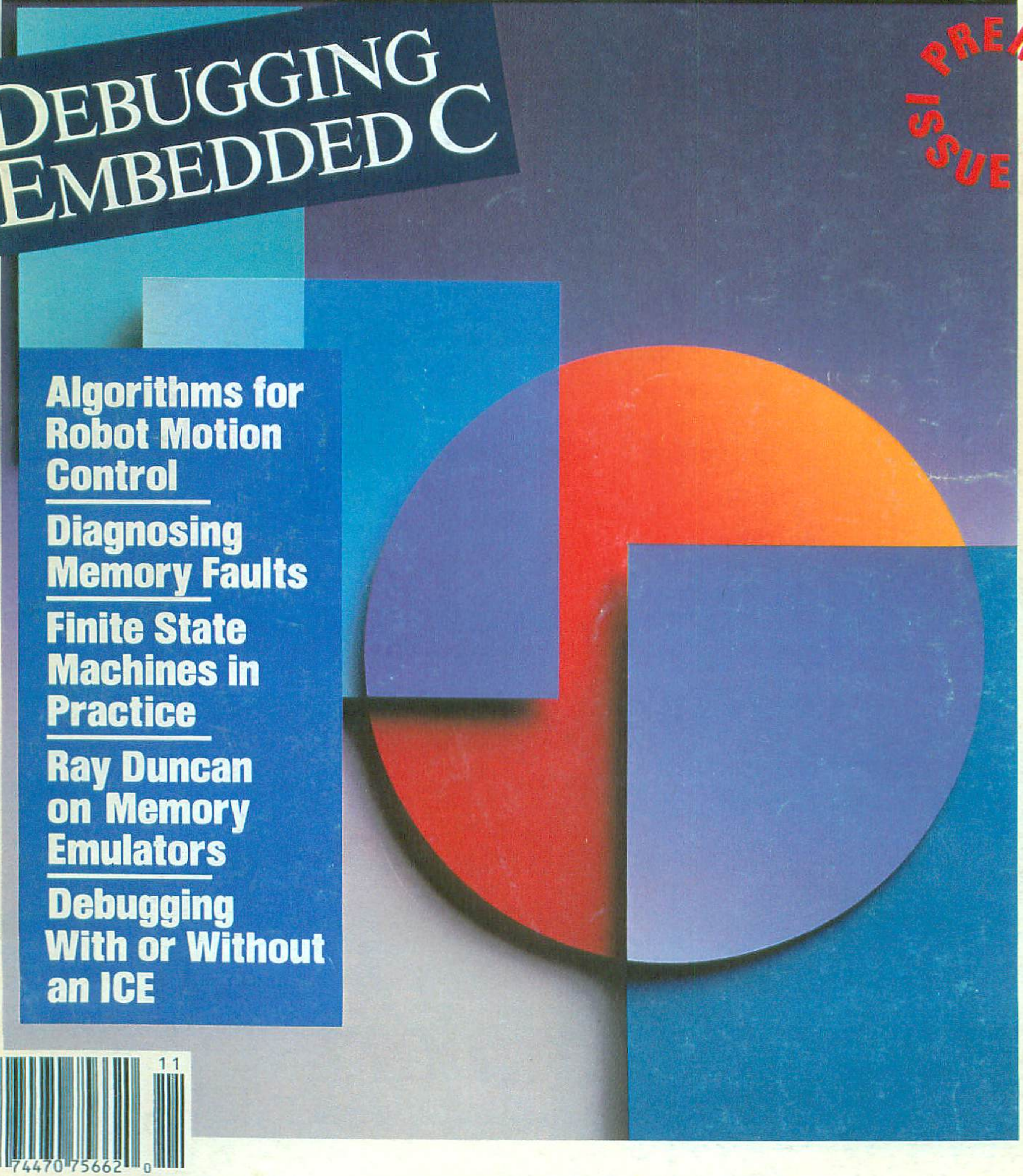
VOLUME 1, NUMBER 1> $3.95 (CANADA $4.95)
PREMIERE 1988> A MILLER FREEMAN PUBLICATION

