority of the device pins are available for I/O.

Others operate in an externally expanded mode. These emulators interface to external program and/or data memory where a substantial number of the device pins must be dedicated to providing the necessary data, address, and control interface.

If your target system is designed using an externally expanded architecture, it's no problem. But, a ROM emulator is of little or no use if your design requires the microcontroller to operate in single-chip mode in order to provide the number of I/O pins you

need. It also won't help if your microcontroller doesn't support external expansion (typical of low-end devices in packages of fewer than 40 pins).

In a target system where a microcontroller is in single-chip mode and supports external expansion, you might temporarily do without the I/O pins that interface external memory (the ROM emulator) while debugging. However, this is inconvenient at best.

Some manufacturers produce special development devices with a socket for a memory device containing the control program piggybacked onto the package. But, these devices are relatively uncommon and may cost as much as the ROM emulator itself.

The second major shortcoming of emulators is that they only provide a means to submit program code to the microcontroller for execution. Control is limited to holding the microcontroller reset during the download process-something your target system must tolerate without upsetting ancillary subsystems-and then releasing it on completion. Execution then begins from the standard reset point.

From then on, you cannot influence the program's operation or monitor its activity. The emulator has only sped up the process of program submission (which might otherwise require a tedious burn-test-erase sequence using erasable EPROM devices).

Some vendors attempt to offset this shortcoming by including additional memory capacity in parallel with that required for the control program. With companion software on the host computer, flags are set in that memory for ranges of program addresses, which can in turn trigger an oscilloscope or other item of test equipment. You can then determine whether a particular portion of a program is being executed, verify order of execution, and time the execution duration.

Some vendors have added logic-analyzer-type hardware. This hardware monitors and records program addresses and other external logic states as instructions are fetched. It displays execution traces and profiles.

Of course, these features inflate price and bring the ROM emulator into competition with the other more capable emulation tools discussed in subsequent sections of this article.

Recent advances in microcontroller fabrication technologies have in some cases rendered the ROM-emulator concept irrelevant. Some microcontroller devices now have internal program memory implemented using electrically erasable nonvolatile memory technologies (e.g., flash memory). Code downloads directly into the device in situ in the target.

ROM emulators undoubtedly have a place in some applications, but they are better suited to traditional multi-chip architectures than single-chip microcontrollers. Apart from low cost, their other advantages include:

- universality-they can be used with any externally expandable microcontroller or replace a ROM device
- ability to be ganged for wide-memory systems
- high speed and large capacity-units with megabytes of memory and access times in the few tens of nanoseconds are readily available
- minimal power requirements-normally power can be derived from the target system

Next month, we'll discuss more sophisticated tools including monitor-based debuggers and true in-circuit emulators. ❑

Graham Moss is a design engineer with the applications laboratory of Philips New Zealand, which designs and markets a variety of low-cost development tools for microcontrollers. He may be reached at graham8 pds.co.nz. The laboratory's web site is at http://www.he.net/~pds/.

Ross McMillan and Ken Mardle are design engineers with Applied Digital Research, a New Zealand company specializing in embedded systems and development tools. You may reach Ross and Ken at info@adr.co.nz.

IRS

404 Very Useful
405 Moderately Useful
406 Not Useful

FEATURE ARTICLE

Jim Sibigtroth

A Single-Wire Development Interface

The speed, pin count, and external addressing of new MCUs defeat old debugging methods. Jim overcomes such problems with the new single-wire background debug mode interface on Motorola's latest.

As new MCUs are introduced, some of our old debugging methods are running into big problems. Pin counts are going up, so emulators that clip over or replace the main MCU are becoming more mechanically unmanageable and/or ridiculously expensive.

Higher bus speeds are very difficult to emulate through an umbilical. Single-chip MCUs with large internal-memory systems don't even have external address and data buses, so logic analyzers and ROM-replacement approaches don't always work.

Serial background debugging can overcome these problems. In "No Emulator? Try a One-wire Debugger" (INK 54), Hank Wallace shows the need for a one-wire debugger and does a good job of using the materials at hand in a small system.

A small piece of code was embedded into the application program to send desired debug information out on a single output pin. A very simple connection to a PC parallel port and software in the PC monitored this debug information.

While this approach is great for getting you out of a jam, it's better if you don't need to add the code in the application system or share the output pin.

It's even better if the debug link is bidirectional, so you can interactively debug the target application.

Enter the new single-wire background debug mode (BDM) interface on Motorola's newest family of MCUs. Unlike previous background systems, the MC68HC12 uses a single dedicated pin and no target-system resources.

Target-system memory can be read from and written to while the target application is running, without affecting real-time operation. The application system and the debug interface are completely independent.

The system understands a complete set of primitive debug commands from memory modify to single-instruction trace. Everything you expect in the debug monitor of an emulation board can be implemented with this single-wire interface.

BIRD'S EYE VIEW

Before getting into the details, let's look at the whole system-from the PC to the target system under development. As shown in Figure 1, an interface pod acts as a translator between your terminal or PC and the target application system. The pod serves two main purposes.

First, the pod contains firmware to talk to the custom serial protocol of the MC68HC12's single background interface pin (BKGD).

Second, the pod includes a monitor or ASCII command interpreter so the terminal or PC can issue commands to the target system and retrieve data from it.

The wiring is a simple RS-232 connection between the host PC and the pod and a 2-to-4–wire cable between the pod and the target system. Although only BKGD on the target MCU is used for communication, you still need a common ground connection.

The optional third wire connects to the target system Reset. This feature isn't absolutely necessary, but it provides a nice way to remotely reset the target system from your PC keyboard.

The fourth wire can steal V_{DD} power from the target system for the pod. If your application power supply can handle a few extra milliamps, doing this is more convenient than

providing a separate power source for the pod.

Typically, the pod contains an MCU such as the MC68HC912B32 and an RS-232-level translator chip. The 'HC 12's 'B32 version has 32 KB of flash and 1 KB of RAM, so there's plenty of room for a complete debug monitor similar to the Buffalo monitor for the 'HC 11.

With a full monitor in the pod, the host PC can be anything that can run a simple terminal emulator. You don't have need a fancy debugger for your PC.

If you want to interface a commercial debugger program with a nice GUI, the pod implements a standard set of serial debug server commands instead of the monitor program. The M68SDI is an example of a pod with standard server commands. It's more user-friendly, but it costs more and ties you to a specific host such as a Windows-compatible PC.

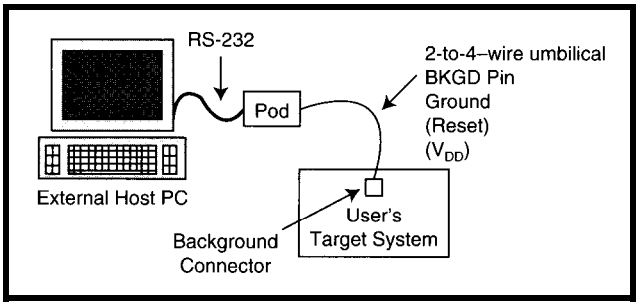


Figure 1-A *BDM system can use a standard RS-232 link to communicate between a host PC and the debug pod. Firmware in the pod interprets host commands and communicates with the user's target system through a 2-to-4-wire umbilical that connects to a background connector on the farget system. The reset and V_{DD} connections are nice but not required.*

THE BDM COMMAND SET

The 'HC12 background system understands a set of primitive commands which consists of 24 commands divided into two groups.

The hardware commands listed in Table 1 are implemented in dedicated logic and can be executed without disturbing the running application program. BDM logic looks for cycles where the CPU is not using the buses, so there's no real-time impact on the running program.

The second group of commands can't be done while the application program is running, like modifying CPU registers or tracing one instruction. Since the CPU is not executing user instructions at this time, it's available for use by the background debug system. Commands in this second group (listed in Table 2), are called *firmware commands* because they are implemented in a small BDM ROM.

To execute firmware commands, the application program is stopped and the MCU enters the active background mode. While BDM is active, a small BDM ROM is in the MCU memory map as well as a few BDM storage and control registers. These resources aren't visible when the user application is running, so they don't interfere with the application system.

The hardware commands include separate commands for accessing user memory and the BDM resources which occupy some of the same addresses. For instance, the BDM command READ_BD_BYTE @ $FF01 returnsthe same value in the BDM status register as is in the BDM memory map.

This BDM status register is not visible in the application program's memory map. The BDM command READ-BYTE @ $FF01 returnsavalue from the user's program memory.

SINGLE-WIRE COMMUNICATION PIN

To fit relatively high-speed bidirectional communication onto a single pin, a custom protocol was developed. To avoid a separate pin for a clock signal, the protocol designates the external host as the master.

All bit times are initiated when the external host drives a falling edge on the BKGD pin. Timing is controlled by the speed of the bus-rate clock in the target MCU.

The external master must either be told what this speed is by the user, or the master must determine the target speed before it can communicate. Each bit time is 16 bus clocks long. Ones are differentiated from zeros by the length of time the BKGD pin is held low.

The time between bits is arbitrary because the external host and the target MCU are asynchronous to each other. There is a maximum interbit delay, after which the MCU resets its BDM logic and starts looking for the first bit of a new command.

If the communication link ever gets out of synchronization, the host can simply stop driving the BKGD pin for at least 5 **12** target bus clocks to force this time-out.

Usually the external host is the device that drives the BKGD pin. But, when the host requests data from the target MCU, the target MCU also needs to drive this pin. The pin is designed as a pseudo-open-drain driver. When there's an unintended conflict, both the host and the target MCU drive the pin low, which avoids any electrical harm.

Normal open-drain pins tend to be slow due to R-C rise time, so this system is modified to provide brief active-high pulses to speed up the rise times. This design modification allows much faster communication than you'd normally expect to get from an open-drain pin.

When communication is from the host to the target MCU, the host gen-

	Opcode	Operands	Description
BACKGROUND	90		Enter active BDM if firmware enabled
WRITE-BYTE	C0	ah al ee xx (even) ah al xxoo (odd)	Write a byte of data to user memory
WRITE_BD_BYTE	C 4	ah al ee xx (even) ah al xx oo (odd).	Write a byte of data with BDM in map
WRITE-WORD	C8	ah al ee oo	Write a word of data to user memory
WRITE_BD_WORD	CC	ah al ee oo	Write a word of data with BDM in map
READ-BYTE	E0	ah al ee xx (even) ah al xx oo (odd)	Read a byte of data from user memory
READ_BD_BYTE	E 4	ah al ee xx (even) ah al xx oo (odd)'	Read a byte of data with BDM in map
READ-WORD	E8	ah al ee oo	Read a word of data from user memory
READ_BD_WORD	EC	ah al ee oo	Read a word of data with BDM in map

Table 1--The *BDM instruction set consists of the 9 hardware commands given here and the15 firmware commands shown in Table 2. Hardware commands may be executed without disturbing a running user program. A hardware command is used to set the firmware-enable bit.*

erates brief speed-up pulses at the end of each driven-low period.

It's a little more complicated when the target is sending return data to the host as shown in Figure 2. In that case, the host initiates the bit time by driving BKGD low just long enough for the target to recognize the falling edge.

Next, the target either reinforces the low, holding it for 12 cycles (for a logic zero), or allows the pin to rise (for a logic one).

In the case of the logic zero, the target drives the brief high speed-up pulse as soon as it releases its low drive to the pin.

For a logic one, the target waits for seven cycles and then drives a one-cycle-high speed-up pulse.

Timing is a little tricky. The special pulse comes after the host stops driving the low-bit-start pulse and before it samples the bit level at about cycle nine. The timing variation is due to the uncertainty of when the 'HC12 sees the falling edge at the start of the bit compared to when the host generates the edge.

Commands, addresses, and data are transferred MSB first. Each command starts with an 8-bit command word from the external host to the target MCU. Some commands nothing else.

Hardware commands that read or write memory require a 16-bit address after the command to identify the location in the target system to be accessed. If the command is a memory access, a 16-bit data word is sent to [writes) or received from (reads) the target after the address.

Data is always 16 bits to simplify the logic in the target MCU. When the command calls for only 8-bit data, the host is responsible for getting it to or from the correct half of the 16-bit data word, and the other half of the data word has a "don't care" status.

LET'S DO SOME WORK

Let's be very optimistic and assume you never have bugs, so you don't need to debug anything. This system is still useful in the manufacturing process.

You can build your whole system using unprogrammed flash memory. After final assembly, you can connect the pod and download your application software into the target system from your PC. This feature gives you more time to develop and test your code before committing to production.

The background system can access anything the target CPU can, so the nonvolatile program memory can be an external memory or an on-chip flash like the 32-KB flash in the MC68-HC912B32.

The background debug mode can also be used for system calibration. Imagine that your system has a sensor component that varies from one system to another. A correction factor stored in nonvolatile memory can compensate for the variations in sensors.

Rather than including a calibration program in your application code, you can monitor the operation of the sensor through serial debug commands. Next, you can program the calibration adjustment factors into the nonvolatile memory using other serial background commands.

Have you ever noticed how your auto-service dealer connects the service computer to a connector on your car and reads out information about recent system failures? This capability is supported by extra software in an embedded body-control processor.

The presence of the service computer enables software to respond to requests for service information. This capability implies that all the questions the service computer can ask were agreed on before the car was built.

Command Mnemonic	Opcode (hex)	Operands	Description
WRITE-NEXT	42	dh dl	X=X+2; Write word to 0,X
WRITE-PC	43	dh dl	Write user program counter
WRITE-D	44	dh dl	Write user D accumulator
WRITE-X	45	dh dl	Write user index register X
WRITE_Y	46	dh dl	Write user index register Y
WRITE_SP	47	dh dl	Write user stack pointer
READ-NEXT	62	dh dl	X=X+2; Read word from 0,X
READ-PC	63	dh dl	Read user program counter
READ_D	64	dh dl	Read user D accumulator
READ_X	65	dh dl	Read user index register X
READ_Y	66	dh dl	Read user index register Y
READ_SP	67	dh dl	Read user stack pointer
GO	08	—	Resume user program at current PC location
TRACE1	10	—	Trace 1 user instruction and return to active
BDMTAG_GO	18	—	Enable instruction tagging and resume user program at current PC location

Table 2—*To execute the firmware* commands in *the BDM* instruction *set, they must be enabled and BDM must be active* (user *program stopped).*

Now, imagine what you could do if the service connector was the single-wire background connector.

The embedded body controller still needs to record system failures in memory somewhere. But, you no longer need special routines to handle communication through the service connector.

This approach leaves a little more processor bandwidth for the actual application. It also relieves the application programmer from writing the communication routines.

Now, let's be more realistic and talk about debugging programs that don't quite work on the first try. You load the code into your system and try to run it. Some bug causes the program to fail.

If you're relying on a debugging routine in the application code to tell you what's going on, you may be out of luck because the bug(s) may cause this code to run incorrectly or not at all. In this environment, the code has to be almost working before you can use the debug routines.

With this new single-wire BDM, you connect the debug pod to the target system. Depending on how serious the bug is, you can try to start talking to the target without resetting it to see what it *is* doing (instead of what it ***should*** be doing).

If things are really bad, you force a reset and startup with the background mode active. (Don't run the user program.)

From here, you set the program counter to the start of the user pro-

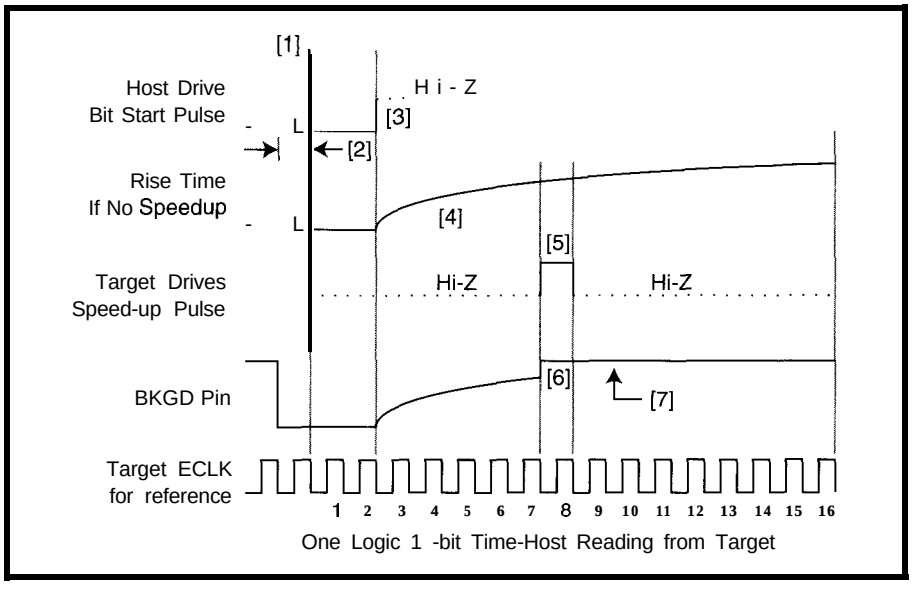


Figure 2—*The host reading a logic one bit from the farget is the* most complex *case. The hosf starts fhe bit by driving low [1]. The target fakes from 0 fo 1 bus cycles to synchronize fo this* falling edge *[2]. At [3], the* host *stops driving low and the pin begins a slow R-C rise [4]. At [5], the farget MCU drives a high speed-up pulse to snap the pin* to a logic one *[6].* This high is held by a weak external *pullup after [5]* stops driving. A short time later, the host *samples the bit level [7], but this* is referenced *fo [1] in the host and not to the farget E clock.*

gram and trace single instructions, checking the state of registers and memory between steps to see if everything has the expected values.

If this is too slow or if you are sure that certain parts of the program are okay, you can set a breakpoint after the known-good code and single-step the parts you are unsure about.

In previous systems, debuggers established breakpoints in application code by replacing an instruction opcode with the code for a software interrupt (SW I). The service routine for SW I would be a call to the debug monitor.

This method was intrusive because it interfered with any other use of the SW I by the application program. It also involved the addition of SW I service code for the debugger in the application program.

In the 'HC 12, the background (BGND) instruction opcode can be used instead of SW I. Since this opcode isn't used in normal application programs, it isn't intrusive.

The code that is executed to service the BGND instruction is in the BDM ROM, which is not part of the normal user map. So, it doesn't take up any space in the user's program memory.

THE FUTURE OF DEBUGGING

The debug capabilities offered by this new single-wire system are comparable to those you get in a full-featured emulator. However, in this system, the actual target MCU is not replaced by an emulator.

The background debug pod requires only two to four connections to the target system-and that's reasonable, even in the tightest applications. No debug code needs to be incorporated into the application program. The designer is free to work on the application instead of devising debugging routines.

As MCUs get more complex and packages get smaller, traditional parallel emulation techniques become less practical. Prior to serial background systems, most development approaches required access to address and data bus information.

The availability of large on-chip flash memory makes expanded mode systems less common. The need for serial debug systems like this one increases. Some new versions of the 'HC12 even include on-chip logic for hardware breakpoints, which simplifies debugging programs in flash.

The M68HC12 family represents another step in the evolution of serial background debug systems. Compared to previous serial background systems, the 'HC12 has fewer connections and accesses memory while the application runs in real time. The logic and resources for this system are separate and hidden from the normal application. They don't interfere with the real application.

Debugging microprocessor systems has evolved as these systems have become more sophisticated. Originally, few tools other than logic probes and oscilloscopes were available. Logic analyzers and in-circuit emulators were developed, but they require a lot of target-system connections.

Embedded systems continue to develop with faster clock speeds and higher-pin-count packages. To keep up, emulators and logic analyzers have become more sophisticated, and the connection systems have become more unmanageable.

Most of these tools are out of reach for a typical experimenter. While serial debuggers don't completely replace traditional parallel systems, they offer good debug capabilities, while eliminating some of the problems.

The simple serial connection eliminates a costly umbilical and is not required to operate at submicrosecond speeds. For low-cost debug, this new breed of serial debugger is the obvious choice. ❑

Jim Sib&troth is a system design engineer working on advanced microcontrollers for Motorola. His latest project was the 68HC12 where he was a co-architect for the CPU12 instruction *set. He devised the background debug system described in this article. You may reach Jim at jims@seasick.sps.mot.com.*

CONTACT

CPU12RM/AD,
MC68HC812A4TS/D,
MC68HC912B32PP/D
Motorola Literature Center
P.O. Box 13026
Austin, TX 78711
(800) 7657795, Ext. 950
Fax: (800) 7659753

I R S

407 Very Useful
408 Moderately Useful
409 Not Useful

Designing with Flash Memory

FEATURE ARTICLE

Arvind **Rana**

After bringing us up-to-date with what flash is and when it's used, Arvind probes the hardware and software issues related to designing with flash. The big issues come down to proper software algorithms.

Any day now, you won't be able to buy good old ROM chips and you'll have to use flash memory-whether you like it or not. It's a simple matter of economics.

Unless you have some legacy application and the government is paying you to buy the parts, you'll pick and use flash memory wherever you were using ROM before.

These days, flash often simply replaces the ROM. In this article, however, I'll be talking about using flash in its full functionality.

WHAT IS FLASH MEMORY

Flash is nonvolatile, erasable, writable, low-power, high-density, solid-state memory in a chip. It's a ROM, it's a RAM, it's a solid-state disk. But, it's smaller, denser, and less power-hungry than just about all of them.

However, the similarities stop there. The ROM, RAM, and disk drives have been around a long time. We know how to design with them, how to use them, and what their capabilities and limitations are.

None of these things are true for flash. Flash has gone through a lot of evolutionary change.

Flash devices have a certain peculiarity. Although they're writable, you must remember they don't operate like RAM. You may write bytes of data to the memory locations in the device, but to rewrite any data, you must first erase either the entire chip or the particular sector.

Also, when processing, the device is unavailable for all but status-read operations.

WHY FLASH

In deciding whether or not you should use flash, the relevant questions are:

- will flash devices make your products better, cheaper, and more competitive?
- will flash devices enable you to make products you couldn't make before?
- what are its real advantages?
- how can you exploit flash now that it's here?
- how are others taking advantage of flash and what are they doing with it?

When you see what flash is and how it works, you'll arrive at the answers relevant to your situation.

USES OF FLASH

The potential use of anything is unlimited. It's up to the user's imagination. But, there are things that flash is quite suitable for. Some of these innovations have yielded significant advantages.

For example, there are clear advantages to storing the firmware for a product in flash. Unlike ROM, flash is programmable in-circuit. The software can be updated in the field and customized to user needs on demand.

Flash is quite fast. Software can run at full speed directly from flash. One popular application in this area is BIOS storage. BIOS no longer has to be cast in stone.

Systems may need the BIOS updated to accommodate processor upgrades, support for newer graphics devices, power-management features, fixing of bugs, improvements in operation, or support for plug-in PCMCIA devices. In hand-held and portable systems, or in systems that are difficult or impossible to disassemble, flash nonetheless provides software upgrades.

For hand-held measuring devices or remote sensing and measuring devices, flash offers something not easily attainable before: nonbattery-powered, nonvolatile memory.

Flash reliably stores information. You don't have to worry about it being lost because the backup battery died.

It's also more immune to shock and temperature-related changes. And, it offers much higher density than other available technologies (e.g., SRAM or EEPROM).

Storing large amounts of data for lookup tables (e.g., a phone-switching application) enables the service provider to customize and store customer preferences. Other applications such as printer fonts, phone numbers in portable phones, address information in your PDA, and so on also need quick lookups.

Last but not least, flash memory is popularly used as a file system. Many custom and off-the-shelf flash file systems are available. These systems enable you to use flash memory as a solid-state disk.

Flash is blindingly fast and compact for the amount of storage it offers. It gives more convenience, more reliability, and lower power than a magnetic spinning medium. Very small and light portable computers obtain storage capability matching the much heavier and power-hungry portables of yesterday.

HARDWARE DESIGN ISSUES

The flash device you pick dictates the complexity of your design. The oldest ones simply marked as 28F5 12, 28F010, or 28F020 require 12-VDC programming voltage and externally controlled erase and programming algorithms.

These devices also feature bulk-erase. You must erase the entire device before writing individual bytes. These devices are most susceptible to flaws in hardware design and software algorithms. I don't recommend these for a new design.

Second-generation devices include several different products. Intel, the inventor of flash memory, offers several varieties of devices.

Highly specialized devices such as flash-file components and synchronous- and DRAM-interface flash-memory components are intended for PCMCIA and desktop or portable computing applications. In these applications, the DRAM-like interface allows easy integration of flash with existing CPUs.

However, the most interesting and popular components for embedded applications are the boot-block components offered by Intel and AMD. Intel parts are second-sourced by several other manufacturers.

AMD and Atmel offer additional simple sector-erase components. Atmel offers 5- and 3-V devices in densities of 32-5 12 KB. Their numerous, smaller sectors make them attractive alternatives to big parts from Intel and AMD, which generally offer parts from 1 Mb and up.

These parts are programmable in-circuit using only 5 V. Some operate at only 3 V.

The boot-block devices offered by both AMD and Intel in pin-compatible packages allow a hardware-protected boot block for your code. This block cannot be erased by any software algorithm. It is programmed on an external programmer.

There are two flavors of these devices: those with boot block at the top of memory and those with it at the bottom. These devices also feature parameter and code blocks. They come in both 16- and 8-bit sizes, with 16-bit ones being usable in 8-bit mode. I expect these to proliferate in embedded applications.

Read Reset	FO restores flash device to read mode
Auto Select	90 allows reading of manufacturer and device IDs
Byte Program	A0 programs a single byte
Erase	80 requires double-unlock sequence for chip or sector erase
Chip Erase	10 erases entire chip minus the protected sectors
Sector Erase	30 erases one or more sectors, won't erase protected ones
Erase Suspend	BO temporarily suspends sector-erase operation
Erase Resume	30 resumes sector-erase process immediately

Table 1-By embedding low-level write and *erase* algorithms inside *the* flash device, *AMD enables the* developer to execute previously *complicated flash functions by issuing a simple command.*

SOFTWARE DESIGN ISSUES

Of the two possible ways to program a flash, the first is to program it before it is installed in the target system. You'd use a device programmer (i.e., a ROM programmer) that can also program flash devices.

This way, the flash is programmed beforehand, and sectors are protected by applying the higher voltages needed for such tasks. This method is common when the flash replaces ROM in an existing application or when you're placing boot code in a protected block.

The second method is to program the flash in-circuit after the part has been soldered down or socketed. You accomplish this by issuing commands to the flash device with proper software algorithms.

Unfortunately, more than one type of algorithm is needed for the different vendors' parts. The algorithms used by AMD and Atmel parts go back to the days of the EEPROM, which is what flash really is.

To avoid inadvertent change in the memory data, an unlock sequence accesses the parts for erase and programming functions. The sequence consists of two write cycles with address/data:

5555/AA
2AAA/55

At this point, the flash device is ready to accept a command. The commands are given by doing a write cycle:

5555/CC

where CC is the command code in Table 1.

The chip- and sector-erase commands first require an erase command be issued. The actual chip- or sector-erase command then issues after

Read Reset	FF restore flash device to read mode
Auto Select	90 allows reading of manufacturer and device IDs
Byte Program	40 program a single byte
Erase	20 sector erase
Sector Erase	DO sector erase confirm
Erase Suspend	BO temporarily suspend sector-erase operation
Erase Resume	DO now resume sector-erase process
Read Status	70 read device status register on next read
Clear Status	50 clear device status register

Table 2—*Intel's flash commands are similar but not identical to AMD's. If you use flash devices from more than one manufacturer in your design, it's important to read device IDs and issue the correct commands for that device.*

another unlock sequence. This two-step unlocking sequence doubly ensures device integrity.

Instead of using the 5555 address for sector erase, it's an address within the desired sector. Multiple sectors are erased by issuing more sector-erase commands (minus the unlock sequences) before an internal timer of -80 µs expires. The device indicates the start of an erase cycle by setting a timeout bit.

The device generally goes busy once it accepts the erase or programming command. The target system reads a status register during these operations.

The status register reports on the progress of the process or failure conditions. When failures are encountered, the device is restored to normal operation by the Read Reset command.

Atmel devices feature a different write process. Since their sectors are only 256 or 512 bytes, the write process requires the entire sectors' worth of data to be written in successive cycles before a timeout occurs. With this method, the entire sector is programmed.

Intel devices, on the other hand, did away with the unlock sequences. Instead, most commands are one or two write cycles. The commands, listed in Table 2, are given as coded data bytes issued with relevant stable address.

The address portion of the bus contains either the byte, the sector, or some other stable address for the commands.

In addition, flash-file components and synchronous- and DRAM-interface devices support a 256-byte buffer to allow faster writes. Some fancier features allow operations for improved performance for these devices.

PROGRAMMING IN-CIRCUIT

Programming the flash in-circuit requires executing the algorithms from software running out of some memory device other than the flash being programmed.

The basic programming algorithms are simple. The error-recovery process, however, varies for each application. Generally, a true flash-device failure leaves part of the device unusable (e.g., a sector).

Listing 1—*Unlike traditional read-only memory, flash can be modified at any time from a high-level language like C. Using f/ash, a developer can provide periodic updates to a product's firmware or design a system that adapts to changing user or environmental requirements.*

```
/*********************************************
 *   Flash erase                             *
 *   Sector erase                            *
 *   Byte programming                        *
 *   Chip information:                       *
 *      Manufacturer code                    *
 *      Device code                          *
 *      Sector protection information        *
 *********************************************/

#include "80386ex.h"
#include <dos.h>
#include "exser.h"
#include <stdio.h>
#include <stdlib.h>
i/include "comm.h"

/*********************************xx*******
 * Sector addresses for a 29F010             *
 *********************************************/
#define SECT0 0xe0000UL
#define SECT1 0xe4000UL
#define SECT2 0xe8000UL
#define SECT3 0xec000UL
#define SECT4 0xf0000UL
#define SECT5 0xf4000UL
#define SECT6 0xf8000UL
#define SECT7 0xfc000UL

/*********************************************
 * Unlock addresses for a 29F010             *
 xx*********************************************,
#define ULOCKA 0xe5555UL      // First unlock'"5555"
#define ULOCKB 0xe2AAAUL      // Second unlock "AAAA"

/*************************************************
 * Pointers to beginning address of each sector,     *
 * base ROM address and unlock addresses             *
 ************************************x************,
typedef far unsigned char* FLASHPTR:

volatile FLASHPTR ulocka = (FLASHPTR) MK_FP((unsigned int)
        (ULOCKA >> 4), (unsigned char) (ULOCKA & 0x0f));
volatile FLASHPTR ulockb = (FLASHPTR) MK_FP((unsigned int)
        (ULOCKB >> 4), (unsigned char) (ULOCKB & 0x0f));
volatile FLASHPTR sect0 = (FLASHPTR) MK_FP((unsigned int)
        (SECT0 >> 4), (unsigned char) (SECT0 & 0x0f));
volatile FLASHPTR sect1 = (FLASHPTR) MK_FP((unsigned int)
        (SECT1 >> 4), (unsigned char) (SECT1 & 0x0f));
volatile FLASHPTR sect2 = (FLASHPTR) MK_FP((unsigned int)
        (SECT2 >> 4), (unsigned char) (SECT2 & 0x0f));
volatile FLASHPTR sect3 = (FLASHPTR) MK_FP((unsigned int)
        (SECT3 >> 4), (unsigned char) (SECT3 & 0x0f));
volatile FLASHPTR sect4 = (FLASHPTR) MK_FP((unsigned int)
        (SECT4 >> 4), (unsigned char) (SECT4 & 0x0f));
volatile FLASHPTR sect5 = (FLASHPTR) MK_FP((unsigned int)
        (SECT5 >> 4), (unsigned char) (SECT5 & 0x0f));
volatile FLASHPTR sect6 = (FLASHPTR) MK_FP((unsigned int)
        (SECT6 >> 4), (unsigned char) (SECT6 & 0x0f));
volatile FLASHPTR sect7 = (FLASHPTR) MK_FP((unsigned int)
        (SECT7 >> 4), (unsigned char) (SECT7 & 0x0f));
volatile FLASHPTR rombase = (FLASHPTR) MK_FP((unsigned int
        (SECT0 >> 4), (unsigned char) (SECT0 & 0x0f));

int Get__choice(void)
{
  int got_int;
  int x;

  do {
    x = inportb(COM2 + LSR);

  while (!(x & 0x01));   /* Wait until data comes */
  got_int = inp(COM2);
  printf("%c", got_int);
  return(got_int);
}
```

(continued)

Programming involves issuing the unlock sequence to the device and then issuing the command. The first command is typically Auto Sel ect, so that the proper device type and manufacturer is identified.

The code in Listing 1 is written in C. It runs on a '386EX target board with a 29F010 flash memory chip, from which it also boots. The program demonstrates the common algorithms for accessing the chip.

Error recovery is eschewed. You can download programming algorithms from various manufacturer's bulletin boards. This code is just an example.

PROGRAMMING VIA JTAG

Generally, once the algorithms are in, the flash can be programmed conveniently. But, how about programming a flash part that is already on the board and is totally blank?

This programming is accomplished over a JTAG port on the CPU, if your target system has one.

JTAG (Joint Test Access Group) developed a standard known as IEEE 1149.1. It's a serial interface to directly drive the pins of a CPU chip.

The internal circuitry provided on the chip to support JTAG executes CPU instructions by feeding them over the JTAG port. The port is a 4- or 5-pin connection with just as many pins for ground. You see the pins defining the JTAG interface in Table 3.

This interface is implemented variously as BDM (Background Debug Mode) and is also referred to as Boundary Scan.

The lowest-cost interface to the JTAG port is simply a cable connected to the parallel port of the PC and a piece of software feeding the bits to accomplish the operation.

You can practically run code out of a host over the JTAG on the target. It comes in handy when the target memory system is malfunctioning. But, in this case, it is used simply to program the boot code in a blank flash device.

The host-based software feeds and executes the proper instructions to program the flash memory in-circuit. Many debuggers support JTAG for debugging, and some let you program flash over the JTAG.

Listing 1-continued

```
/*********************************************
 * Writes Oxab to f0000 (sector 4 base add) *
 *********************************************/
void Byte_program(void)
{
    *ulocka = Oxaa:   // unlock sequence
    *ulockb = 0x55;
    *ulocka = 0xa0;   // Byte write command

    *sect4   = (0xab); // Write ab to address pointed to by sect4


/*********************************************
 * Chip erase function                       *
 *********************************************/
void Flash_erase(void)

    *ulocka = Oxaa;   // Unlock sequence
    *ulockb = 0x55:
    *ulocka = 0x80;   // Erase command
    *ulocka = Oxaa;   // Unlock sequence
    *ulockb = 0x55:
    *ulocka = 0x10;   // Erase chip
}

/*********************************************
 * Erase sector 4                            *
 *********************************************/
void Menu_sect_erase(void)
{
    *ulocka = Oxaa;   // Unlock sequence
    *ulockb = 0x55;
    *ulocka = 0x80;   // Erase
    *ulocka = Oxaa;   // Unlock sequence
    *ulockb = 0x55:
    *sect4   = 0x30;  // Erase sector 4


/*********************************************
 * Get chip information                      *
 *  Displays:                                *
 * Manufacturer code                         *
 * Device Code                               *
 * Sector protection information for sector 3 *
 *********************************************/
void Menu_chipinfo(void)
{
    volatile FLASHPTR tmp;      // Temporary pointer
    *ulocka = 0xAA;             // unlock sequence
    *ulockb = 0x55;
    *ulocka = 0x90:             // enter autoselect

    printf("\n\nManufacturer code = %x\n",*sect0);

    // Device code is located at offset 1 from beginning of ROM
    tmp = sect0 + 1:
    printf("Device code = %0x\n",*tmp);

    // point to sector-protection offset in sector 3
    tmp = sect3 + 2:
    if (*tmp == 0x01)
          printf("\nSector 3 Protected"):
    else
          printf("\nSector 3 Not Protected");
                                  // Put flash back in read mode
    *ulocka = 0xAA;               // Unlock sequence
    *ulockb = 0x55;
    *ulocka = 0xf0;               // Read/reset


void main(void)
{
    char buff[BUFFER];
    int  choice:
    for(;;) {
      printf("\f FLASH MENU\n\n");
      printf("   1   Erase Chip\n");
      printf("   2   Erase Sector\n");
      printf("   3   Chip Information\n");
      printf("   4   Byte Program\n");
```

(continued)

Listing 1-continued

```
        printf("        Enter selection:");
        choice = Get_choice();
        switch(choice) {
          case '1':
            Flash_erase();
            printf("\n\nFlash Erase Complete\n")
            break;
          case '2':
            Menu_sect_erase();
            printf("\n\nSector Erase Complete\n");
            break;
          case '3':
            Menu_chipinfo();
            break;
          case '4':
            Byte_program();
            break;
        default:
        printf("\n\n ERROR! WATCH YOUR FINGERS WHEN YOU TYPE ")
        break;

      printf("\nPress ENTER to continue");
      gets(buff);
```

You can also acquire specific software for JTAG operations.

HARDWARE HANDSHAKE

Most of the newer flash devices allow the target hardware to manage the flash properly during system reset or other recovery.

Instead of using the Read Reset command, a Reset pin on the device allows return-to-read mode when system reset is applied.

Also, instead of polling for a device to get done, a Busy/Ready pin interrupts the target CPU if necessary to indicate the device status.

DEBUGGING

In firmware development with flash memory, you have three separate tasks:

- code development
- flash programming algorithms and error recovery
- data storage algorithms

Pin	Description
TCK	Clock for data transfer
TMS	Selects mode to control which operation
TDI	Serial Data in
TDO	Serial Data out
TRST	Reset (Optional)

Table 3-A *microprocessor's JTAG interface can now be used to program blank flash devices after they are soldered into your target.*

Code development is the easiest part of using flash. You can pretty much ignore that you're using a flash device. It's only a place to store the finished and debugged software. Code in the flash ROM is no more peculiar than code anywhere else. Whatever techniques you use to develop your firmware work here just as well.

The software routines that erase and program flash in-circuit are quite critical and device-specific. The first-generation devices require controlling the 12-V (V_{PP}) programming voltage and software commands externally.

Voltage level and timing of operations (e.g., how many times to pulse the V_{PP}) are externally controlled and very sensitive to spikes or partial counts and restarts. This complexity must be maintained and controlled by the designer.

Fortunately, for most newer flash devices, the programming algorithm as well as programming voltages are internally controlled. The external software simply issues the appropriate commands and then monitors the status for completeness.

Keep in mind that flash becomes busy soon after a command is issued. Your programming algorithms need to run from some other memory device than flash.

Typically, you copy the algorithms out of flash to some other memory and

execute them from there. Failures here are caused by an interrupt routine requesting code from the flash to be executed. Appropriate handling of such conditions has to be part of the algorithms.

All write operations, like sector- or chip-erase, take time. Even simple operations such as writing a byte make the device appear busy. It is imperative that, for proper operation, the algorithms verify the operation.

Your system should include some sort of checksum for verifying that the operation completed successfully. On the surface it seems simple, but imagine a power failure during a flash programming cycle. Unless your software can verify the sanity of flash on power-up, the incomplete programming cycle may not be detected.

Unfortunately, no tools are available in the marketplace to tell you how and what you're doing to the flash. You pretty much have to try and see.

It can be very difficult to verify your algorithms and test error conditions and handling of device failures. The so-called boundary conditions are impossible to test since flash devices most likely won't experience any failures until they've been out in the field and used for a while.

That's when you find out that, because of faulty algorithms, the device's lifetime is severely reduced or specific locations are getting burned out. Before we dive into what kind of help is available for testing the algorithms, let's consider the third development issue related to flash.

Believe it or not, data storage is the main advantage and feature of the flash. Flash is intended for storing data more than code. As mentioned earlier, it provides high-density, low-power, nonvolatile data storage on-the-fly.

Generally, data storage is organized as a file system, but it doesn't have to be that elaborate. Very often, the implementation specifically addresses the product's need.

The main consideration here is that the flash device has a useful life. Its usefulness is measured in the number of erase/write cycles that can be done on a location. The numbers for older devices are on the order of 10,000 and for the newer devices are more like 100,000. In real life, you may get a million useful cycles out of the part.

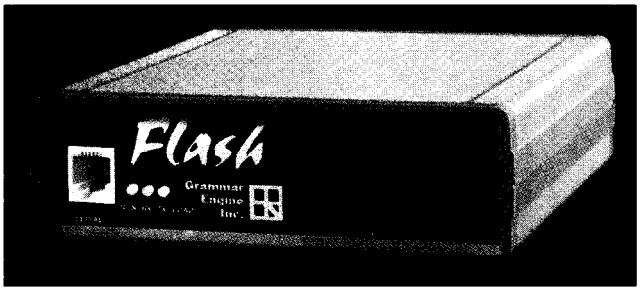


Photo 1—*Grammar Engine's FlashICE is the first memory emulator that* recognizes and *responds to* flash commands.

However, the software that deals with the flash must ensure that proper programming is achieved. Since frequently used locations may wear out, smart software includes "wear-leveling" algorithms.

It all boils down to this. How do you know that you're doing all the right things in the software-everything from proper commands to the flash, proper handling of the error conditions, and proper wear-leveling?

The answer: you can't-at least not until now.

FLASHICE

FlashICE (see Photo 1) is a ROM emulator with special circuitry to emulate flash functionality. As a new product, it only supports 29Fxxx parts.

Since the software access to the embedded algorithms follows a state machine, it's possible to emulate and verify proper access to the flash. This emulator plugs in where the flash normally goes and then behaves like flash memory.

FlashICE is unique because it not only emulates the ROM, it also monitors and reports the flash functions. It effectively provides a back end to the flash device.

The interface to the host system lets the user monitor all flash access as well as setup error conditions. The

#121

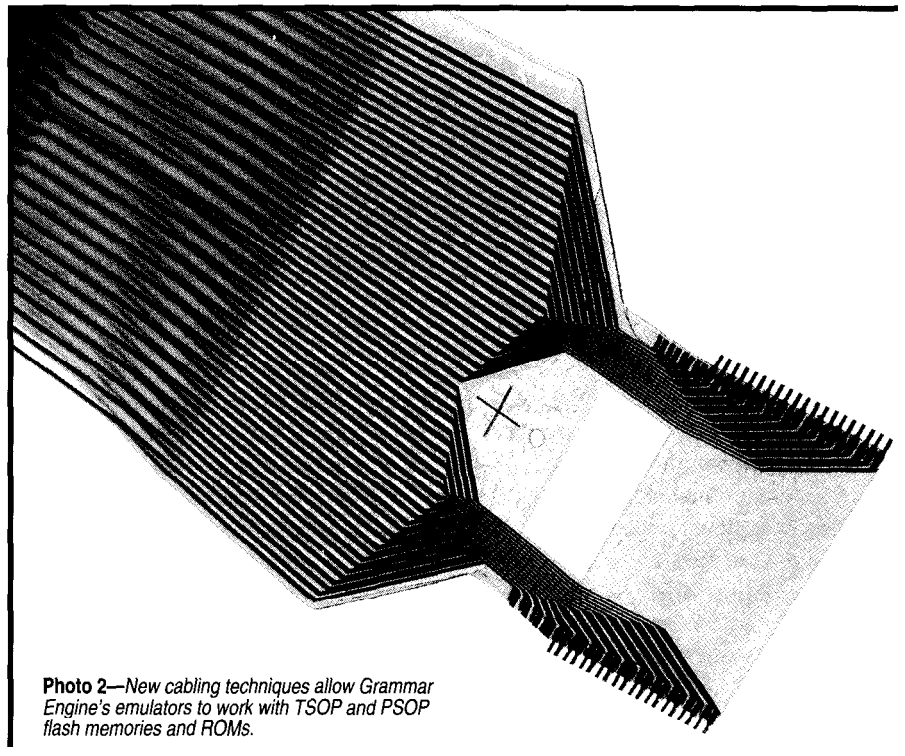


Photo 2—*New cabling techniques allow Grammar Engine's emulators to work with TSOP and PSOP flash memories and ROMs.*

user then accomplishes three different things:

- they can verify programming algorithms on the target system. By monitoring the host interface, errors in the target system software relating to erasing or programming the flash can be discovered
- they can log flash activity. FlashICE logs all flash access (other than reads) on the host system, so the user can monitor the extent of flash activity and its use by the target system
- they can force error conditions. The user tests the target software's error handling by specifying certain failures to be simulated

Finally, with the flash interface extended by one command, FlashICE implements a virtual serial channel between the host and the target system.