Embedded Systems PROGRAMMING

PREMIERE ISSUE

DEBUGGING EMBEDDED C



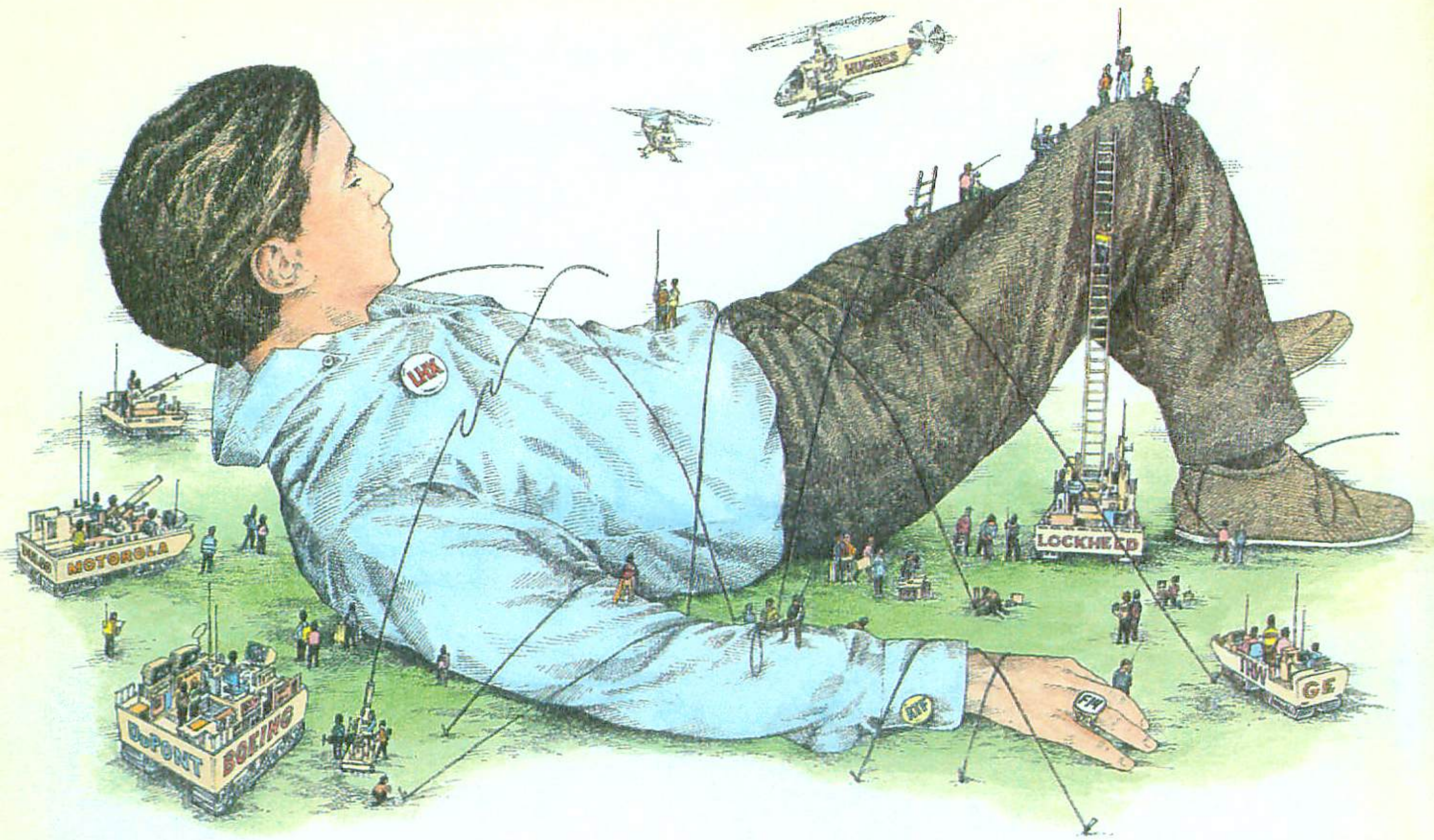
Algorithms for Robot Motion Control

Diagnosing Memory Faults

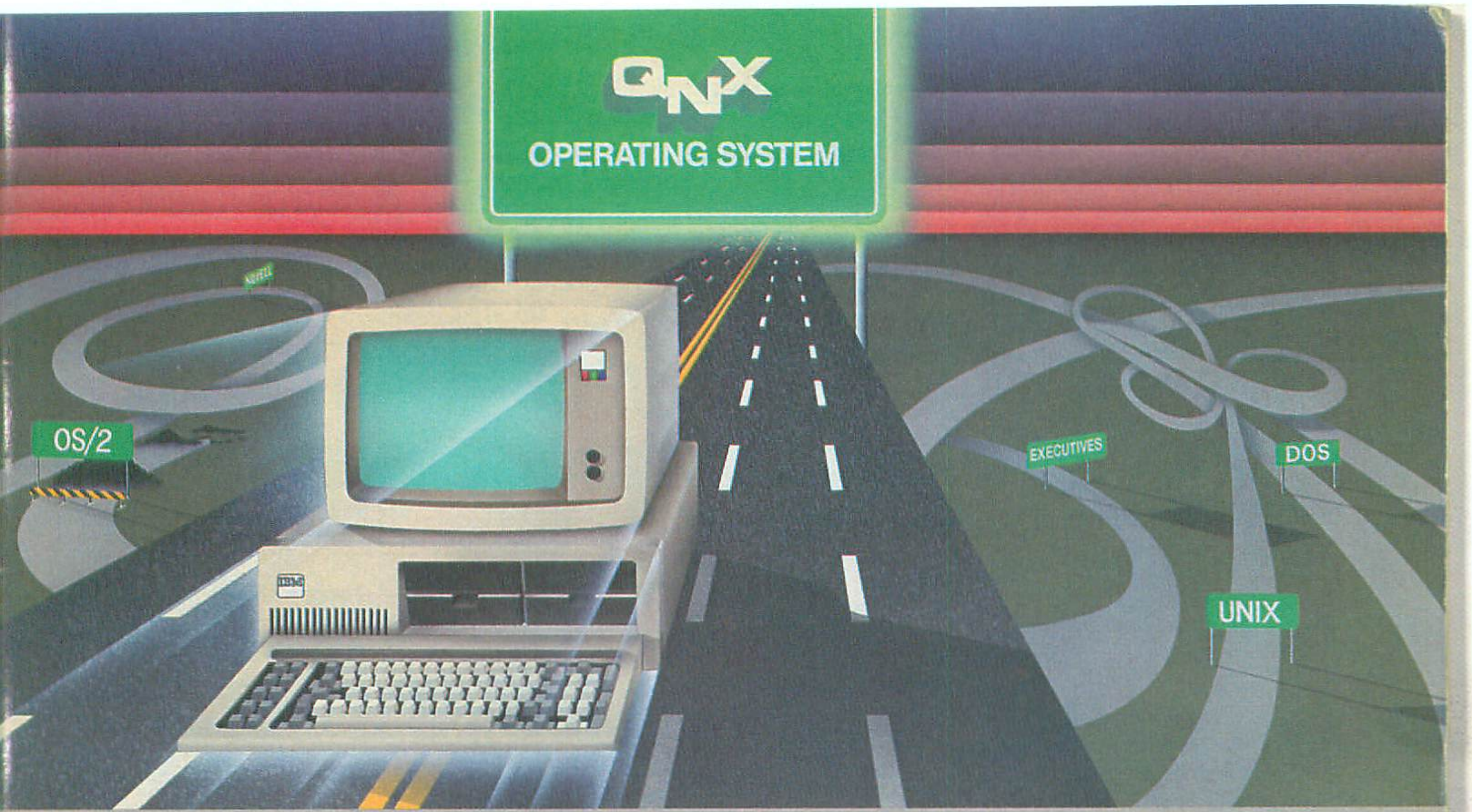
Finite State Machines in Practice

Ray Duncan on Memory Emulators

Debugging With or Without an ICE



HUGHES
LHX
MOTOROLA
LOCKHEED
DUPONT
BOEING
TRW
GE



REAL TIME

Speed without compromise.