This channel enables the host-based debugger to debug target software via ROM monitors. It may be used for other purposes as well.

CONNECTION TO TARGET SY STEM

Real problems arise if the flash part is soldered on to the board. Unfortunately, they don't necessarily go away if there is a socket.

Even though you can buy flash devices in the standard DIP format, it's generally available only in fine-pitch surface-mount parts. It is intended to be soldered on.

Sockets are available for just about any footprint. That makes it a little bit easier to attach a device like FlashICE to the target system. Very often, however, the socket is delicate and attachment to it requires custom cabling.

FlashICE currently supports cables for connection to TSOP and PSOP parts. Photo 2 shows a TSOP cable.

SYSTEMDEBUGHEADER

This proposal is brand new, but the concept is fairly simple. Essentially, a fine-pitch header occupies a small amount of space on the target board.

The system debug header directly connects to external tools, such as ROM emulators, for debugging and developing software. This header is not specific to a given system or device. It emulates simple ROMs or flash memory or monitors target activity without emulating anything.

The driving need for defining such a connector is to provide a universal means of connecting to an arbitrary

target for various development purposes. Such a header can be built into any target regardless of what else is onboard. It can be used during development or even for servicing the embedded system in production.

Those familiar with JTAG realize that it is limited in its capabilities to access the processor or other onboard devices supporting the JTAG interface. However, this header is much more useful and allows simpler and generalized access to the target system.

Short of something like the system debug header, there's no method of developing for systems that have fully soldered-on components. As time goes by, some generalized and universal means must be established to provide effective but simple access to the target system.

Arvind Rana founded Grammar Engine, a company that has pioneered the art of ROM emulation and debugging in the ROM socket. He previously worked for CompuServe and AT&T Bell Labs. You may reach Arvind at arv@gei.com.

REFERENCES

Advanced Micro Devices, ***Flash Memory Products Data Book/ Handbook,*** BAN80M-11/95-0 11796D, 1996.

Advanced Micro Devices, ***Introduction to JTAG and Five-Volt Programming with MACH 3 and 4 Devices,*** Application Note CP-15.3M-3, 1996.

Atmel Corp., *ATMEL* ***Nonvolatile Memory Data Book, 1995-1996.***

Intel Corp., ***Flash Memory: Volume I, 1995.***

SOURCES

FlashICE, System Debug Header
Grammar Engine, Inc.
921 Eastwind Dr., Ste. 122
Westerville, OH 4308 1
(614) 899-7878
Fax: (614) 899-7888

I R S

410 Very Useful
411 Moderately Useful
412 Not Useful

#123

FEATURE ARTICLE

Bob Meister

LSI-11 Simulator on a Personal Computer

This simulator faithfully executes the PDP instruction set and DEC's RT-11 single-user OS. It supports floppy and hard disks, line printers, clocks, console input and output.... It even runs at the same speed.

Back in the early '70s, when minicomputers were fairly new, I cut my teeth on a DEC PDP-8 minicomputer. It ran at only 1.2 µs per cycle with 8 192 12-bit words of magnetic core memory.

It used a Teletype ASR-33 as the keyboard, printer, and paper-tape reader and punch. It had a front panel with switches and lots of lights that burned out. If you were lucky, it had some 8" floppy disks that held 256 KB of data apiece.

In 1978, the Heathkit version of DEC's LSI-11 single-board minicomputer cost about $1300 and consisted of a card cage, CPU, power supply, and enclosure. For this price, you didn't even get a serial interface, but the CPU board came with 8 192 8-bit bytes of dynamic memory.

I bought a serial interface for $100 and connected it to a Teletype ASR-33. Another $100 purchased an additional 8 KB of memory. Two years later, I bought a dual-drive 8" floppy-disk subsystem with a 500-KB total capacity for the (now) unbelievable price of $1900.

Although those good old days are gone, I wasn't altogether ready to give them up. I loved writing code on my LSI- 11, primarily in assembler language. So, I decided to write a program to run PDP- 11 binary code directly on a PC.

I was determined that the program look and act like an LSI-11 system. And, I succeeded. My LSI-11 program runs on an 80x86 and executes PDP-11 instructions.

Granted, since it's interpreting binary code, it runs at a slower rate, but there's not much that can be done about that. The simulated program's speed depends on the number of instructions required to execute it on a target machine.

While the program and the instructions executed are particular to the LSI- 11, the principles in this article are common to many simulators and can be transferred to other target processors.

PDP-11 MINICOMPUTER FAMILY

The PDP-11 was a 16-bit extension to the 12-bit PDP-8 computers. Those 16 bits gave the PDP-11 an addressing range of 65,536 bytes and increased the number of registers from the PDP-8's single accumulator to eight general-purpose registers.

There are over 70 instructions, most of which deal equally well with 8-bit bytes or 16-bit words. All instructions access data in memory or registers. It makes no difference to the CPU.

The effect of most data manipulations is stored in a Processor Status Word (PSW) that has flag bits for carry, overflow, negative, and zero. This PSW exists at the highest address in the machine, 177776 octal.

The I/O devices use memory-mapped I/O, rather than separate I/O instructions or an I/O bus. The PDP-11 reserved address space for the device registers, which were handled like any other memory address by reading or writing their contents.

The PDP- 1 l's memory layout reserves locations in low memory for interrupt and trap vectors (O-477 octal). Available main memory usually goes up to 56 KB.

The last 8192 bytes are the I/O page, which hosts specific hardware and the CPU's internal registers. The PDP-1 l's interconnecting bus provided 16 bits for data, several control lines, 4

interrupt request lines, and 18 bits for addressing up to 256 KB of memory.

Memory-management hardware extended a program's memory-access range to several banks of 64 KB each. With more than 64 KB of memory, the I/O page moves to the highest 8-KB area from 248 to 256 KB.

The LSI-11s are single-board versions of the **PDP-11s**. Besides not having the PDP-11's front panel, the LSI-11 had 8 KB of onboard memory and a microcode routine to examine and change registers and memory locations. When the CPU executes a Halt instruction, it enters this microcode, commonly referred to as the Octal Debugging Technique (ODT) control routine.

Four more addressing lines in the interconnection bus enable peripherals and some LSI-11 models to access up to 4 MB of memory. The newer interconnecting bus multiplexes the address and data lines to save space. Unfortunately, this feature slows the bus up somewhat.

The original LSI-11 CPU only recognized one interrupt-request line. Later models returned to four. The PSW became an internal register, and new instructions were developed to access it. (Some later LSI-11 models also accessed the PSW located at 177776 octal.)

PDP-11's had a unique interrupt system. Depending on the processor, one or four interrupt-request lines were available. An interrupt-request signal passes through each interface board in the backplane to the CPU. Boards physically closer to the CPU have a higher priority.

When the CPU acknowledges an interrupt, the board requesting it prevents the acknowledgement signal from reaching any lower board trying to interrupt. When the higher-priority board finishes, the lower-priority board's interrupt request is processed.

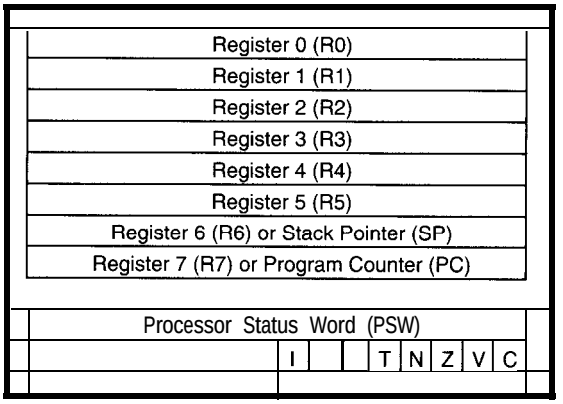
Direct Memory Access (DMA) is also provided. Unlike the PC's onboard DMA, each interface board has its own DMA hardware to transfer data between the peripheral and memory.

The CPU arbitrates DMA requests and relinquishes bus control between instructions. Although data is normally transferred to and from main memory, a memory-mapped I/O device can be the target of such a transfer, assuming its address space is large enough.

PDP-11 ARCHITECTURE

As Figure 1 shows, the PDP-11 has eight general-purpose registers. R7 is the Program Counter (PC), which is often used by instructions, but not usually modified directly.

The R6 Stack Pointer (SP) may be used as a general register but subroutines and interrupts also use it and the memory it points to. The other registers can be used with any instruction and addressing mode. Registers aren't tied to certain instructions.



igure 1-General-purpose *registers* and *PSW show the programner's* model of *the basic PDP-11* and *LSI-11CPUs. On some* models, the *PSW is also accessed as a word at address 177776.*

The PSW holds the condition code flags and an interrupt-enable bit. The larger PDP-11 and LSI-11 models have additional bits within the PSW for multiuser environments.

PDP-11s operate with the octal numbering system. One three-bit digit represents a register, while another digit defines the addressing mode. The eight addressing modes combine with any of the eight registers. Some of the useful modes are: register direct, register indirect, indirect with autoincrement, indirect with autodecrement, indexed, and indexed indirect.

For the autoincrement and autodecrement modes, the register is postincremented or predecremented by one for byte instructions or by two for word instructions, which are always aligned on even addresses.

The instruction set is fairly robust. Almost all data-handling instructions deal with byte data as easily as word data. Registers and memory (including memory-mapped hardware) are treated equally. Data can be moved between any source and destination.

When used with the PC, a few of the addressing modes become the more familiar immediate, absolute, and PC-relative modes. Through this technique, you can write code that runs independent of the address it is loaded into since all data references are relative to the current value of the program counter.

PROGRAM IMPLEMENTATION

I wanted to simulate the LSI-11/03 and have it run programs under the RT-11 OS. It had to execute each LSI-11 instruction like the real thing, which meant having a 64-KB memory space available. The C Small model (64 KB for code, 64 KB for data) seemed perfect for this solution.

The LSI-11 simulator begins with command-line argument processing and some once-only initialization code. The command line specifies which DOS files and/or floppy drives are seen as logical units by the RT-11 OS.

I wrote a device driver called PF (Personal-computer Floppy) that handles eight logical units or devices. Any device that can contain a directory and be randomly accessed can have a bootstrap put on it.

If floppy disks are used, the simulator allocates a one-track buffer in memory so that it won't span the PC's 64-KB address boundary. This buffer is needed to read more than one sector per disk request.

The simulator also clears the general-purpose registers and reads the bootstrap block (block 0) from the first device or file specified on the command line into location 0 of simulated memory. Execution begins at location 0 where the simulator finds the first instruction.

This procedure is comparable to the PC's bootstrap that loads the first disk sector into address 0000:7C00. On

some LSI-11s, a bootstrap ROM prompts for a device's standard DEC two-letter name and logical-unit number. My simulator provides this selection functionality by always booting the device or file associated with PFO.

For the instruction decoder, I initially used an array of 65,536 byte values indexed directly by the instruction word. A value from 1 to 17 selects the major instruction group. Since the same functionality was available by using only 13 of the 16 instruction word bits, I now use an 8 192-byte array. I generate an index into the array by shifting the instruction word right by 3 bits.

The various decoded cases perform requested operations by turning off specific PSW bits, obtaining source and/or destination addresses, fetching source and/or destination data, performing operations, setting specific PSW bits, and storing the results.

While branch instructions don't have operands, they do force a change in the program counter if a specific combination of PSW bits exists. Obviously, certain instructions are simpler to implement.

Most instructions were easy to code. Difficulty lay in managing the appropriate PSW bits based on the result of executing the instruction. Performing a mathematical operation in C or any high-level language produces a zero, nonzero, positive, or negative result. Determining if the result should set the carry or overflow bits requires more work since C doesn't know about these outcomes and the 80x86's flags can't be obtained on an instruction-by-instruction basis.

I test the input data for conditions that cause a carry or overflow and then set the appropriate bits. These additional statements require longer execution time.

I also enhanced the branch-instruction logic. I used the four condition code bits of the PSW as an index into a 16-word array. I convert the branch instruction into one of 16 bits and test this bit in the array's word. If set, a branch occurs, and the PC adjusts accordingly. This streamlines the logic considerably, making all branch instructions operate at the same speed.

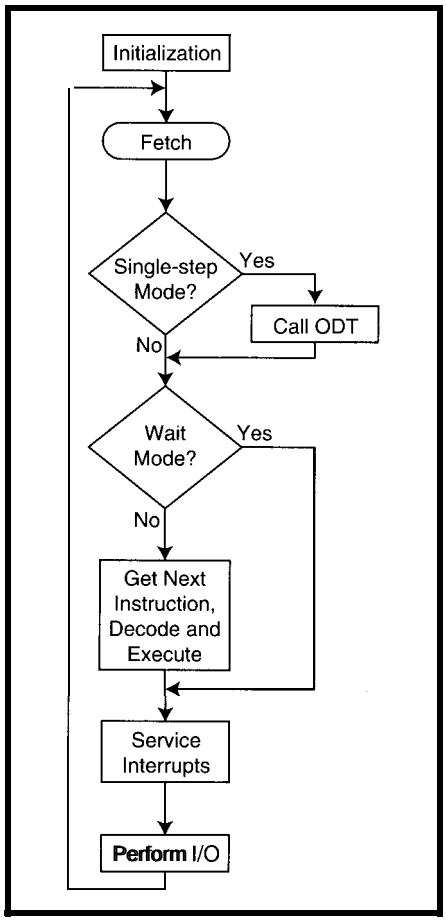


Figure 2—*This* flowchart depicts *the main* simulator *process.* Additional routines *(not* shown) handle addressingmodedecoding, *terminal keyboard and screen* emulation, *and file and disk I/O.*

Like all computers, instructions execute sequentially. At the end of each instruction, several tests handle single-step mode, deal with pending interrupts, check for operator console intervention, process outstanding I/O operations, and maintain housekeeping duties normally performed by a CPU. Figure 2 shows the simulator's process.

DEALING WITH "HARDWARE"

The I/O device-register code was the most difficult to get working. Remember the address space of the PDP-11's I/O page is typically the upper 8 KB.

In the simulator, the main memory goes up to 60 KB, leaving only 4 KB for I/O (see Table 1). This arrangement gives programs more breathing room and is safe as long as I/O devices are limited to the upper 4 KB.

A subroutine in the simulator decodes the source or destination addressing mode and register bits and sets a pointer to the operand's address. When the address is determined, it is compared to the highest memory address to check for known I/O addresses.

If the operand accesses a legal device or memory address, everything proceeds normally. If the program attempts to access an address where no device or memory is present, the simulator performs a TRAP indicating nonexistent memory.

A TRAP or interrupt pushes the PSW and PC onto the stack, and a new PC and PSW are obtained from predetermined vector addresses. All interrupt vectors contain these two words to be put into the PC and PSW. An RTI instruction pulls the stacked PC and PSW back where they belong.

Assuming a program wants to access a hardware-device register and has the correct address, the simulator points to the pseudoregisters of the hardware devices. These bytes are modified by the running program.

Periodically, these pseudoregisters are tested for certain conditions, such as data output to the screen. If required, an I/O operation using the host PC's real hardware (e.g., keyboard, screen, line printer, and disk and file I/O) is performed.

For the keyboard, a PC BIOS function checks for keystrokes. If so, a bit is set in the pseudoregister that a running LSI-11 program can test.

In addition, most hardware devices interrupt when a significant event occurs (e.g., keyboard input or disk operation completion). If a device register has interrupts enabled, the simulator acknowledges the interrupt by saving the present PSW and PC and picking up new ones at predetermined addresses.

In a real system, pending interrupts are recognized and serviced if the CPU's priority allows at the completion of each instruction. The simulator checks specific flags set when an I/O operation generates an interrupt.

The appropriate interrupt vector is accessed and instructions continue at the new location specified. If no devices request interrupts, the necessary I/O operations get done.

By processing interrupts and I/O requests in this order, I introduce a few instructions of interrupt latency. With this delay, the simulator more closely represents a real LSI-11, since few devices can immediately execute, interrupt, and complete tasks.

Since the simulator can't deal with parallel events, I test flags every 100 instructions. If an I/O event starts that interrupts on completion, I test every three instructions. This action shortens interrupt latency so the interrupt occurs closer to the generating event.

Real LSI-11 hardware interrupts immediately on event completion. For a time-critical application, 80x86 interrupt service routines could set bits in a control word, which the simulator could check and respond to.

Some device registers contain read-only bits. Writing zeros to them results in a certain bit combination in the real hardware. The simulator had to test for these conditions and perform mask and bit-set operations to maintain specific bits in the same condition the real hardware would.

Octal Address Range	Typical LSI-11 System Usage
170000-177777	I/O page (device registers)
160000-167776	RT-11 monitor and device drivers
001000-157776	Runnable program space
000500-000777	Program stack space
000000-000477	Interrupt vectors

Table 1—*The 60 KB of simulated RAM decodes LSI-11 addresses of 000000–167776. Standard I/O device addresses are decoded separately and do not actually exist in the simulated memory space.*

Although this code only occurs every 100 instructions, it's extra code which slows the simulator down.

DISK AND DEVICE INTERFACING

I used the PC's BIOS routines for the LSI-11 video, keyboard, and printer. I added support for a Hazeltine model 1510 VDT. This model translates the escape sequences into calls to the video BIOS to clear the screen, position the cursor, and so on. It also translates certain keycodes into the multiple-character sequence a real terminal's keyboard sends.

Real LSI-11 computer systems contain a 60-Hz signal from the incoming power line, or LTC (Line Time Clock). Depending on the system, the hardware generates a real interrupt, or an LTC signal is fed into a separate event line on the CPU board, which causes an interrupt through the normal vector.

The RT-11 OS supports an LTC. Using `ctime()`, I read the host system's date and time. I tracked the number of seconds between successive calls. When there was a one-second difference, I multiplied the number by 60 (60 ticks per second) and forced interrupts through the proper vector until the tick counter reached zero.

The clock kept perfect time since it received 60 counts each second, even though all 60 occurred in a tight loop at one time each second. It was perfect for system clock timekeeping.

Of course, one program wanted to delay a message onscreen for a number of seconds. The program counted the

number of instructions the CPU could execute in a loop between two successive LTC interrupts after skipping one for synchronization. It then multiplied this count by 60 to calibrate a delay loop of 1 s.

On a real LSI-11 system, where the same 1/60s exists between LTC interrupts, this worked fine. On the simulator, the program measured the time between the last pseudointerrupt 1 s ago and the one presently happening. It assumed this large number of counts equaled the amount of time for 1/60s, and multiplied it by 60. The result was a 60-s delay for every second required by the program. Oops!

I first tried the ROM BIOS interrupt 15h, function 83h—a function that sets up a timer and interrupt service routine that works in conjunction with the CMOS RTC available on PC/AT clones. The clock accepts a value in microseconds, but interrupts at 1024 times per second, rounding the given value to within one tick. Converting these 1024 ticks into 60 ticks yields exactly 17 1/15 of the PC's interrupts per 1/60s.

My interrupt service routine resets the timer and increments a counter every 17 1/15 ticks (see Listing 1). In the simulator's main code, I generate a pseudointerrupt for the LTC whenever this counter is greater than zero.

This worked fine until I ran the simulator in the DOS box under Windows. The clock then froze, probably because of Windows' time-slicing and feeble attempt at multitasking. However, the benefit of real LTC interrupts far outweighs the need for running in Windows. Real interrupts also allow the simulator to run faster. So, I didn't worry about this problem.

I simulated a printer interface by using the PC's BIOS printer code to output a character and obtain the status. Real LSI-11 systems only have an error bit to indicate the condition of the printer. I manipulated the various PC bits to create the desired effect.

Although the current implementation prints, it's very slow. The printer driver checks to see if the printer will accept data. If it won't, it sets the interrupt enable bit and returns to the calling program. After 100 instructions, the interrupt condition is detected and serviced. A lot of real time passes between data transmissions, leaving a lot of room for improvement.

Listing 1--The *LTC* interrupt service *routine uses the PC's* 1024-p *timer to generate the 60 ticks per second that are provided by a real LSI-11* power supply.

```
void interrupt int_70()          /* process 976-µs timer interrupts */
{
  long far * lfp:

  lfp = MK_FP(0x40, 0x9C);       /* where count interval is */
  *lfp = 1000L;                  /* reset counter for more time */
  if  (++jiffies >= 17){         /* done enough for 16.667 ms? */
    jiffies = 0;                 /* reset */
    ltccnt += 1;                 /* count a tick */
    if ((++parts % 15) == 0){
      jiffies -= 1;              /* add one jiffy every 15 ticks */
      parts = 0;
    }                            /* end if */
  }                              /* end if */
  _chain_intr(int70a);           /* do the original code now */
}                                /* end int_70() */
```

The disk interface was originally implemented with 1.2-MB floppies. I performed single-sector reads from and writes to the floppy. This was slow, but I coded it without problems.

I preset some of the disk's characteristics, so only a drive letter had to be specified on the command line. Eventually, I expanded the disk interface to allow any DOS or UNIX file containing an image of an RT-11 system disk to be read or written.

I also enlarged the disk buffer to deal with an entire track at a time, rather than a single sector. Presently, the disk interface works with any floppy disk that has 80 tracks, 2 heads, and a user-specifiable number of sectors per track.

I/O operations are performed by the PC's BIOS and DOS services before the simulator processes the next instruction. Real hardware interrupts are handled with counters and various condition tests.

PCs and DOS don't generally use interrupts. A normal DOS program waits for I/O to finish before going on. Other than the line-time interrupts, all I/O takes place instantaneously as far as a simulated program is concerned.

VERIFICATION AND TESTING

Some testing with a real LSI-11 was necessary to learn which bits changed under certain conditions. For debugging, I wrote and compiled small programs (fewer than 10 instructions) on my LSI-11 and then attempted to debug them on the simulator.

After several runs, I knew I needed a better way, so I added an ODT routine to the simulator. I could then code some machine-language instructions into memory, execute the code, store the results, and examine the various registers.

To make a device driver deal with the floppy disk "hardware," I invented some memory locations and register configurations based on common standards, picked the functionality I desired, and modified an existing device driver to put RT-11's parameters into the simulated device registers.

I integrated this into the existing LSI-11 program's floppy-disk routine. I placed the real calculating burden on the PC where instructions aren't simulated.

I copied a set of DEC's LSI-11 paper-tape diagnostic programs onto the RT-11 floppy disks and wrote a program loader, so I could run them without a paper-tape reader. This loader was modified to read a program from a known place on the disk.

To get the RT-11 OS disk to work, I chose another diagnostic-one that tested traps and other oddities of the LSI-11 processor. However, it took a third diagnostic, one that dealt with interrupts, before it actually booted.

Once I had a functional operating system, I tried all the programs I could find. One program scanned the entire

64-KB address space, printing the existent and nonexistent addresses. With this memory map of main memory, I could track device registers that responded when read.

Various programs verified the system and processor, and printing was normal, albeit slow. The LTC kept perfect time, even during operations that would have disabled interrupts for a significant amount of processor time.

In other words, everything that ran on a real LSI-11 computer ran on my simulator. I tested the simulator on a '386-40, a '486-33, and a '486-66. Each ran the program and operating system identically, except for the respective speed differences.

Since the simulator initially processed instructions by sequencing through an optimized list of opcode patterns, instructions found first were executed fastest.

I spent a lot of time counting instructions and optimizing instruction order. All of this optimization was unnecessary once I used a table lookup for instruction decoding.

FUTURE ENHANCEMENTS

I plan to implement a serial port to allow file transfers via Kermit or its equivalent. To keep up with the PC's speed and not miss characters, I need a PC interrupt routine for the serial port and other I/O devices.

The ODT routine should be able to modify the hardware's simulated registers, but small programs in main memory can do the same thing.

The ability to reboot the computer from within the ODT would be convenient. Right now, the user can exit the simulator and press F3 or retype the entire command line to reboot.

A more useful terminal emulation would be one of DEC's VT- 100 series. These are similar to ANSI, although the state machine handling the escape sequences is more complex than the Hazeltine 1510.

I'd like to access an MS-DOS file from within RT- 11 and vice versa. I've already written a DOS program that reads files from an RT- 11 volume.

I have another RT-11 program that knows how to deal with a DOS floppy disk, but there should be an easier way.

When you're designing the environment, you have the luxury of being able to create new, custom operations.

Give it a whirl. It's an experience worth having.

I'd like to thank my officemate, Chris Hilton, for pointing out the timer routines, helping to sort out the interrupt mechanism, and for the use of his systems as an additional test of the simulator's operation.

Bob Meister is a programmer/analyst at the National Association for Securities Dealers. He's been fixing and programming computers for over 20 years and has written simulators for a DEC PDP-8 minicomputer and a Motorola M6800 microprocessor. Bob may be reached at kcbooboo@aol.com.

IRS

413 Very Useful
414 Moderately Useful
415 Not Useful

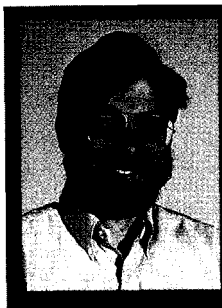
DEPARTMENTS

FIRMWARE FURNACE

Ed Nisley

80x86 Performance

Touring the CPU Spectrum



Last month, Ed showed how all the CPU horsepower in the world won't help a program that doesn't do much computing. This time, he burns CPU while doing hardly any I/O at all. Finally, he gets results more in line with the marketing hype.

Once upon a time, when I was the departmental New Kid, I worked down the hall from a guy whose job it was to determine the performance of not-yet-built mainframe CPUs.

The architecture wizards gave Dan'1 the plan du jour. He wrote yard upon yard of FORTRAN code, and ideally, the results determined what we underlings would design and build.

His programs read instruction-trace tapes from existing mainframes, translated opcodes into their new equivalents, recomputed the timings, simulated superscalar dispatching (even though that term hadn't been coined yet), figured the effect of new cache-management strategies, and combined everything into an overall CPU-performance rating.

And, Dan'1 got it right, too, within a surprisingly small margin of error.

High-end microcomputers incorporate many of the same design features found in those late- 1970s mainframes. That's not surprising, as a single IC now has roughly the same number of transistors as an entire mainframe CPU.

With fortunes made or lost on how one implementation stacks up against another, performance prediction is a critical part of the business plan.

We, on the other hand, have a simpler project-given a specific program, determine how well it runs on a variety of systems. Benchmarking can become as complex as performance prediction, but for just one program, we can observe what happens and find reasonable explanations in the hardware.

Last month, we saw that all the CPU horsepower in the world won't help a program that doesn't do much computing. An 80-MHz '486DX2 was only three times faster than a 16-MHz V40! When your code is I/O bound, don't spend money on exotic CPU hardware-improve the I/O channel first.

This month, we run a program that burns as much CPU time as possible, while doing hardly any I/O at all. The performance range of those same CPUs on this task approaches 50:1, more in line with the marketing hype.

LET THE GAME BEGIN

Selecting an appropriate program for this column was easy. It should run on any 'x86 CPU, perform a vast number of calculations on data from memory, be easy to program, and produce interesting results. I rummaged around in my program heap and came up with Conway's classic Game of Life.

The Game uses a rectangular array of cells. In principle, you update all cells simultaneously based on their current values.

In practice, you use two arrays: the previous and current generations. The algorithm reads the previous generation, computes the current generation, displays the results, and then swaps the arrays.

Once again, I used three Megatel boards and my '486DX2 desktop test system. All four support VGA displays, making video mode 13 (hex) a natural choice.

In that mode, the screen shows 200 rows of 320 columns each, for a total of 64,000 (decimal, exactly) pixels with a palette of 256 colors.

Unlike most VGA modes, each pixel corresponds to a single video-buffer byte, and the program accesses the whole array as a single, linear chunk of memory.

Each cell in the array is either dead or alive. A dead cell is simply a byte holding a binary zero. A live cell holds any value between 0x20 and 0x37.

Those values correspond to a pleasant rainbow sequence in the default BIOS-mode 13-VGA palette. You can use the standard cartoon colors (0x00 through 0x0F), but the rainbow is far easier on the eyes.

Listing 1 shows the nested loops that calculate each Game of Life generation. For each cell in the array, you simply add the number of live neighbors.

If the cell was dead in the previous generation and had exactly three neighbors, it will come to life in the current generation.

If it was alive and had fewer than two or more than three neighbors, it will die of loneliness or overcrowding. Otherwise, it remains unchanged.

The program increments the color number for each generation, wrapping from 0x38 back to 0x20 as needed. Newly born cells contain the current color number, making it easy to spot their birth. Cells remaining alive do not change colors, producing a static background.

The calculation loops exclude a one-cell border that holds only dead cells. More complex versions of the Game use toroidal arrays that connect the top border to the bottom and the left to the right.

You can also fill the border with always-live cells to see how that affects the results.

While Life may not be as complex as fractal geometry, it's certainly easier to explain.

One pass through the inner loop in Listing 1 requires about 100 instructions, so computing a single generation burns slightly more than 3 18 x 198 x 100 = 6.3 million instructions.

Displaying the results involves nothing more than copying the output array directly into the video buffer using the C library memc py ()function.

Tallying the neighbors of each cell accounts for most of the instructions. Listing 2 shows the compiler output for the first two clauses of the eight-way addition statement in Listing 1.

I left all the compiler optimizations and CPU selections turned off because

Listing I--The inner loop of *this simple-minded (but cache-friendly!) Game of Life implementation runs at 1.0 MIPS on a 16-MHz V40 and 46 MIPS on an 80-MHz '486DX2. The nested loops fit info about 340 bytes. Process memory addresses in ascending order and reuse half of the cells in each iferafion.*

```
outp(PortBase,0x02);
for (Row=1; Row < (NUMROWS-1); Row++) {
   outp(PortBase,0x04);
   pPrevDot = pPrevious + (Row * NUMCOLS) + 1;
   for (Col=1; Col < (NUMCOLS-1); Col++){
      Neighbors =
          (0 != *(pPrevDot - NUMCOLS - 1)) +
          (0 != *(pPrevDot - NUMCOLS    )) +
          (0 != *(pPrevDot - NUMCOLS + 1)) +
          (0 != *(pPrevDot - 1)) +
          (0 != *(pPrevDot + 1)) +
          (0 != *(pPrevDot + NUMCOLS - 1)) +
          (0 != *(pPrevDot + NUMCOLS    )) +
          (0 != *(pPrevDot + NUMCOLS + 1));
      if (0 == *pPrevDot){
          if (Neighbors == 3) {
             *(pCurrent + (Row * NUMCOLS) + Col) = ColorNumber;
          }
      }
      else
          if ((Neighbors < 2)|| (Neighbors > 3)){
             *(pCurrent + (Row * NUMCOLS) + Col) = 0;
          }
      }
      pPrevDot++;
```

Listing 2—*The first two clauses in the statement calculating the number of neighbors for each cell generate this assembler code. A cached '486 CPU averages one instruction every 1.7 clock cycles, quite good for a CISC architecture and close to the maximum rate possible for a scalar processor.*

```
; Neighbors =
; (0 != *(pPrevDot   NUMCOLS   1)) +
    les  bx,dword ptr [bp-10]
    cmp  byte ptr es:[bx-321],0
    je   short @1@926
    mov  ax,1
    jmp  short @1@954
@1@926:
    xor  ax,ax
@1@954:
    push ax
; (0 != *(pPrevDot NUMCOLS)) +
    les  bx,dword ptr [bp-10]
    cmp  byte ptr es:[bx-320],0
    je   short @1@1010
    mov  ax,1
    jmp  short @1@1038
@1@1010:
    xor  ax,ax
@1@1038:
    pop  dx
    add  dx,ax
```

I wanted code that would run on any of the test systems.

If you wrote your own assembler code, you'd do much better, but I suspect most current embedded-PC projects use little hand-tweaked code.

In any event, optimum optimization occurs much further back in the design process, well before you get to assembly language. Down at the instruction level, there's only so much improving you can do.

Many high-powered mathematicians and programmers have studied the Game of Life in the decades since its inception. You can take advantage of highly optimized algorithms that compute and display generations in a fraction of the time required by the naive loop I'm using for this column.

If you want to live Life in the fast lane, drop me a note and I'll aim you at some useful references.

With this in mind, Photo 1 shows the timings for a single generation on an 80-MHz '486DX2. Calculating one generation takes 132 ms or 48 MIPS.

That number compares quite favorably with the results from last month, where the same CPU maxed out at 3 MIPS running the Direct Digital Synthesis loop.

Do the division: a program can run 16 times faster than another on the same CPU!

HITTING THE CACHE

The secret to high-speed program execution is very simple: avoid slow operations. I/O instructions on ISA

systems exact a severe performance penalty, easily outweighing all other factors for a fast CPU.

The Game of Life algorithm does absolutely no I/O until the end of each generation loop, freeing the CPU to do what it does best: compute.

The two nested loops in Listing 1 comprise about 130 assembly-language instructions that fit into the '486's 8-KB internal cache. Even when branches flush the prefetch queue, the CPU still hits the cache with no wait states.

Surprisingly, even though 64,000 cells won't fit into the cache, most of the memory accesses are cache hits. The code in Listings 1 and 2 steps through the array in more-or-less sequential order, which is precisely what the cache controller expects.

Calculating the first cell's neighbors causes three cache misses, one each for the leftmost cells on the three top rows. After the cache controller selects and fills three different cache lines with the first 16 bytes from those three rows, all the remaining accesses are cache hits.

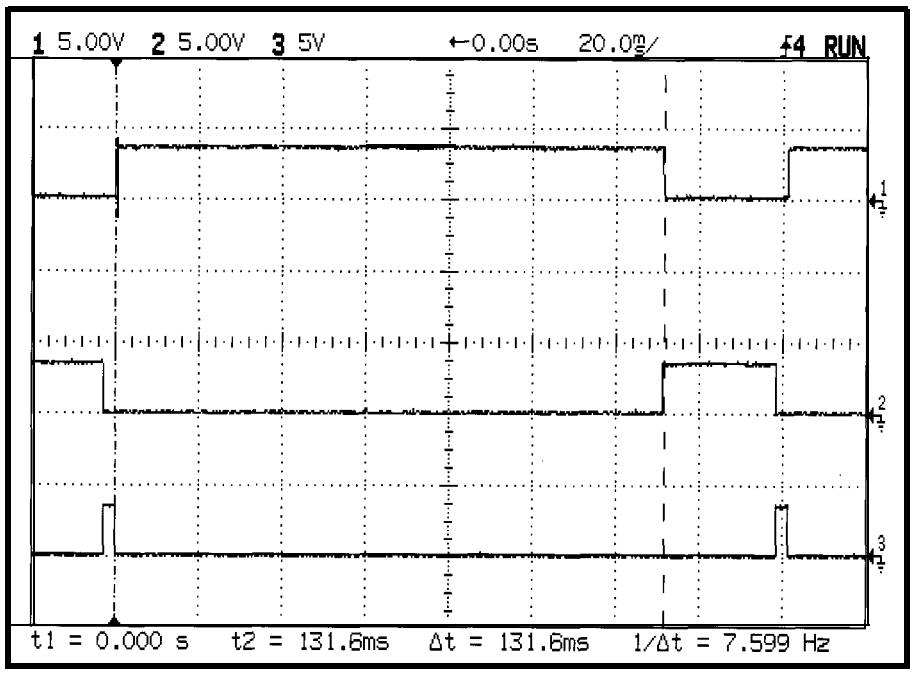


Photo 1—*Computing each Life generation fakes ~130 ms on an 80-MHz '486DX2 with both caches enabled. Copying the resulting 64,000 bytes into the video buffer takes about 10 times longer than the same copy into ordinary system memory because the video card sits on the ISA.*

Even better, for the next 13 cells, every cell-array memory access is a cache hit. A rough estimate says that calculating the neighbors for 3 18 cells in the first row requires 2544 memory reads, only 60 of which miss the cache, That works out to a hit ratio of 97.6%, ignoring all other effects.

CPU	Memory Caching	Clock MHz ext/int	Generation Time, ms	MIPS in Generation Loop	Clock Cycles per Instruction
'386SL	none	25/16	6200	1.0	16
'486SL	none none	25	1300 1100	4.9 5.8	4.9 5.1
'486SL	internal	25	430	15	1.7
'486DX2	none	40/80	1800	3.6	22
'486DX2	internal	40/80	140	46	1.7
'486DX2	external	40/80	750	8.5	9.4
'486DX2	int + ext	40/80	140	46	1.7

Table I--Running a *compute-bound application on the same CPUs as last month produces completely different results!* Notice how *caching affects the* generation time and *thus the effective MIPS and cycles per instruction values.* Computing each Life generation *requires 64,000 iterations of a 100-instruction loop.*

In ***INK 70, we*** saw that a cache miss costs -520 ns, equivalent to 40 single-cycle instructions for a CPU running at 80 MHz. Photo 1 shows that the calculations for one line take -670 µs, so the cache-miss penalty of 30 µs works out to about 4.5%.

Because the entire cell array fits into a 256-KB external cache, the situation this month is even better. Only my '486DX2 test system has an external cache, but the results are interesting.

Table 1 summarizes the measurements for all the systems. The actual time required to calculate each generation depends on the number of active cells, so the "Generation Time" column contains the value for a particular run. I rounded the measurements and calculations to two significant figures in honor of this uncertainty.

With both caches enabled, the '486DX2 rips through a generation in 140 ms. Dividing that into the total instructions gives about 46 MIPS, a respectable, if somewhat artificial figure. Dividing the clock frequency by the MIPS number shows that the CPU requires about 1.7 clock cycles per instruction.

Remember that these CPUs can dispatch and decode only one instruction per cycle, even if their internal pipelines hold many instructions in various stages of execution. An overall rate of 1.7 cycles per instruction means that a CISC CPU like the '486 actually has many single-cycle instructions.

Disabling just the external cache doesn't change the results very much. Even though both cell generation arrays fit into the external cache, the internal cache-hit ratio is so high that speeding up the memory doesn't count.

Disabling both caches has a catastrophic effect on performance. With each memory access slowed to DRAM rates, the CPU drops to 3.6 MIPS and 22 clock cycles per instruction. That's only 8 % of the cached performance!

In round numbers, enabling the external cache doubles the CPU's base performance and enabling the internal cache gives you an order of magnitude. Different benchmarks give different results, with the Game of Life staking out the nearly ideal end of the spectrum.

The Megatel PC/II+*i* board, with a '486SL running at 25 MHz, clocks 15 MIPS with the cache enabled and 3.6 MIPS with it disabled. Interestingly, this CPU requires the same 1.7 cycles per instruction as its clock-doubling kin.

Evidently, for this benchmark, most of the instructions just don't speed up as much as you'd expect. Intel makes a clock-doubling SL processor, but if Megatel used it, I'd expect at least a bullet item in the features list.

The clock-frequency difference seems to account for nearly all the performance difference. Multiplying 15 MIPS by 80/25 shows that the PC/II+*i* would run at about 48 MIPS if the memory system kept up. Converting to 40 MHz-a more realistic number—gives a respectable 24 MIPS.

I don't have details on the Megatel board's memory interface. Because disabling the cache slows the board by only a factor of 2.6 instead of 13, I presume that the memory is relatively faster than that on my DX2 system. In any event, you want the cache on all the time!

COUNTDOWN: 4, 3, 2...

Comparing the Megatel PC/II+*i* and PC/II+ boards highlights the difference between the '486 and '386 CPUs. The boards are, by and large, identical: same clock frequency, same peripherals, same support chips. The chip settings may be different, but I don't have any control over them!

Last month, we saw that the two boards gave nearly identical performance when running an I/O-bound program. That's no surprise, as the ISA Compatibility Barnacles limit these boards just as firmly as desktop boxes.

Given a free run, though, the '486SL runs three times faster than the '386-SL. Intel improved the '486 CPU's hardware and microcode, speeding up memory-address computation and reducing the number of cycles per instruction. If you have a compute-bound problem, the '486 runs your algorithm much faster at given clock frequency.

The '386SL on the PC/II+ board runs at 4.9 MIPS and 5.1 cycles per instruction. That's only 2.2-times faster than the I/O-bound test case, indicating that the CPU and I/O are more closely matched.

In effect, the CPU simply can't run much faster than the ISA I/O lets it. If you are running I/O-bound code, spending significantly less on the CPU might be good economy.

As expected, the V40-based PC/II+*v* has the lowest performance: an even 1 .O MIPS and, thus, 16 cycles per instruction at 16 MHz. Last month it ran at 0.9 MIPS, which shows that the instructions and I/O rates are pretty well matched.

Having compiled all those numbers, I turned to the dots on the VGA....

VIEWING THE VIDEO

The Game of Life produces interesting patterns on the screen, which may account for much of its popularity some years ago. After all, what other graphics program could you run with just a monochrome character display?

Watching the display pointed out another truism: not all memory is created equal.

After computing each generation of cells, I copied them into the VGA buffer and the other generation array. Listing 3 shows the two `_fmemcpy()` functions that do the deed.

The version of Borland C I used defaults to Small memory model, which meant I allocated the two buffers on the far heap. The ordinary `memcpy()` function works on the near heap. Ah, for a flat memory model!

The `outp()` functions produce traces that measure the time required for each memory copy. Although I knew the copies were less expensive than the nested generation loops, it's always a good idea to measure what you're doing.

The bottom two traces in Photo 1 show that copying 64,000 bytes into video memory, while fast when compared to the generation time, takes ten times longer than the same copy into system memory.

When I looked at the C library source code, `_fmemcpy()` boils down to a `REP MOVSW` instruction. Ignoring a few tricks on '486 CPUs, that's as fast as it gets.