QNX® DELIVERS QNX delivers the speed of a dedicated real-time executive as well as multi-tasking, integrated networking and a multi-user development environment as rich and powerful as UNIX.

SPEED The tightly coded QNX kernel performs 3200 task switches/second on an AT, with full pre-emptive prioritized scheduling.

TASK COMMUNICATION QNX is based on a message-passing architecture, radically more innovative than PC-DOS, UNIX, or OS/2. User tasks and system tasks use the same messaging interface. This results in a single unified environment.

INTEGRATED NETWORKING On the QNX network, any task can send messages to any other task anywhere on the network. This direct communication is not available on other networks. The resultant "feel" of the QNX network is that of a homogeneous, tightly connected array of computers, rather than a collection of computing islands strung together on a network with comparatively limited functionality.

DEVELOPMENT ENVIRONMENT QNX comes with a rich set of utilities including a powerful full-screen editor, C compiler, symbolic debugger and multiple full-screen windows.

RUNTIME ENVIRONMENT QNX architecture is modular not monolithic. The system consists of a set of tasks that provide services. Software developers can easily write tasks that add services to suit their specific application needs. It is straightforward to write tasks that interface to hardware through interrupts, I/O ports, DMA and dual-ported memory.

TECHNICAL SUPPORT Technical support is provided free of charge, and updates can be downloaded 24 hours/day from our online BBS.

QNX is now installed at over 60,000 sites in North America and Europe for manufacturing, process control, process monitoring, point-of-sale and many other applications.

Eliminate compromises in your real-time applications. Call for details today.

THE ONLY MULTI-USER, MULTI-TASKING, NETWORKING, REAL-TIME OPERATING SYSTEM FOR THE IBM PC, AT, PS/2, THE HP VECTRA, AND COMPATIBLES.

Multi-User	10 (32) serial terminals per PC (AT).	**C Compiler**	Standard Kernighan and Ritchie.
Multi-Tasking	64 (150) tasks per PC (AT).	**Flexibility**	Single PC, networked PC's, single PC with terminals, networked PC's with terminals. No central servers. Full sharing of disks, devices and CPU's.
Networking	2.5 Megabit token passing. 255 PC's and/or AT's per network. 10,000 tasks per network. Thousands of users per network.		
Real Time	3,200 task switches/sec (AT).	**PC-DOS**	PC-DOS runs as a QNX task.
Message Passing	Fast intertask communication between tasks on any machine.	**Cost**	From US $450. Runtime pricing available.

QNX

For further information or a free demonstration diskette, please telephone (613) 591-0931.

Quantum Software Systems Ltd. • Kanata South Business Park • 175 Terrence Matthews Crescent • Kanata, Ontario, Canada • K2M 1W8

QNX is a registered trademark of Quantum Software Systems Ltd.

CIRCLE #202 ON READER SERVICE CARD

UNIX is a registered trademark of AT & T Bell Labs. IBM, PC, AT, XT and PS/2, PC-DOS and OS/2 are trademarks of International Business Machines. HP and Vectra are registered trademarks of Hewlett-Packard Company.

Automate the critical task of Configuration Management with easy to use and highly flexible tools from POLYTRON. You will discover why thousands of programmers and managers at the leading software, aerospace, manufacturing and service companies use the POLYTRON Version Control System (PVCS™) and PolyMake™ to control the revisions and versions of source code and automate the rebuilding process with unequalled power and precision. PVCS and PolyMake can be used independently or together.