Recall that the '486 CPU has a write-through cache that caches memory reads and passes writes directly to the external memory. In this case, `_fmemcpy()` sees a 15/16 read-hit ratio and an agonizingly long pause for every video-buffer write. Counting `REP`

Listing 3—*Although these two memory copy functions look identical, the first is dramatically slower than the second because video memory accesses incur many waif states. The ratio ranges from 1.41 on an uncached 25-MHz '386 board to over 10:1 on an 80-MHz '486DX2 with both caches enabled.*

```
outp(PortBase,0x08);
_fmemcpy(VBUFFER,pCurrent,NUMCELLS);   // update video buffer

outp(PortBase,0x10);
_fmemcpy(pPrevious,pCurrent,NUMCELLS); // set up new dots

outp(PortBase,0x00);
```

MOVSW as a single instruction, the CPU runs at 37 instructions per second.

That's unfair, of course. The **MOVSW** executes **32,000** times in 26.9 ms, running at about **1.2** MIPS. Writing into the video buffer costs nearly as much as accessing the I/O ports!

Photo 2 shows the '386SL timings. Because this board has its VGA controller on the CPU's Local bus, copying to video memory takes only 15.2 ms, rather than 26.9 ms for the desktop ISA VGA controller. If I had a video Local-bus controller on the desktop system, the results would probably come out the way you'd expect.

Copying to '386SL system memory takes four times longer, a slowdown from the more efficient '486DX2 caching. With the '486 caching turned off, though, the '386SL actually runs twice as fast! Of course, nobody would run a '486DX2 without caching, but the raw number is certainly interesting.

Table 2 summarizes the memory performance for all the systems. If your application sloshes huge arrays back and forth, these numbers may be more relevant than Table 1.

More likely, though, you'll wind up with whatever memory interface happens to be attached to the CPU you need for the rest of the job.

Keep these values in mind, though, as they put an upper limit on how fast you can update memory-mapped peripherals.

IN CONCLUSION...

So, there you have it. Two completely different benchmark programs that abuse the systems in two completely different ways.

Based on their results, you can follow the hardware progression from a relatively simple '286 to a relatively complex '486 and see which features provide the most benefit for each program.

The faster CPUs gain their speed, naturally enough, by executing faster instructions at a higher clock frequency. You can extrapolate to find a Pentium's performance: very fast, unless you do a lot of I/O. Then, regardless of how fast the CPU might be, it spends far too much time just waiting around.

The Game of Life uses only integer arithmetic, leaving the floating-point hardware on the '486SL and '486DX2 CPUs completely unused. If your application depends on real-number calculations, the V40 and '386SL CPUs are certainly not the right hammers for the job!

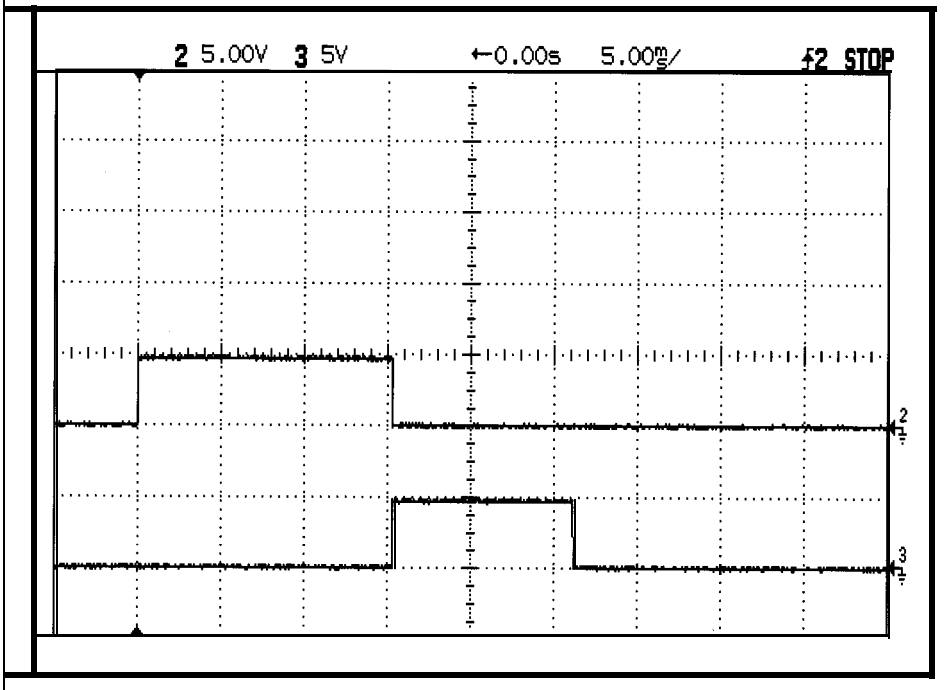


Photo 2-A *25-MHz '386SL* **computes Life generations** in -1800 *ms,* far too long *for the scale of this scope shot. However, copying the results info the Local-bus* video *buffer* requires *only* **40%** *more time than the same* operation *info system RAM.*

CPU	Memory Caching	Clock MHz ext/int	copy to Video Buffer ms	Copy to Video ns per word	copy to Memory ms	copy to Memory ns per word	Video Slowdown Ratio
V40	none	16	117	3700	45.6	1400	2.6
'386SL	none	25	15.2	480	10.8	340	1.4
'486SL	none	25	16.4	510	9.64	300	1.7
'486SL	internal		15.8	490	5.30	170	3.0
'486DX2	none	40/80	31.3	980	19.5	610	1.6
'486DX2	internal	40/80	27.6	860	6.44	200	4.3
'486DX2	external	40/80	31.0	970	5.08	160	6.1
'486DX2	int + ext	40/80	26.9	840	2.61	82	10

Table 2-Somewhat *surprisingly, memory and video accesses run faster on the 25-MHz Megatel embedded-PC boards than on the 80-MHz desktop '486DX2 system. It turns out that the Local-bus video controllers on the Megatel boards outperform an ISA desktop VGA board.*

Conversely, if your application does lots of I/O backed by relatively little calculation, splurging on a '486DX2 won't buy you much beyond a warm feeling that you're covered when the specs change.

In any event, we've certainly pinned down the performance extremes. You'll be hard-pressed to find better or worse performers!

WINNING THE DESIGN CONTEST

One harbinger of the sleet season around here is the arrival of a huge stack of ***INK*** Design Contest Entries. I spread the folders all over the floor, read through their contents, sort them several ways according to the rating criteria, and pick the best of the lot.

All this takes days of reading and shuffling, but the winner usually becomes apparent during the first pass. Every year, it seems, all the judges independently agree that one entrant got everything right. The contest then boils down to ranking the others.

Conversely, another entrant gets nearly everything wrong. The project may be interesting, the work may be technically adept, but the contest entry doesn't stand a chance. Strangely enough, all the judges seem to agree on that one, too. A single photo clipped to a program listing does not a winning contest entry make.

To put this on a personal level, think of me sitting across a table from you with your project between us. Your assignment: tell me about your new widget, as best you can.

The first thing I want to know is what your project does or—better—what problem it solves. You are thoroughly familiar with what you did, but remember, I've never seen it before.

Worse, your contest entry just describes the hardware and program. I can't twiddle the knobs, push the buttons, or watch the screen. If you don't mention something, I'll never discover it.

Some entries arrive without a scrap of explanatory text. All I see is a schematic, perhaps a program listing, and maybe a photograph or two. With no background, I can't tell what the project is, what it does, or why I should care. It might represent the most challenging project in the stack, but if somebody else gives a better explanation, well....

Next, if you're doing something completely off the wall, give me a chalk talk on the technical background. For example, Rick May figured power consumption by multiplying current times voltage in a quad opto-isolator. He included the original design description, sparing me the trouble of figuring out how it worked, and eliminating the risk that I might not bother.

If the schematic isn't trivial, explain what's going on. You can assume I know electronics pretty well, so you needn't explain the obvious. Similarly, describe just the really tricky sections of the software, as I'm not up to speed on every CPU in the world.

Don't be afraid of diagrams showing how things work. Very often, one simple, hand-drawn sketch makes a page of text perfectly obvious.

Conversely, don't bury me in meaninglessly detailed flowcharts of your code. Just make the higher levels clear; the rest follows.

You needn't spice up your narrative with humor, shaggy-dog stories, or fancy prose. Make every word, every schematic, every figure explain something useful. Convince me that you've done something really neat. Save the stories for your article.

Oh, yeah, your article. Guess what? That explanation forms the basis for the ***INK*** article telling everybody else how you took the top prize! If you can explain it to me, you can explain it to them!

Go for it!

RELEASE NOTES

LIFETIME.EXE shouldrunonany stock DOS PC with a VGA-compatible video interface. LIFECD.EXE disables the '486 on-chip cache using, naturally enough, '486-specific instructions that crash other systems. Pick the appropriate one for your situation.

The programs produce blips on the system's highest-numbered LPT port. You can hitch up a scope to track execution times through various sections of code. Press any key (other than the shifts, OK?] to return to DOS.

Next month: tuning up. ❑

***Ed Nisley** (KE4ZNU), **as Nisley Micro Engineering, makes small computers do amazing things. He's also a member of Circuit Cellar INK's engineering staff. You may reach him at** ed.nisley@circellar.com **or 74065.** 1363@compuserve.com.*

SOURCE

PC/II+, **PC/II+*i*, PC/II+*v***
Megatel Computer Corp.
125 Wendell Ave.
Weston, ON
Canada M9N 3K9
(416) 2452953
Fax: (416) 2456505

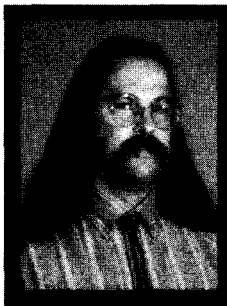
IRS

416 Very Useful
417 Moderately Useful
418 Not Useful

Hard Facts About Soft Ferrites

FROM THE BENCH

Jeff Bachiochi



Fabricated magnetic materials mystify most. Jeff explains the differences between hard and soft ferrite. He closes with an overview of its use in power supplies, EMI and RFI beads, power inductors and transformers, torroids, and baluns.

In the early 1600s, English scientist William Gilbert published his work on the theory of magnetism. For the first time, it was suggested that Earth itself was one giant magnet.

Invisible lines of force leave the north magnetic pole and reenter the earth through the south magnetic pole. Luckily, civilization prospered around the equator and not the poles where the compass was useless (either pointing down in the north or, worse yet, up in the south).

It wasn't until 1820 that a Danish physicist, Hans Oersted, observed a relationship between magnetism and electricity. A compass needle in close proximity to a current-carrying wire is attracted toward the wire.

From this experiment came the "right-hand rule." When the right hand is held in the hitchhiker position with the thumb pointing in the direction current flows through a conductor, the fingers point in the direction of the magnetic field created by the current flow.

Ten years later, Michael Faraday created an electric current by passing a magnet near an electrical conductor. He summarized the phenomenon as:

- moving electrical charges produce magnetic fields
- magnetic fields exert forces on moving charges
- changing magnetic fields cause electrical currents to flow

At some point during our elementary or secondary school careers, most of us have demonstrated these invisible lines of force by sprinkling iron filings on a piece of paper held above a magnetic source.

The mathematical theory describing the disturbances caused by moving magnetic fields and electrical charges was published in 1864 by James Clerk Maxwell. His experiments led to the discovery that the disturbances can be treated as wave motion and that they travel at the speed of light (3×10^8 m/s).

A magnetic field is set up by electric current flowing through a conductor. It gets stronger if either current increases or the conductor is looped so that the magnetic fields produced are cumulative.

This arrangement created the first coil (solenoid) and led to the discovery of electromagnetism. The ability of a magnetic field to strongly attract iron, cobalt, or nickel was named *ferromagnetism*. As you remember from chemistry, *ferro* comes from the Latin for "relating to iron."

ATOMICS

What's so special about these ferromagnetic materials? It begins with the atomic structure of materials.

The negatively charged electrons circling the nucleus of an atom form shells as they whiz around at different energy levels. Each electron can have a mate or not depending on whether an odd or even number of electrons are in the shell.

Electrons having a mate often have opposing spins, and the magnetic effects of the opposing charges cancel. Single electrons can be coaxed into spinning in the same direction by an external magnetic field. When this happens, the material becomes magnetized, and the magnetic field is enhanced.

These "soft" magnets lose most of their magnetism when the external magnetic field is removed because the single electrons are no longer working together. Internal forces enable "hard" magnets to retain their magnetism unless affected by an external stimulus like heat or a hard whack.

Not all magnets are composed entirely of metal. Lightweight ceramics, known as *ferrites*, have many of the same properties.

Let's take a look at these materials and see what makes them so useful.

SOFT CERAMIC

Although magnetite, a naturally formed magnetic mineral, is found in plentiful deposits in Asia, it wasn't until 1945 that soft magnetic materials could be produced in the lab. These materials were produced in the Netherlands for commercial applications by J.L. Snoek of Philip's Research Laboratories.

Ferrites are homogeneous ceramic materials made up of various oxides, primarily iron. Their composition hasn't changed much over the years, but process control has improved their purity.

Ferrite materials are found in almost every electronic product used today. High-electrical resistivity dramatically reduces the eddy current losses associated with metal laminations. In higher-frequency applications, this resistivity is even more important as the losses increase with the square of the frequency.

The production process of ferrites lends itself to special shaping by pressing, extruding, and/or grinding techniques. And, the material is efficient. High magnetic permeability concentrates and reinforces the magnetic field better than any other class of magnetic material from audio to 1 GHz.

The mix of materials used to produce the ceramic shapes varies to achieve different characteristics. These features are pretty much a tradeoff of high permeability for that of higher electrical resistivity. The manganese-zinc ferrites have the highest permeability, while the nickel-zinc ferrites have the highest electrical resistivity.

Most ferrite is produced as single- or multiholed cores. These are toroids and beads. However, ferrite is made in many other shapes such as rods, pot cores, and tuning slugs for variable inductors.

Many of today's products would not be possible without the use of ferrite. Electronic applications include power transformers and chokes, pulse- and wide-band transformers, GFIs, and EM1 and RFI suppression.

Let's look at some of these in more detail.

50/60-HZ POWER TRANSFORMERS

Ferrite-core power transformers look very much like their counterpart, the laminated-core transformer. The high electrical resistivity of ferrite reduces core heating, and the high permeability helps to concentrate magnetic flux, reducing stray magnetic fields. Designs, however, are limited by core saturation or heating due to core or winding losses.

Iron-core laminations are usually produced in the shape of an E-hence the name E-core. The E-shapes are inserted into a bobbin from alternating sides to build up a solid figure-eight core around the bobbin. Ferrite is easily molded into E-shapes. Only two E-shapes are necessary to complete a core, eliminating the need for iron laminations.

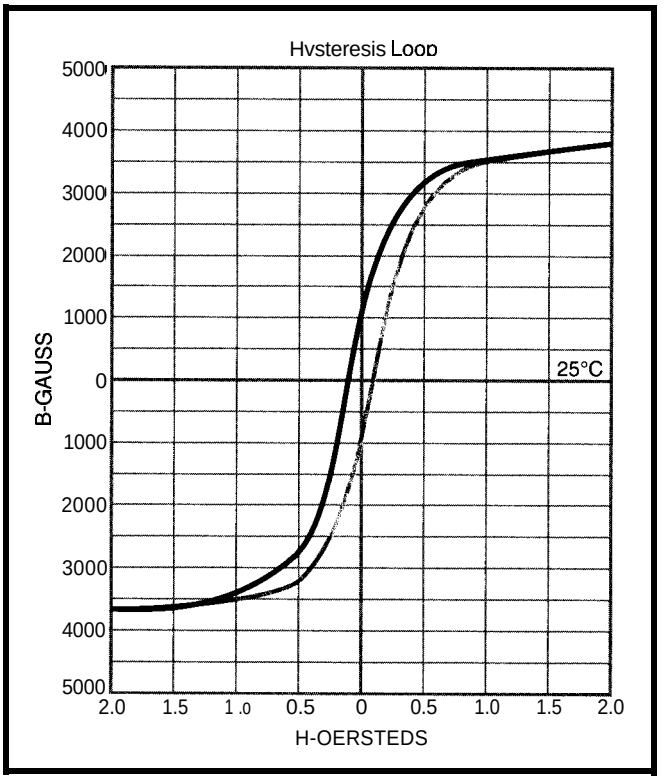


Figure 1--The hysteresis loop *for* this *particular ferrite* material *shows* a tendency to remember *a small amount of magnetism once the* field is reduced to zero.

Inductance for a particular core material is calculated with the formula:

$$L \text{ (in nanohenries (nH))} = N^2 \times A_L$$

where N equals the number of turns of wire, and A_L equals the inductance index listed for each size and composition of ferrite (nH). The inductance increase of a ferrite core over an air core for the most part equals the permeability of the material being used.

The peak magnetic field strength is based on the current passing through the conductor and is measured by:

$$H \text{ (in oersted (Oe))} = 0.4 \times \pi \times N \times \frac{I_p}{l_e}$$

where N equals the number of turns of wire, I_p is the peak current (A), and 1, is the effective path length of the core (cm).

Another important quantity is the peak flux or maximum flux density created in the core material. You calculate this quantity by:

$$B \text{ (in gauss (G))} = E\left(\frac{10^8}{K \times A_e \times N \times f}\right)$$

where ***E*** equals RMS voltage, ***K*** is a constant depending on wave shape (4.44 for sine waves), ***A,*** is the cross-sectional area of the core (cm^2), N stands for the number of turns of wire, and f is the frequency [Hz].

The peak flux density is important because this must remain within the linear part of the B/H hysteresis loop for the material to remain unsaturated.

The graph in Figure 1 shows the relationship between magnetic field strength (H, in Oe) and the magnetic flux density (B, in G). The hysteresis loop is created by the flux-density remanence or flux density remaining once the applied magnetic field strength is reduced to zero.

POWER INDUCTORS

Although transformers may be wound using toroid forms, you're more

likely to find these donut shapes being used for chokes.

For low-power inductors [less than 100 G), the coil's "Q" is determined by the material's permeability (μ) and loss factor (tan δ/μ). This also relates to the wire size (Ω), inductance (L), and frequency (Hz).

Loss factors increase and permeability decreases as frequency goes up. However, they are nominally flat at less than 1 MHz. As power increases, so will the hysteresis, winding, and core losses. But, in general, these are less than those chokes of iron-lamination construction.

If there is a DC component to the circuit current, as in power-supply designs, that DC current creates a bias magnetic field. If this field is significant, it alters the available saturation level for the AC current.

This peak magnetic field strength should be calculated separately. The overall field strength can be reduced by raising the inductance (fewer turns) or by lengthening the magnetic path length (larger diameter toroid). Slotting the toroid (providing an air gap) also lowers the permeability.

CURRENT TRANSFORMERS

The primary coil of current transformers has as few as a single turn. Toroids used as current transformers may require the primary coil to simply pass through the center of the toroid.

Here, the transformer's primary coil is not the load but a series sensor for current. As in a regular transformer, the conductor's current produces a magnetic field (albeit a small one since the number of turns is small), which is collected by the high-permeability core.

The secondary coil's turns ratio is large to create a small voltage proportional to the primary conductor's current. Current transformers are used as GFIs protecting AC outlets in the bathroom and kitchen.

FERRITE BEADS

An economical method of attenuating unwanted high-frequency noise is to create an RF choke by placing a ferrite bead on the conductor. you've probably seen big hunks of ferrite

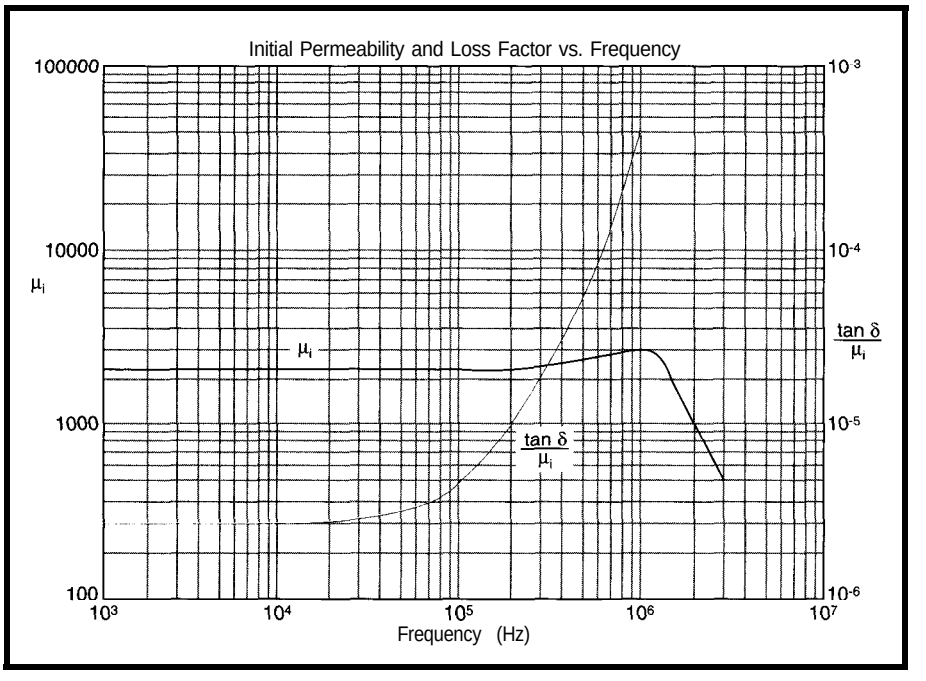


Figure 2—*This* ferrite *material is designed for attenuation of signals above ~1 MHz. Notice the effects of frequency on permeability and loss.*

placed on video cables to reduce EM1 and RFI emissions.

At low frequencies, the conductor (with a ferrite bead) looks like low impedance. But, at higher frequencies, its high impedance chokes off RF.

At low frequencies, the component [conductor and bead) presents a relatively small inductance whose reactance can be neglected.

Permeability remains fairly constant up into the megahertz region, but rises slightly before falling off dramatically (see Figure 2).

The loss factor grows rather linearly over the same range. It is the rising loss which adds series impedance (Ω) to the falling inductive reactance and absorbs the higher unwanted signals.

In selecting the right bead for your application, note where attenuation is necessary and choose the material most effective for that range.

In general the higher-permeability materials begin attenuation at lower frequencies (approximately equal to 1 MHz), while the lower-permeability materials begin attenuating a decade or two higher.

Using the specific charts for the various materials, one can pick off the impedance-per-unit-of-air-core inductance (Ω/H) and substitute this into the formula:

$$Z = K \times L_o$$

where ***K*** equals the impedance-per-unit-of-air core inductance. L_o is determined by:

$$L_o = 0.046 \times N^2 \times \log_{10}\left(\frac{OD}{ID}\right) \times Ht \times 10^{-8} H$$

where N stands for the number of turns (1, in this case), *OD* is the outside diameter of the bead (mm), ***ID*** is the inside diameter of the bead (mm), and ***Ht*** equals the height or length of the bead (mm).

As with all ferrite, the temperature affects each material differently. Refer to individual temperature specifications if the equipment is to be operated at elevated temperatures.

And, as with any transformer, DC bias decreases the effective impedance. This decrease can be as much as 50% with a field strength of only a few oersteds.

Since some beads are placed directly on PCBs, the resistivity of the material is important. High-permeability materials (manganese-zinc) generally have low resistivity (100 Q/cm). The beads act as conductors, and circuit traces must be placed so that the beads are not shorted out.

An alternative solution is to use beads that come processed with an

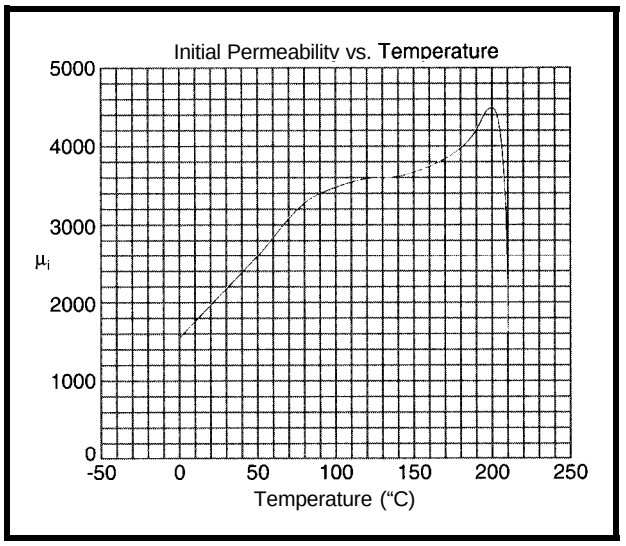


Figure 3—*Permeability increases with temperature up to its* Curie *point. After* this, if falls dramatically.

insulating coat. Nickel-zinc ferrites have much higher resistivity (10^5–10^{12} Ω/cm), but the tradeoff is lower permeability.

Multihole cores increase the impedance via multiple turns through the same core. The number of turns used here is generally low and most effective at lower frequencies, as interwinding capacitance increases with the number of turns.

Baluns are special transformers using a twin-holed bead. They are usually used for impedance matching where two turns (the center tap being grounded) form the unbalanced connection. One winding and centertap form the second balanced connection. The impedance ratio is $1:N^2$ or, in this case, 1:4.

Twin-hole beads are also used as common-mode chokes. Much conducted RFI is common-mode. This is reduced by applying the choke so the signal and return paths on the input side both go on the same core, wound in the same direction.

The output side sees-or, better yet, does not see-the common RFI that cancels. Yet, the important signal slides on through.

ENVIRONMENT

Since ferrites are inert ceramics free of any organic substances, their performance is not degraded by most environments (including heat up to a few hundred degrees Celsius). Above the Curie temperature (which varies in materials from 120°C to 500°C), ferrite loses all magnetic properties.

The effect reverses as the material regains its properties once temperature reduces. In general, permeability increases as the temperature rises, although each material has its own characteristics as shown in Figure 3.

SURFACE-MOUNT COMPONENTS

You've undoubtedly noticed the proliferation of surface-mount products. Most would not be possible without today's SMT ferrite inductors. Most inductor manufacturers have a complete line of SMT parts.

Power products, where heat is of concern, can be shrunk due to the reduced losses within ferrite cores. And, ferrite is much better at containing stray-magnetic fields than laminated-style transformers.

Because manufacturers continue to improve materials and process control, new uses for ferrite will continue into the next century.

Remember core memory? Well, try to forget it. Although ferrite is being used more and more, I doubt we'll see a renaissance in the use of ferrite for improved memory products. ❑

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@circellar.com.*

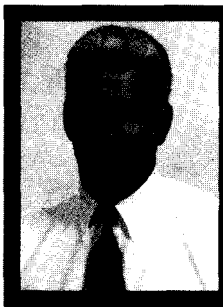
IRS

419 Very Useful
420 Moderately Useful
421 Not Useful

Flight of the Phoenix

SILICON UPDATE

Tom Cantrell



Zilog has risen from its ashes. It offers the S180 processor along with an evaluation/emulation board. Ironically, it's a chip off the old Z80, CTC, PIO, and SIO, the cornerstones of Zilog's glory in the 1970s.

Take a close look at the logo on a Zilog chip. Though admittedly abstract, it depicts a Phoenix, also known to those up on their mythology as the bird that rose from the ashes.

It's rather fitting as far as logos go, given the history of the company. I must admit to a personal interest in this, having fond memories of the early days of microprocessors and personal computing. Indeed, shoved in a closet is my old IMSAI (a '70s-era "PC") with a 4-MHz Z80, 64 KB of RAM, and mighty 8" floppies.

Just for kicks, every now and then, I fire the old bird up. Sure, I have to wait a few minutes for the chips to warm up, but soon I'm flipping switches and scanning the LEDs of that wonderful front panel (wish they still had 'em).

Stop-Reset-Examine-Run is rewarded with a satisfying clanking sound you don't hear anywhere but at a machine shop these days as the 8" floppy pounds into action. Surprisingly quickly, the familiar A> prompt appears. I type in DIR just like now, except it's CP/M instead of MS-DOS.

Yeah, Zilog was flying high in the late '70s, not just on the strength of the Z80, but on a critical triad of peripheral chips-the CTC (counter/timer), SIO (serial I/O), and PIO (parallel I/O).

Each peripheral either matched or exceeded its equivalent from Intel and Motorola. Like the tail wagging the dog, the attraction of the peripherals often tipped a CPU decision in favor of the Z80.

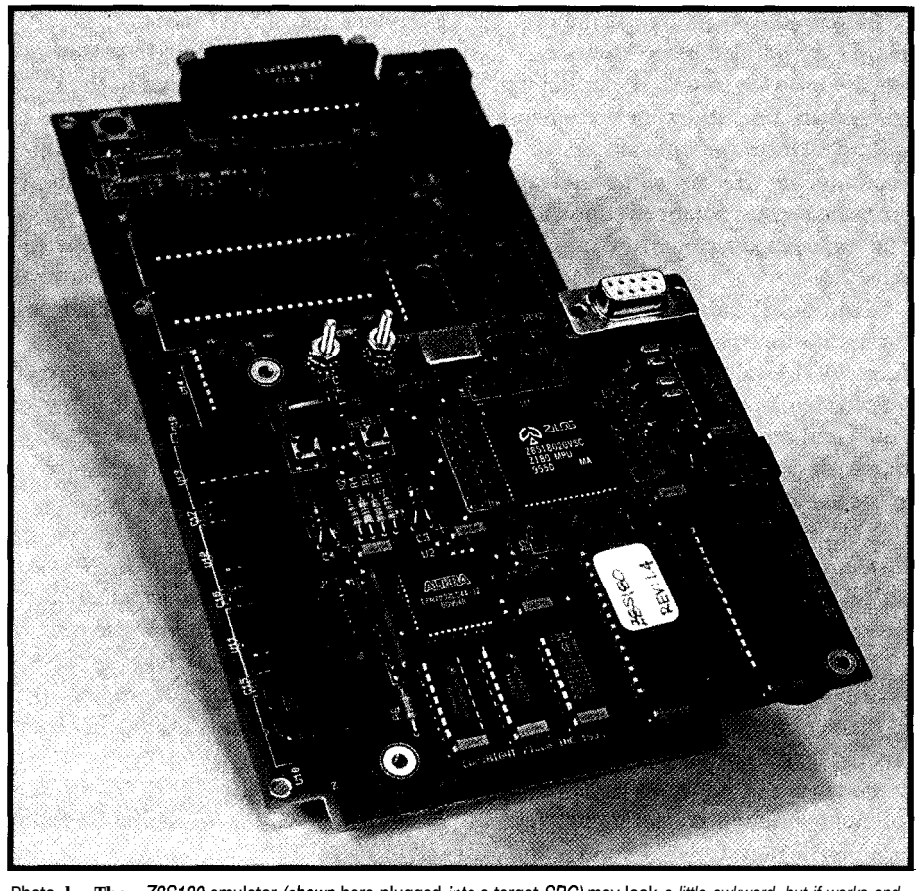


Photo 1--The *Z8S180* emulator *(shown* here plugged *into* a target *SBC)* may look *a little awkward, but if works and the price is right.*

By the end of the '70s, however, the first whiffs of smoke could be detected as Intel, Motorola, and Zilog all geared up for the 16-bit battle between the 8086, 68000, and 28000, respectively. Despite competitive technology, Zilog was no match for the goliaths-it was kind of like being an ant in a boxing ring.

Certainly, once the PC and Mac established themselves, Zilog, having burned the candle at both ends, was little more than toast. While most businesses and semiconductors boomed in the '80s, Zilog stumbled along under the tutelage of Exxon (it didn't make sense then either).

Chips were announced but not delivered, perhaps not surprising given that employee turnover was at revolving-door level. The UNIX workstations were semipromising, until an outfit called Sun came along and spoiled the party. Like an over-the-hill champ getting pounded, it was sad to watch.

Finally, in a classic example of the destruction and creation of capitalism (i.e., the corporate equivalent of a call to Dr. Jack), the bankers stepped in, and in 1989, the company was taken private (i.e., kind of like firing your boss).

Since then, a "new" Zilog has risen from the ashes of the '80s with increasing sales and healthy profits in the '90s. Sure, they'll never be an Intel or Motorola, but that doesn't mean they can't have a decent business and deliver some neat chips just the same.

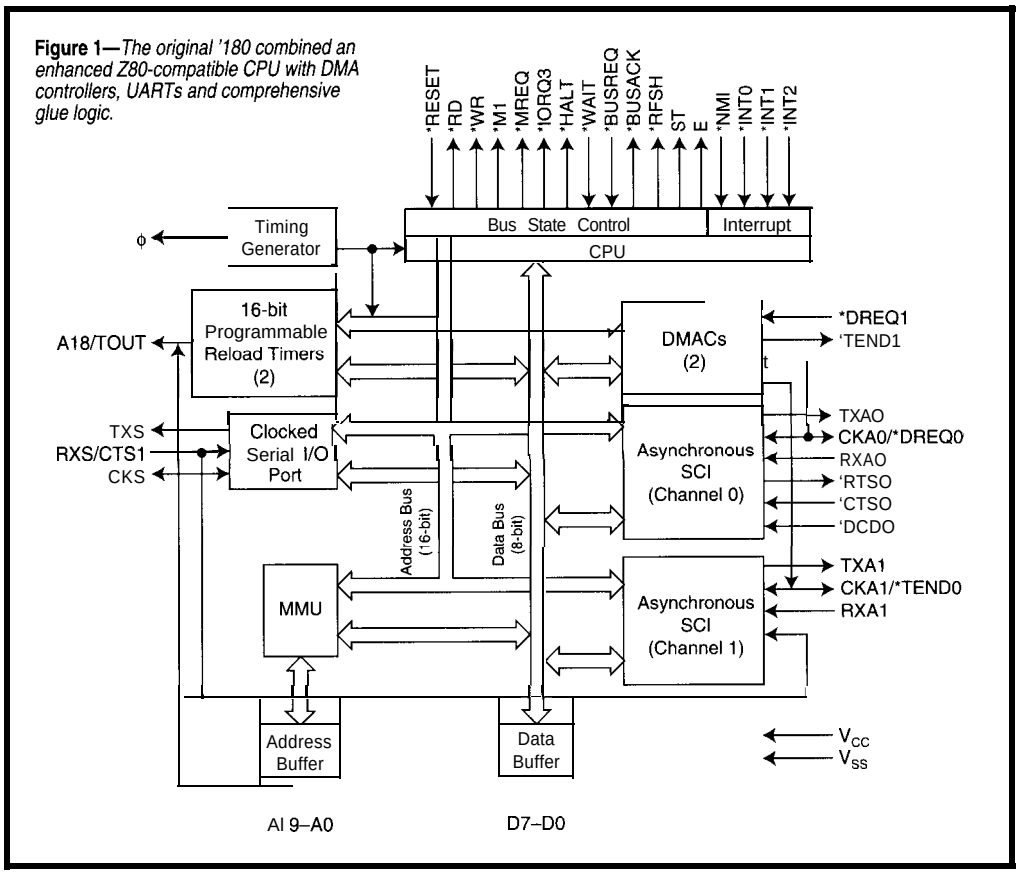


Figure 1—*The original '180 combined an enhanced Z80-compatible CPU with DMA controllers, UARTs and comprehensive glue logic.*

FLASHBACK

It's somewhat ironic that, nearly twenty years later, Zilog still counts on descendants of the Z80, CTC, PIO, and SIO as a cornerstone (along with Z8s and DSPs) of their business.

A brief Z80 history for those of you who weren't around (or can't remember) starts with the introduction of the 8080 by Intel, defining the core of the programmer's model and instruction set that persists largely unchanged today.

Back then, things were pretty loosey-goosey on the legal front, so a bunch of the 8080 team simply went across town and formed Zilog. Whether due to lack of legal savvy, 'x86 distraction, or simple fairness (Intel did much the same thing to Fairchild), Zilog was allowed to sell their enhanced 8080 clone, the Z80.

For customers who'd bought into the 8080 but didn't relish having 16 bits crammed down their throats, the Z80 was welcome indeed.

First, the programmer's model was upgraded with a number of features, (index registers, block-move instructions, and alternate register set). Depending on the applications, some of these were useful, some not so. But overall, the Z80 struck a good balance between enhancement and compatibility.

For the hardware designer, the Z80 was a vast improvement over the 8080. Single +5-V versus the 8080 triple-power supply, single chip versus the 8080/8224/8288 trio, built-in daisy-chain interrupt scheme (i.e., no need for 8259), DRAM refresh, and so on.

Intel made a belated and, consumed as they were with the forthcoming 'x86 campaign, half-hearted response with the 8085, but few chose it over the 280.

Of course, this rosy time for the Z80 was also the beginning of the end for Zilog. By the early- to mid-'80s, they were well into the Dark Age. The Z80 sputtered along on autopilot, effectively falling behind as 8-bit competitors (68xx and 805 1) moved ahead.

Fortunately, Hitachi stepped into the breach with the HD64 180. Taking inspiration from both the Intel '186 and single-chip MCUs, the '180 combined a freshened Z80 CPU core with a rather extensive complement of glue and I/O logic. The '180 kept up with competitors and achieves notable success to this day in modems, faxes, and printers.

The '180 carried the Z80 torch through the tough times, and now Zilog has picked up the ball again. Besides the circa-'80s stock Z180 (i.e., same as HD64180), they've concentrated on fine-tuning the CPU for both

```
Zilog Z80180 Monitor V. 1.4
Z80180>Help

A - Alter Memory
B - Set or Show Breakpoints
C - Compare Memory Data
D - Display Memory Data
E - Edit/Display I/O Data
F - Fill Memory
G - Go to Program
H - Help
I - Input Byte From I '0 Address
K - Kil1 Breakpoint(s
L - Load Hex File
M - Move Memory to Memory
N - Change Serial Data Rate
O - Output Byte to I/O Address
R - Display/Alter Registers
S - Step (over Subroutine CALLs
T - Step (into Subroutine CALLs
U - Disassemble Instructions
V - Display Version of Monitor
X - examine the MMU
```

Figure 2-The 'S180 monitor has all the usual commands, which includes the handy disassembler shown in action in Figure 3.

speed (the 'S180 up to 33 MHz) and low power (the 'L180 at 3.3 V).

CHIP OFF THE OLD BLOCK

I covered the 'S 180 you see in Figure 1 in an earlier article (INK 29, "I'm 18.432—and I Like It"). Recently, a press release from Zilog announced a low-cost (under $200) 'S180 evaluation and emulation board.

The setup you see in Photo 1 consists of a "brain" that's essentially an 18.432-MHz 'S 180 SBC including EPROM, RAM, PLD, and RS-232 interface (DE-9) and a PLCC plug-in pod that mates to the bottom of the brain, with the whole shebang dropping into an 'S 180 CPU socket.

Meanwhile, the other end of the connectors that mate brain and pod serve double-duty, emerging as pins on top for connecting a logic analyzer.

Admittedly, the form factor calls for a fair amount of head and shoulder room for installation. Otherwise, it's a quite workable solution, especially given the bargain price.

Like other low-cost emulators, this one works by effectively constructing a pin-compatible single-chip CPU with ROM and RAM running a monitor program that talks to the PC via RS-232.

Thus, it doesn't have fancy real-time trace (you can use a logic analyzer instead), complex breakpoint, or nonintrusive debug features. Also, "soft" emulators of this ilk invariably impose some restrictions on the target design.

For instance, the emulator switches control between the local and target memory by gating the

- MREQ (Memory Request) line to the pod. Thus, the target must qualify its own memory-enable logic with *MREQ (which is typical practice).

Also, there's no easy way to debug an application's use of a few core functions required by the emulator, such as a couple of the RST vectors (restart instructions are for breakpoints and monitor services) and *NMI (nonmaskable interrupt).

Changing the CPU clock rate requires soldering a new crystal and rebuilding the EPROM. The emulator traps to the monitor when executing an instruction at 0000H, so you've got to ORG your programs at 0001H.

The monitor contains the usual commands (see Figure 2), allowing you to look at and manipulate memory, registers, and I/O. One helpful addition is a simple disassembler (see Figure 3), handy for quick-and-dirty patchwork. As expected given the bargain price, there's no special hardware for breakpoints, which are thus limited to instructions in RAM.

That's OK though, since the unit includes a socket for emulation RAM. It's shipped with an 8 K x 8 RAM, but can handle a larger chip within the limit imposed by the 28-pin socket [i.e., 32 K x 8). This handles the typical less-than-64-KB three-chip system, but proves confining for larger applications, which should probably use industrial-strength tools anyway.

When it comes to software, needless to say, there's no shortage of Z80 tools. Besides quantity, a lot of it is very good quality, having been well-debugged and optimized over the years. I don't know about you, but it's comforting to use software that hasn't had a bug since disco!

If you're new to the '180 and don't already have favorite tools, the kit includes a decent PC-based assembler (macros, expressions, lots of pseudo-ops), linker, assorted object-file utilities, and a simple terminal program.

Knowing that the best way to check out the emulator was with a known good system, I rummaged through my board stash to find the '180 SBC that was upgraded from 9.216 to 18.432 MHz in the "I'm 18.432" article. Yes, there it is, the one with the 45-ns EPROM and 'ALS decoder to handle the higher speed.

Being the cautious type, before lashing the whole setup together, I verified the SBC still worked-good, no rusty transistors.

Next, I connected the emulator brain alone (i.e., not plugged into the SBC) to the PC RS-232 port, running it as a stand-alone EV board. The emulator DE-9 is female which required a male-male adapter to work with my DE-9-to-DB-25 adapter gadget.

Powering the emulator up and hitting carriage return, I was greeted with the monitor signon. Excellent! Other than the cable sexual-identity crisis, nary a single RS-232 hassle.

Feeling rather pleased with my progress, I plugged the emulator into the PLCC adapter pod and then the '180 SBC. Making one last wiring and pin-l check, I hit the power switch.

```
Z80180>Disassemble Starting at (just CR = From PC): 6875
Number of Instructions : 8

6875 ED3834    LD   A,(34)
6878  47       AND  B,A
6879  E63F          A,3F
687B ED3934    OUTO (34),A
687E CB78      BIT  7,B
6880  2806     JR   Z,6888
6882  215778   LD   HL,7857
6885 CDA275    CALL 75A2
```

Figure 3—Here's the Disassemble command in action.

LOST WEEKEND

Actually, it was only Saturday night and Sunday morning, but given chores and familial interface duties, it pretty much shot the whole thing.

Of course, the emulator/SBC setup was dead, the screen mute

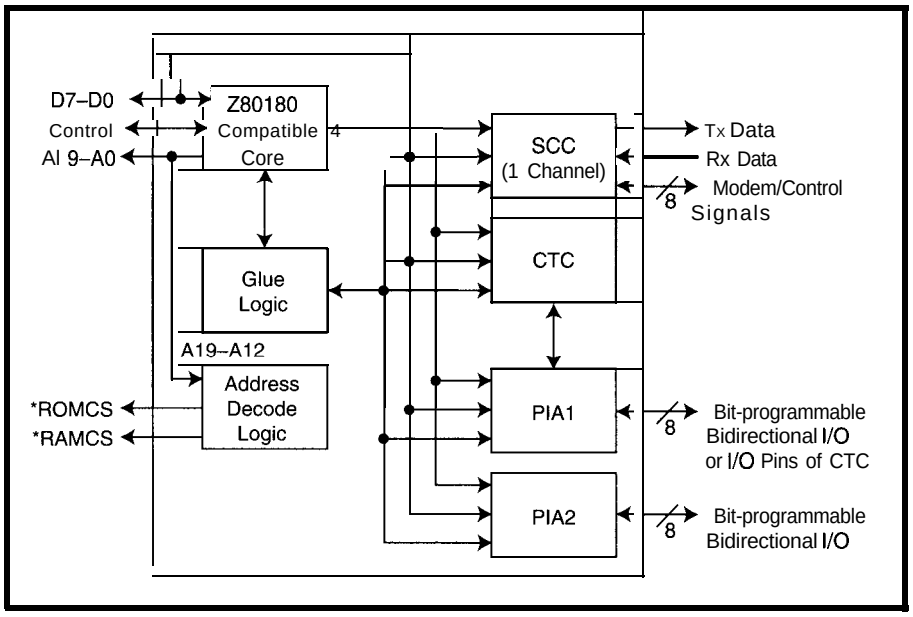


Figure 4—*With* a double-dose of **integration,** *the Z181* combines a complete *'180* CPU (including ifs *DMAC, UARTs,* and *timers) with* additional WART, timers, *and PIO.* The *only question* is: where did they *put the kitchen* sink?

testimony to my pitiful, and now demonstrably futile, optimism.

I paused to consider my situation. Quitting now would go against my natural instincts. And, the work I'd put into the article so far would go to waste. On the other hand, going forward had an uneasy "Forsake hope all ye..." feel about it.

Oh well, if I'm going over a cliff I couldn't ask for better, or more familiar, company than the '180. Heck, I got this stuff working before when I upgraded in "I'm 18.432." I can do it again.

Gathering all the schematics, manuals, data sheets, and my trusty logic probe, I poked around, reading the fine print. I suggest Zilog include a schematic and/or some kind of pinout diagram to identify the connections between the PLCC pod and the emulator CPU-PLD combo that runs everything.

For instance, the emulator has jumpers to determine which of the '180 UARTs is used for the console. However, it wasn't clear exactly how that same UART port connects to the PLCC socket and, thus, interface logic on the SBC.

Since the monitor software waits for a carriage return before signing on, RxD interference could be the hang-up. I pulled the RS-232 transceiver out of its socket on the SBC to leave the console UART lines open. No dice.

Similarly, the manual kind of explains in words how Reset and NMI switches and jumpers on the emulator interact with the target, but a connection diagram would make it clear. Probing these and other key signals, nothing looked obviously out of line. The emulator was running, just not getting anywhere.

To try to get my hands around the problem, I started sticking a small piece of tape on the PLCC pod to isolate sets of pins. Using a kind of binary-search technique, it wasn't more than a few plug-ins later that I had it down to one pin. With the trouble-maker taped off, the emulator would boot when plugged into the SBC. Dare I hope....

Unfortunately that pin was V_{CC}, dashing my fantasy of a quick fix. Yeah, the emulator works fine, just as long as the target isn't powered!

The manual mentions the issue of power-supply integrity. Basically, you can power the target via the emulator, power the emulator via the target, or power them both separately. I tried all the permutations without success.

I figured I'd better check out that MREQ precaution, just to be sure. I quickly flipped to the memory interface page of the SBC schematic, remembering it from the earlier article.

Yes, there's the '138 decoder (the one that had to be upgraded from 'LS to 'ALS), qualified with *MREQ plain as day. Just to be sure, I probed the decoder and confirmed no memory-select output activity.

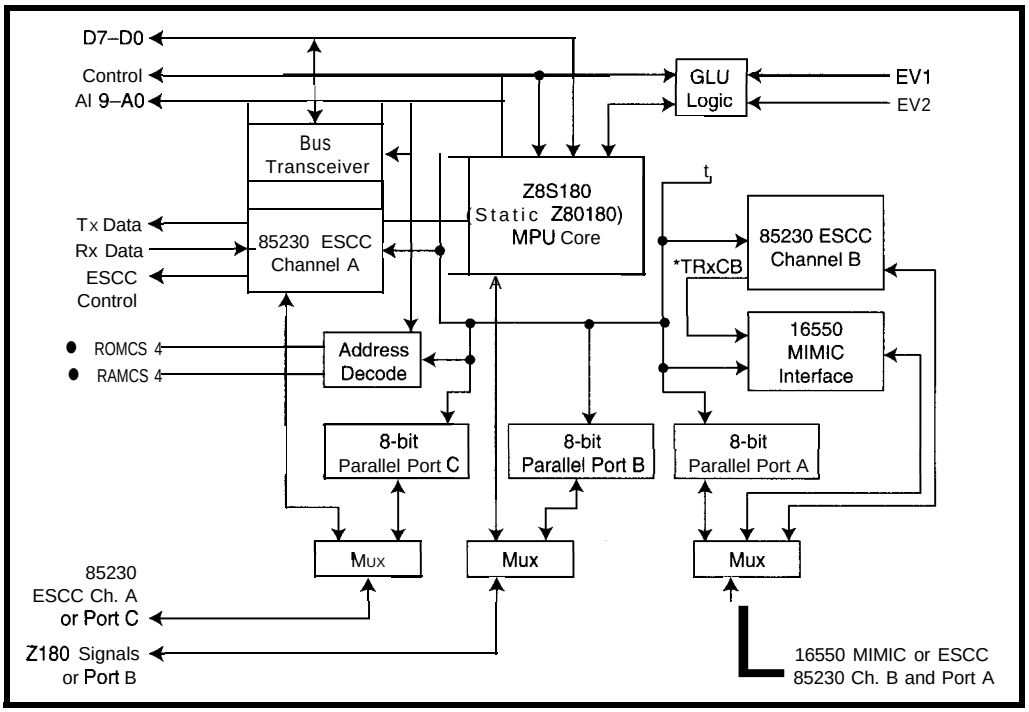


Figure 5—*With* an *extra USART and* easy PC plug-in via *the* Mimic interface, *the Z182* is idea/ *for communications* applications.

Getting desperate, I pulled the memory. At this point, there was little recourse but to keep pulling chips. If I pulled them all, I'd find the trouble or have plenty of ammo for calling Zilog.

Almost as a matter of formality, I decided to take one last glance at the schematic before descending to such a dubiously brute-force debug tactic. Is there any other chip on the SBC likely to be butting in?

Let's see, there's the memory decoder. There's the memories. There's the PIOs. There's the data-bus buffer. There's the other data-bus buffer.. . .

Aargh-staring right at me off the first page of the schematic, and a quick glance confirms still snugly tucked in its socket, is another '245.

I'm not surprised it's enabled by the *RD (not the *MREQ). Even though the SBC memories weren't enabled, the '245 was scrambling the data bus.

Pull the '245—boom, the emulator-SBC combo works. Plug the '245 back in, but leave the enable pin hanging out and jumpered to ● MREQ-it works. Plug the memories and most of the other chips in-it works.

As suspected, installing the SBC RS-232 transceiver interferes with the console interface. But now, with everything working, it's easy to confirm that the RS-232 interface on the target can be used just by removing all the UART jumpers on the emulator.

Having paid my dues the night before, everything proceeded without any major problems, just a few small glitches along the way.

One annoyance is that the monitor, which otherwise is completely happy with any terminal emulator program, adds an unwelcome feature for the download hex-file command. It's a special protocol that works in concert with the included terminal emulator program, so you can type in a filename to be downloaded.

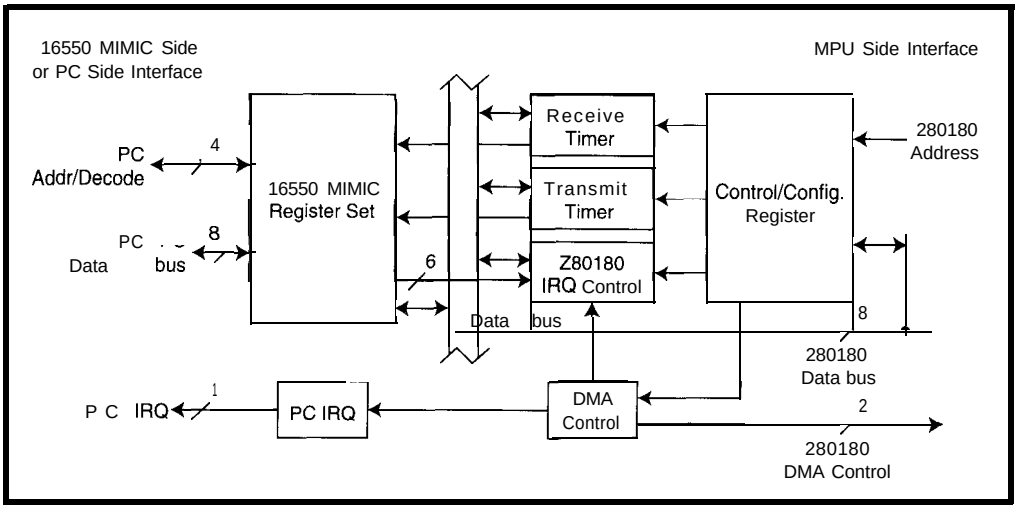


Figure 6—*The aptly* named 16550 *MIMIC* logic *impersonates* a *PC UART, including FIFOs, for compatibility with existing communication software.*

I took a quick glance at the offending code in the monitor source (included) to see if I could fake it into working with my preferred terminal software, but didn't succeed. Since you can't go far without downloading a hex file, you're pretty much forced to use the terminal emulator Zilog provides.

Other potential headscratchers relate to the fact that the emulator initialization code doesn't leave the CPU in the Reset state. If the programs you're testing expect things to start up a certain way, watch out. I found I had to use a monitor command to initialize the MMU to the more Reset-like memory map expected by my program.

Similarly, I found a Zilog demo program that wouldn't work unless the monitor set the stack pointer ahead of time [though the manual says it's supposed to be handled automatically).

Another potential toe-stubber is that the emulator initializes itself to run in turbo (XTAL/1) mode while a real 'S180 chip defaults to '180 compatibility (XTAL/2) at Reset. Adjust [or rebuild) the monitor to slow down if your target only handles XTAL/2.

Fortunately, since the target SBC was already upgraded for full-speed operation, everything worked grand, including zero-wait-state execution out of the SBC's EPROM and proper operation of the overlay RAM.

NEW CHIPS ON THE BLOCK

Today, Zilog is replaying their '70s strategy by combining the '180 CPU with descendants of those old-favorite peripherals. They have some heavy-duty 8-bit chips indeed-contenders against 16-bit chips (including low-end 68ks) and even embedded PCs, especially for communication applications.

Besides speed, low power, and high integration, the new chips are ideal for downsizing (the product, not the job). Photo 2 traces the '180 from its original 64-pin shrink-DIP to its latest 100-pin VQFP, a ~10× space saving.

Consider the Z18 1 in Figure 4-a modern reflection of the classic Z80 and peripheral combo. Remember, the '180 core carries its own I/O forward.

So, you're looking at a list of features matched by few other micros: CPU; two DMA channels ('180); two UARTs ('I 80); one USART (SCC); 2-channel, 16-bit timers ('180); 4-channel, 8-bit timer/counter with prescale and external clock

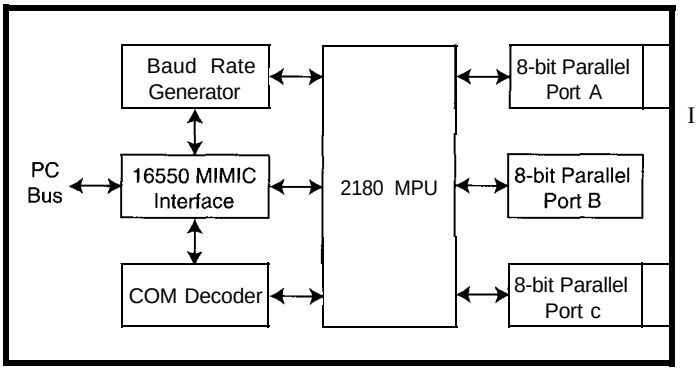


Figure 7—*Replacing the '182's USART with additional PC interface logic and I/O makes the Z189 a general-purpose coprocessor.*

and trigger (CTC), and 16 parallel I/O lines (bit-programmable PIO).

By the way, the "S" in USART stands for "synchronous." Besides working as a UART (i.e., the "A"), the SCC in the 2181 and other Z18x chips' claim to fame is the ability to manage serial links under a regime know as HDLC (High-level Data Link Control).

HDLC is rather complicated stuff that could easily consume an entire article-I'll put it on the to-do list. But, the bottom-line is: HDLC is at the core of many communications applications including modems, ISDN, and T1. One familiar HDLC-based network is Appletalk (which was, in fact, designed around the SCC).

For higher performance than the 2181, especially for PC communications applications, the 2182 (see Figure 5) switches to the 'S180 core, adds a second SCC channel, and boosts parallel I/O to 24 bits.

Key to easy PC design-in is the 16550 MIMIC logic, which mimics the 16550 UART in PCs, so existing communication software (i.e., anything looking for COM1-4) works transparently. You see the MIMIC block diagram in Figure 6.

The 2189 shown in Figure 7 is a variant of the 2182. As the chips inside a modem or fax shrink in number, so a particular function may migrate.

Turns out, the latest modem datapumps has taken over the HDLC formatting, rendering the '182 ESCC superfluous-so, it's removed from the '189. What's left is an enhanced (with faster UARTs and linked DMA) 'S180 CPU, the MIMIC, some extra PC interface logic (COM port decoder and DMA interface), and 24 I/O lines.

Besides communication applications, I suspect the '189 makes an excellent data acquisition and control subsytem processor, since high clock

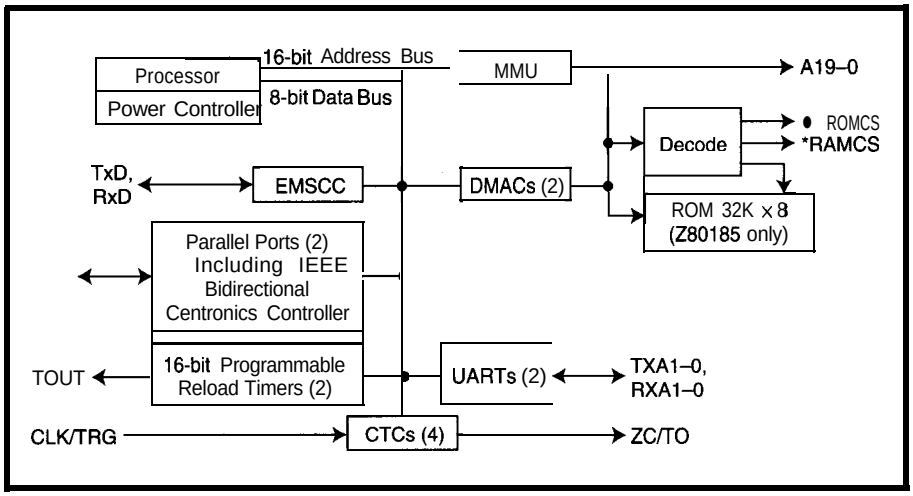


Figure 8—*The Z185/195* are among the first chips to handle the latest and greatest *IEEE P1284* PC *printer-port protocol. The '185 is a/so* notable for inclusion of 32 *KB of* zero-waif-state ROM.

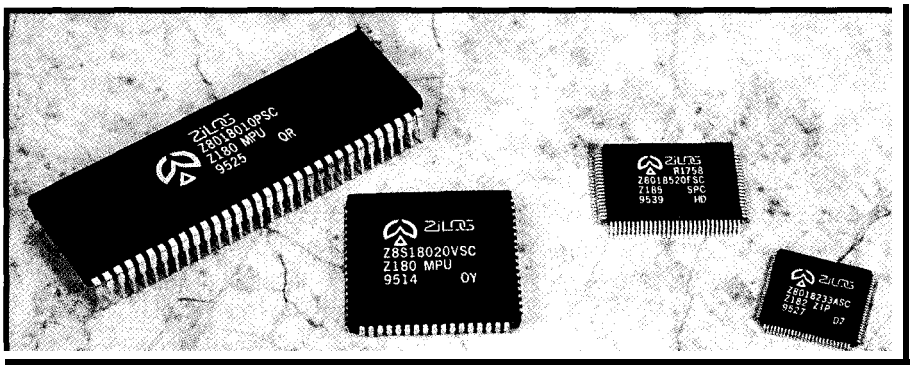


Photo 2—*The* incredible shrinking '180 *in (from left to right) 64-pin shrink DIP, 68-pin PLCC, 100-pin QFP, and 100-pin VQFP.*

rate, on-chip DMA controllers, and MIMIC FIFOs (16 bytes each direction) ensure speedy data transfers.

The Z185/195 you see in Figure 8 is another take on the communications angle. It's kind of like a Z182 with an IEEE P1284 (bidirectional Centronics) port taking the place of the MIMIC.

Conceptually, it makes sense to consider MIMIC and P1284 as simply different ways to plug into a PC. The '185/195 works as either a P1284 master or slave, so it can be used both in a system and/or peripheral.

Most interesting, the 32 KB of ROM included on the '185 runs with zero waits for full performance. Still on my wish list: turn the ROM into EPROM or flash and add a little RAM. That would make for one macho MCU.

With volume (50k+) prices from $3 to $5, the Z180 (including the regular, S, and L versions) and the new Z18 1 ($7-9) remain formidable competitors in the midrange and high end of the 8-bit market. Meanwhile, chips like the 2182, '189, and '185/'195($8–12) are hard to beat for PC plug-in and other communication applications, especially those calling for HDLC.

All in all, the Z18*x* family proves that chips, like the Phoenix, can rise from the ashes to soar again. ❑

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for more than ten years. He may be reached by E-mail at *tom.cantrell@circellar.com, by telephone at (510) 657-0264 or fax at (510) 657-5441.*

SOURCE

Z18*x*
Zilog, Inc.
210 E. Hacienda Ave.
Campbell, CA 95008-6600
(408) 370-8000
Fax: (408) 370-8056
http://www.zilog.com/

I R S

422 Very Useful
423 Moderately Useful
424 Not Useful

CONNECTIME

conducted by Ken Davidson

The Circuit Cellar BBS
300/1200/2400/9600/14.4k bps
24 hours/7 days a week
(860) **871-1988—Four** incoming lines
Internet E-mail: **sysop@circellar.com**

This month, we start with a short discussion about battery-voltage defectors. While such a circuit may not sound like a big deal, when you're powering if off the battery you're measuring, there is more to think about.

Next, I'm often forced to overlook long message threads due to space constraints. However, I have some extra room this month, so picked an interesting discussion that starts by covering how to make high-current connections, and ends talking about conformal coatings. Talk about thread drift!

Battery loss

Msg#: 3289
From: Ron Turner To: All Users

I need a circuit that detects a battery voltage of 11 V so I can tell when a car battery drops below this voltage. Does anyone know of any such circuit?

Msg#: 3304
From: Michael Sharp To: Ron Turner

Look at a simple comparator circuit built around a 741 op-amp. You put a reference voltage on one input pin (the 11 V) and the varying voltage on the other input pin. The output of the amp goes high or low depending on how you set it up.

Radio Shack offers *Engineer's Mini-Notebooks* wrote by Forrest Mims III that would explain this circuit ("Op-Amp IC Circuits" book).

Msg#: 3307
From: Russ Reiss To: Ron Turner

Maxim and Linear Technology sell a number of nice comparator chips with built-in references that would meet your needs. Get their databooks and check it out.

Msg#: 3361
From: Ken Simmons To: Ron Turner

How about a simple LM3914 dot/bar driver?

Wire it in dot mode, feed the supply directly into the input, and when the "11 volt" LED output energizes, have it drive your alarm trigger. In other words, the V+ supply is also the signal input and enables the appropriate LED output (sinking drivers!) depending on the level.

I have an old car battery analyzer built around a similar chip and it works great. Just follow the hookup diagram that's on the Radio Shack blister pack for the 3914.

Msg#: 3619
From: Larry G Nelson Sr To: Ron Turner

How about a simple comparitor circuit? Use a voltage reference to provide a stable reference point and then a simple resistor divider to sense the battery voltage. As the battery voltage drops, the divided voltage drops. The voltage reference (a zener) does not change as the supply voltage drops and the comparitor can see the difference. If this is in a car, be aware that the voltage drops drastically when you run the starter and that could trip the circuit if you do not have a time delay to filter a short-term drop.

Msg#: 3332
From: George Novacek To: Ron Turner

I am assuming you want a circuit which is also operated from the same battery. What the other gentlemen suggest (11-V reference) will not work without some modifications.

You can always start with a comparator. You will need a reference lower than 11 V, such as 5 V-easy to get. The other comparator input you feed through a suitable voltage divider. Also, I would use a comparator with an open-collector output (e.g., LM139 or one of the linear CMOS quads which have very high-input impedance and a low-current drain), which is much easier for interfacing with other circuits.

Another detail to note: you need hysteresis. When you unload the battery, its voltage is going to creep up. Without hysteresis, you may see some slow oscillations. What you want is a setup where the lower hysteresis point is at the minimum allowable battery voltage, and the high end is slightly below fully charged battery.

There are several integrated circuits suited specifically for this purpose (battery chargers/monitors). Or, you can use a Maxim watchdog IC-most have a built-in voltage monitoring circuit. You just add a proper resistor divider.

If you need very low power consumption, get a smoke detector IC. These are used to monitor a 9-V battery over a minimum of one year. You end up with just a few microamperes of drain. But, you need a voltage divider since these circuits trip at 7.7 V.

You can also build one from discrete parts, starting with a zener, a resistor, and a transistor (you can use a reverse-biased base-emitter junction of a transistor for a zener with only several microamperes knee). With a PUT, I can show you how to build a battery monitor with no more than 1 μA drain, should you require it.

Bus bar

Msg#: 3511
From: Al **Dorman** To: All Users

Imagine this. You have a PC board carrying a lot of heavy AC amps-80 to be exact. The drivers are eight 10-A triacs connected to different loads. All of the MT1 terminals route to a large area of tinned copper on the board. You want to be able to connect to the common of these 10 triacs with a large-diameter power cable and run it back to the main breaker box.

For obvious reasons, you don't want to solder the power cable right to the PC board. Would you bolt a piece of thick, fat bus bar to the board? Would you rivet it? How about soldering it?

The bus bar would stub out the end of the board and have a power terminal on it to capture the power cable. Which solution would allow for zero ohms at the bus bar to PC board connection, not oxidize as time goes on, or cause an IR loss due to dissimilar metal contact?

Now for the kicker. There are four of these nodes spaced 5" apart, and they all must stub out to one power terminal for connection to the main breaker panel. A total of 320 A....

Msg#: 3590
From: Ken Simmons To: Al **Dorman**

I suggest using a massive bus bar for the main common and running a piece of fat-gauge (4/0 or larger) braid from the main board to the bus bar. Use some anticorrosion compound (Caig Laboratories makes some interesting ground plating compounds that might do the trick!) between the braid/board and braid/bus-bar connections to minimize or eliminate corrosion and electrolysis effects.

Using braid interfacing rather than a physical bus-bar mounting greatly reduces the mechanical stress on the main board.

Msg#: 3593
From: Al **Dorman** To: Ken Simmons

The board will be 0.092" thick to handle mechanical stress along with strategically placed standoffs. This paste that you mentioned sounds like the thing I need for long-term reliability. I'll try to find Caig's number unless you have it handy. Thanks for the input, Ken.

Msg#: 3710
From: Ken Simmons To: Al **Dorman**

Even a 0.092" board may not adequately handle the physical stresses you're asking it to.

Caig Laboratories, Inc.
16744 West Bernardo Dr.
San Diego, CA 92127-1904
(619) 451-1799
Fax: (619) 451-2799

Msg#: 3613
From: Pellervo Kaskinen To: Al **Dorman**

Here is what I have done in a similar situation.

I put on the printed circuit board several ¼" wide quick-disconnect tabs. Then I made a fanned-out cable with quick connects at the spread out ends and a big ring lug at the other end. In my case, it was a 50-A application, with six 14-AWG wires. At the common end, there was a crimp-type ring lug intended for a single 8–10-AWG stranded wire. The stud size was ¼".

You could do the same using eight individual wires. You might want to use a ⅜" or ½" stud for the total current requirement. Then you would stack the four ring lugs on a stud to get the total of 320 A. And, another one or two ring lugs for a size 0000 (single cable) or 00 (two cables) would be ready to be stacked in to carry the current away.

The quick-disconnect tabs are readily available. They have two legs that you have to solder to the PCB. A general problem is the solderability. Our tabs always are old by the time we try to solder them and that does not help. You may need to pretin them or use some rather active flux. A double-sided board with plated-through holes may be easier than my single-sided board, but I would still advise you to check for the optimum hole size.

Msg#: 3629
From: Al **Dorman** To: Pellervo Kaskinen

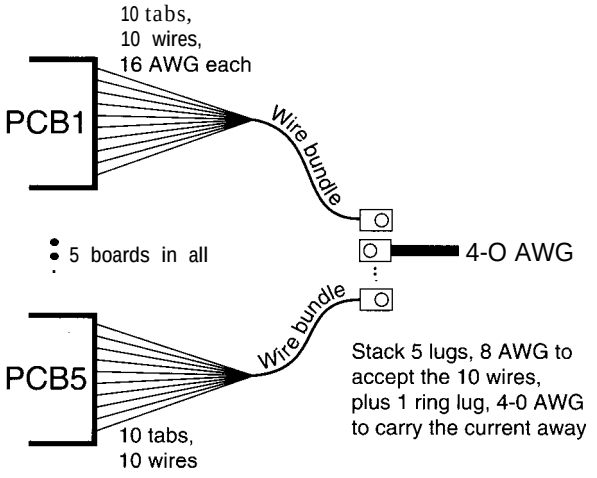
I have seen 0.250"-wide quick disconnects and use them in some of my other products, but have never seen ones the size and current capability that you are describing. Where would I get more info on these monsters?

Msg#: 3883
From: Pellervo Kaskinen To: Al **Dorman**

You have seen them, you just have not realized their potential! The 0.250" quick connects are rated 10-A depending on the other side of the pair and the wire size. I hope I did not lead you to believe they would be good for

the 80-A portion. That has to be handled by the ring lugs on a stud.

Let's see if I can clarify the idea with some graphics.



The stud, of course, has to be mounted somewhere. There are commercial products that may be useful. They consist of an insulating block and a well-anchored stud. Available at least from Marathon, USD, Phoenix, and Weidmuller. Sorry, I did not check the addresses for these companies, or if the first two even exist (under the same name). Weidmuller and Phoenix both exist-I have done business with them quite recently.

One more source I know of is a cabinet maker, Rittal. But of course, you can make your own from 1" fabric-reinforced phenolic block, an ordinary bolt, and some epoxy to lock the bolt head inside the bottom-side counterbore. The epoxy also keeps the bolt head from making a short to any metallic mounting surface.

Msg#: 3692
From: Al **Dorman** To: Pellervo Kaskinen

I decided to opt for a 1"-wide copper bus bar strapped across the boards. This way there is not so much labor involved. The bus bar has two 10-28 studs for common AC connection. I wasn't aware that a 0.25" terminal would hold so much current. Of course, the FAS-PAK triac series used 0.25" slip-on-type connections, and they say you can get 30 A through it with a Molex slip-on female plug.

Msg#: 4147
From: Pellervo Kaskinen To: Al **Dorman**

If the flat tab sits on a heat sink, of course you can get more current capacity. But the normal ratings and what the agencies (UL and others) approve is in the 10-16-A range.

If you look at some circuit breakers or fuse holders, you find somewhere a statement that the flat terminals cannot be used for a current beyond 16 A or so, unless you solder the wire on the tab. In other words, they know that excessive heat causes a problem for ordinary quick-connect combinations.

I think the problem is assumed to be mostly related to thermal cycling, heat-cool-heat-cool, and so on. But, I do not have any actual data about that.

On the other hand, the semiconductor connections, especially the cables of stud-mount-type SCR packages, have always been very flimsy in comparison to the normal cables. That is partly due to the fact that the semiconductors are typically operated at a derated current. Another reason for that is the higher temperature specification.

While a typical cable is insulated with a PVC jacket, the SCR connection cables are either uninsulated or have a silicon rubber jacket. The temperature rating for the PVC is about 105°C, while the silicon rubber can be up to 180°C. Not to forget that the SCR bridges are almost always well cooled with a powerful fan.

Anyway, I am against generalizing the 30-A rating your FAS-PAK suggests.

Msg#: 4216
From: Ed Nisley To: Pellervo Kaskinen

Speaking of high current, there's the issue of soldering ground wires. Turns out that a decent lightning stroke simply vaporizes the solder (it's a poor conductor) and leaves you with two disconnected conductors. Welding, yes, bolted connectors, yes, solder, nope.

That's not an issue with ordinary electrical components, but I can't resist tossing it into the discussion.

Msg#: 5099
From: Pellervo Kaskinen To: Ed Nisley

Good point! Yet, I just cannot bite that without a little qualifier.

The age-old rule about soldering electrical conductors is that the solder is there only to seal the connection, not to make it. I recall some quite big campaigns that one TV manufacturer ran to convince prospective customers that their TVs were made to be reliable-like most of the problems in "conventional" (what else?) units were due to failing soldered connections. They, on the other hand, never trusted the solder alone. No sir! They made each connection between the component leads inside a cone of wrapped copper wire, similar to a wire nut. But then, they filled the cone with solder.

The solder, applied that way, was supposed to immobilize the leads plus protect the actual contact points against oxide formation and other dirt attacks.

CONNECTIME

If you have ever built any Heathkit items, chances are you got a good education about the component mounting and soldering within rows of eyelets or other mounting ears. Push the lead in the hole, bend it, don't solder yet! Add other components the same way, locking the component lead tightly in place so it actually works, even without the solder. Then, finally apply the heat and a little later the solder.

Ahh! Those were the days.. Then came printed-circuit boards. You just could not compete with the manufacturing efficiencies, no matter how reliable the solder-cup method was. A single-sided board, a double-sided board. Multilayer boards and surface mounting. Where have we gone with the idea of having the solder as only a seal? Nowadays, solder is actually part of the conductive path.

Regarding the use of flat tabs on printed circuit boards in high-current applications, here is one personal finding to conclude my reminiscences. Ordinarily, the holes on the printed-circuit board should be made large enough to leave about 0.006" space on all sides of the component lead. That makes the wetting action best. You also can desolder such connections much easier, if necessary.

But, the tabs are next to impossible to keep straight and in place during the soldering operation if the holes are any larger than barely large enough. I make the holes barely large enough to accept the two legs of the tab. A certain amount of force is required to force the legs in the holes. So, in practical terms, I'm back to the "electrical contact without solder" principle. The solder is again applied mostly for the purpose of locking the part in place and protecting the connection.

Msg#: 5187
From: Ed Nisley To: Pellervo Kaskinen

You mean, gasp, that "The Bigger The Blob, The Better The Job" isn't the right rule?

Me, I've always been a fan of structural solder.. .
(Grin!)

Msg#: 7198
From: Ken Simmons To: Ed Nisley

You got it!

I was once mil-spec solder certified and if you compare the amount of solder allowed on mil-spec hardware versus good-quality commercial/industrial hardware, the commercial/industrial stuff looks like garbage in relation to the amount of solder on the pads, etc.

In other words, mil-spec makes everything else look like your structural solder definition.

If you want something even tighter, NASA solder spec makes mil-spec soldering look like structural solder by comparison!

Msg#: 4923
From: Al Dorman To: Pellervo Kaskinen

They go on to say that to get 40 A out of a 40-A triac (FAS-PAK), you have to solder a 4-AWG wire directly to the terminal. I think AMP is exaggerating when they say you can get 30 A from a 0.250" slip-on.

Msg#: 7617
From: Pellervo Kaskinen To: Al Dorman

I have not checked the particulars, but my gut feeling is that AMP is actually playing on the weaknesses of the cheap competition.

I have myself been unhappy about some flat-tab connectors made of too-thin metal. They just folded it more for increased stiffness. Might be fine for 20-AWG wires, but not for the thick 12 (or even 10) AWG. And I know there have been changes during the years in the color codes indicating wire size. Nowadays, I have to use a blue one for some sizes that originally went with the red. Maybe UL or somebody else pointed a few things out to the vendors?

On the other hand, with proper tooling, optimum conditions, and fixed room temperature, you might get ultimate temperatures within acceptable values even at 30 A per tab. Just do not take it as a relevant number for any brand, any contact density, and any ambient temperature. In fact, the neighboring contacts put out heat, which affects the ambient temperature. You always need a derating table to find what the realistic values might be.

This reminds me of a difference in fuse-holder ratings between the European and the American specifications. The American tests are conducted with a fuse-size solid-copper bar in place of the fuse. This naturally causes less heat generation than a real fuse would do, so the holder gets rated for higher current than what the maximum fuse size should be. An American lo-fuse holder may earn a 6.3-A rating in Europe because it is tested with such a fuse in place.

But, the street is sometimes in the opposite direction. Lately, the European cable-size recommendations have started using the UL principles of determining an individual rating and a conduit- or raceway-dependent derating table. The old way assumed "typical" conditions, with increased ambient temperature to be the norm.

You do use the derating table, don't you!

Actually, all I see most of the time is people learning the basic table values: 14 AWG is good for this much current, period. Using the two derating factors given by the NES can easily cut more than one third of the capacity!

Msg#: 13004
From: Al Dorman To: Pellervo Kaskinen

What I have learned, as a rule of thumb, about derating for high-current consumption is to multiply by two. If the

circuit calls for a 20-A load, I use a 40-A triac. If a 10-A fuse is required, I select a 25-A clip to hold it.

After seeing boards burn up due to overheated connections, I assume that oxidization and time are a power connection's worst enemy. If the board is to be used in a harsh environment such as a sign or other locale, this factor multiplies. I use a dry-can desiccant to reduce moisture.

However, all math aside, second guessing Gea involves a more blunt approach. Two times works for me.

Msg#:15368
From: Pellervo Kaskinen To: Al **Dorman**

The factor of two is a little more than the NAVMAT (I think that is what it is called) spec wants you to do. They have based the derating on the power, not the current or voltage. The square law dependency means the current and voltage are derated by a factor of 0.7, while the power becomes derated by a factor of 0.5.

Msg#:13777
From: Ken Simmons To: Al **Dorman**

Don't forget conformal coating as a good option for weatherproofing your board. Urethane-based coatings (like Hysol PC- 18M) are excellent moisture barriers and quite electrically inert. Even silicone RTV coatings, while thicker and a little more expensive, are excellent for real harsh environments.

One disadvantage of conformal coating is high rework cost. It's difficult at times to remove the cured coating (especially silicones!) and you have to recoat the reworked area and let it cure before restoring the device to service.

However, the advantages far outweigh any disadvantages as far as I'm concerned.

Msg#:15369
From: Pellervo Kaskinen To: Ken Simmons

I once was in a seminar where a high-precision-resistor manufacturer described these environment problems and some solutions. The one shocking piece of info was that silicone coating did not work well!

The reason was that silicone did not adhere intimately enough to the component surface to prevent seepage of water. Rapid temperature changes caused the water to expand and further rip open the channel along the component surface. Then cooling sucked more water in and the next heating....

Their solution was a dual barrier. Epoxy that is too hard for the overcoat was good for the inner layer. Very low viscosity to produce a thin layer. Then silicone for *mechanical* reasons.

Msg#:16980
From: Ken Simmons To: Pellervo Kaskinen

Fascinating! Perhaps you should send a dissertation to the DoD and other aerospace electronics manufacturers informing them of this fact.

In my experience with environmental stress screening using silicone-encapsulated PWAs, we've never had any problems with water seepage, infiltration, and so on. Of course, our boards are precoated with primer compound before the actual silicone is applied.

I would think the epoxy would be good for the mechanical securing, not the silicone.

Of course, there are so many formulations of silicone encapsulants, this statement can be both right and wrong at the same time, no?

Msg#:17063
From: Al **Dorman** To: Ken Simmons

I, too, was an advocate of conformal coatings, however over the years and after plenty of rework difficulties, I have changed my views on the advantages of conformal coatings. A weathertight container and a 2" x 3" wafer of desiccant appears to work just as well. Placing the can on or near a radiating heatsink drys out the can and allows for continuous use.

Msg#:18409
From: Ken Simmons To: Al **Dorman**

True, however such an option is not acceptable with military flight hardware that's enclosed in perforated enclosures.

Remember, these things are designed to literally be installed and forgotten, enduring the environment until a failure occurs.

Yes, conformal coatings add a little to the cost, as well as some rework difficulties, but they can't be beat for moisture protection as well as possible short-circuit protection from those unexpected things that invariably end-up making their way into the product (e.g., dust, insects, etc.).

Msg#:19130
From: George Novacek To: Al **Dorman**

Depends on the environment. For some applications, you can't do without conformal coating. If your box is exposed to significant temperature and barometric pressure variations (e.g., on board an aircraft), you must provide a breathing hole in the enclosure (1/32" is all you need) to allow the air pressure equalization and to drain the condensation.

A hermetically sealed box is extremely expensive and difficult to build. Even with gaskets and seals, the box breathes a little (e.g., temperature expansion and contraction, connectors, etc.). There is eventually enough moisture

trapped inside the box to cause problems. (This is not theoretical! It has been shown to happen.)

Making the box hermetically sealed and expandable so that it can equalize pressure by adjusting its volume is not only expensive, but you run into numerous problems with shock and vibration. And still, there are the connectors to worry about.

Modern conformal coating materials allow easy application and full UV curing in about 10 seconds.

Msg#:26269
From: Pellervo Kaskinen To: Ken Simmons

Maybe I didn't make it as clear as I thought. The idea about the epoxy/silicone double coating was not mine. In fact, it came from one of the foremost suppliers of ultra-precision resistors. I have no doubt whatsoever that they are suppliers for NASA and the rest of the aerospace industry.

However, I may emphasize the key points a little more. The epoxy coating was deemed to be too hard and *brittle* to survive the mechanical challenges. Something *softer*, like the silicone was needed for good impact resistance. A combination with silicone inside would not offer any benefits, while the silicone on top does.

Now, this also was not a board coating. It was for individual resistors. The data that was shown covered salt spray, thermal cycling, and just about all the torture tests the MIL system knows of. It indicated an order-of-magnitude improvement over either an epoxy-only or a silicone-only coating performance.

All of that was before Parylene, so maybe it is not valid anymore. Cannot keep truly up-to-date on these issues. The bean counters don't like expensive seminars.. .

Msg#:27148
From: Ken Simmons To: Pellervo Kaskinen

The silicones we use in the DS&G side of Boeing have very high moisture resistance, provided they're applied correctly (i.e., clean surfaces, well-primed). In fact, I've yet to see one of our boards returned for a moisture-related failure. They're usually electrical or mechanical failures.

Now when you talk about individual components, you're dealing with a different area than a completed PWA. Our tests aren't used to qualify vendor components (resistors, caps, etc.), only the completed end-item PWAs. That's where we apply the final silicone (or urethane) conformal overcoat. The individual components are basically on their own in surviving our tests.

Of course, our supply documents only list vendors whose components have been previously determined to meet the required design specs (temperature, shock, humidity, etc.). Thus, it's our PWA assemblies that require the final environmental qualification testing.

Don't believe I've heard of Parylene. Can you elaborate on what it is!

Msg#:36480
From: Pellervo Kaskinen To: Ken Simmons

Parylene is a trade name for coatings that are applied in a vacuum chamber. I once responded to some ad with the appropriate bingo card mark. Received several pages of nice PR with an impressive list of customer names. And too little hard facts, as usual. Moreover, I don't know where I have "safely" filed it.

In any case, I have never used that service and the only exposure that I have had to parts coated with Parylene have been some ferrite toroids. When you see other color than dark gray on a ferrite, it might be coated with Parylene.

If I understood, the vacuum process makes the coating extend to all places, leaving no "shadows." Also, you do not get the ridges a dip and dry process leaves. And, with the thin coating you can maintain close tolerances.

We invite you to call the Circuit Cellar BBS and exchange messages and files with other Circuit Cellar readers. It is available 24 hours a day and may be reached at (860) 871-1988. Set your modem for 8 data bits, 1 stop bit, no parity, and 300, 1200, 2400, 9600, or 14.4k bps.

ARTICLE SOFTWARE

Software for the articles in this and past issues of *Circuit Cellar INK* may be downloaded from the Circuit Cellar BBS free of charge. It is also available on the Internet at **http://www.circellar.com/**. For those with just E-mail access, send a message to **info@circellar.com** to find out how to request files through E-mail.

For those unable to download files, the software is also available on disk. Software for issues prior to 1995 comes on a 360-KB IBM PC-format disk, one issue per disk. For issues from 1995 on, software comes on a 1.44-MB PC-format disk, with three issues per disk. Disks cost just $12 each. To order Software on Disk, send check or money order to: Circuit Cellar INK, Software On Disk, P.O. Box 772, Vernon, CT 06066, or use your Visa or Mastercard and call (860) 8752199. Be sure to specify the issue numbers with your order. Please add $3 for shipping outside the U.S.

I R S

425 Very Useful 426 Moderately Useful 427 Not Useful

PRIORITY

Two of the concepts I remember most about working in a couple of large aerospace companies were the ideas of "designing by committee" and "debugging by default."

First, you take a bunch of engineers and managers and sequester them in a conference room. Lunch, dinner, snacks-whatever-are brought in for however many days are required for management to hammer engineering into submission. Vacations are canceled, sick leave is scorned, bathroom trips are supervised.

After a couple weeks of this routine, and depending on the deviousness of management, the engineering staff has not only become enlightened to the prospects of creating cold fusion and perpetual motion, they've taken full responsibility for creating whatever management thought up. Of course, the engineering staff gets to determine which poor souls actually do the job.

At the bottom of the "to do" list-the last task to be considered-is making it all work (i.e., debugging it). In my large-company experience, this is when everyone heads for the hills. The unfortunate guys without trips, meetings, conventions, or other "important business stuff" get scope-probe duty. Invariably, the newly hired engineers are in this group. Needless to say, they can forget free time and vacations for the next six months.

I'm sure I'm not telling you anything new. If you work for a small company now, your past experience includes a stretch as a disposable debugger. By now, you're positioned to delegate the task to others. If you're still in a large company, you probably have enough experience to come up with some really "important business stuff" and avoid the problem entirely.

For the rest of us-and in this lot I'm including those without the power to delegate as well as masochists like myself who feel that holding a scope probe now and then keeps us closer to the pulse of the business-the debugging process can be either a challenge or a sentence.

Somewhere at the bottom of the heap are these people and tools that make products sensible and designs a reality. Like running a marathon, you need the most commitment and the most effort when you're heading to the finish line. Since human nature, unfortunate as it is, only rewards the winner, you've got to get there.

Even with all the implied responsibilities, I'd like to think we save the best experiences for last. The unsung heroes who actually make the committee design functional achieve this goal through hard sweat and tenacious resolve. In the process, they learn the most important lesson: how to do or not do things next time.

Product debugging isn't a static encounter. Good engineers take the experience and try to reduce the debugging required on their next effort. The others find out that debugging often takes more talent than they have. For them, just being part of the committee or doing other "important business stuff" is safer and less demanding.

steve.ciarcia@circellar.com