> “ *In terms of features, PVCS provides everything necessary to a large multi-programmer project — more than any other package reviewed . . . all aspects of operation can be customized for specific project needs.* ”
>
> *PC Tech Journal*

Unmatched Capabilities

- Storage & Retrieval of Multiple Revisions of Source & Binary Code
- Maintenance of a Complete History of Changes
- Source Code Access Control
- Optional Merging of Simultaneous Changes
- Release and Configuration Control
- Project Activity Reports
- Management Reports
- Control of Separate Lines of Development (Branching)

For Simple & Complex Projects

Automatically rebuild and maintain simple or highly complex projects consisting of thousands of modules, multiple directories & disks, and geographically dispersed development locations.

Multiple Platform Development

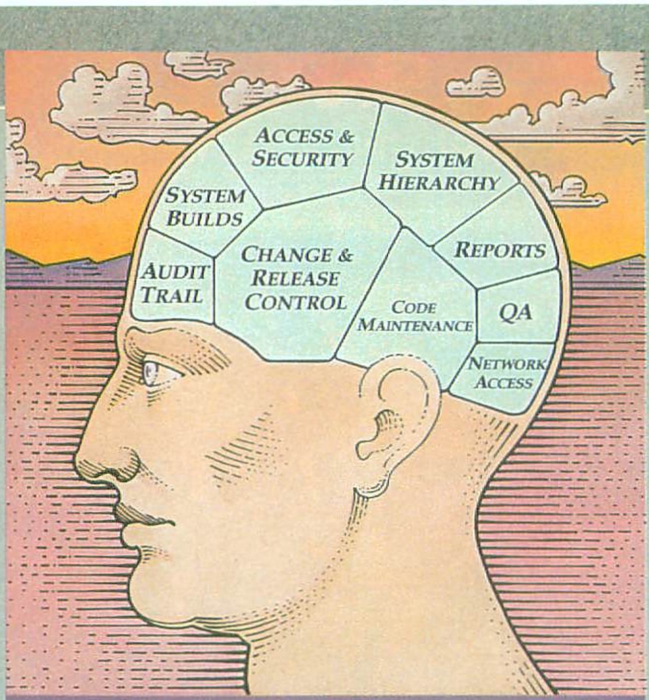
If your projects are developed in a multiple operating system environment, or will be ported to run on another OS in the future, PVCS and PolyMake will make your job easier. The PVCS archive files (logfiles) and the command interfaces are exactly the same across operating systems. The same PolyMake makefiles can run unchanged on the different operating systems.

Supports ANY Language

PVCS maintains individual archives of all project components in your system — source code modules, data files, documentation and even object code. The “source documents” can be written in any language or multiple languages. PolyMake is also language independent.

Call About Our OEM/VAR Strategic Alliance Program

MS-DOS OS/2 Macintosh



CONFIGURATION MANAGEMENT MADE SIMPLE

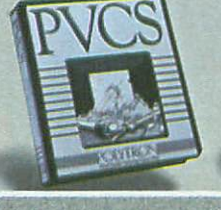
Fast Retrieval of Revisions

PVCS uses “reverse delta storage” which saves disk space and speeds retrieval of the latest versions of any module or an entire system. A delta is the set of differences between any revision and the previous revision. Differences are automatically detected and stored when programmers “check in” a file.

A Practical Necessity for LANs

While important for single-programmer projects, PVCS is absolutely essential for multiple-programmer projects and LAN-based development efforts. In a LAN environment, source code modules are simply too easy to change. Because any change to any module can have major ramifications, coordinating and keeping a record of changes is critical. Project leaders can determine on a module-by-module basis, which programmers can access or modify source files, libraries, object code or other files. Levels of security can be tailored to meet the needs of nearly every project. PVCS works on all major LANs and networks, including networks with multiple computer types.

The Leading Change Management System



The World's Best Selling Build Utility

> “ *PVCS has helped us maintain nearly 90 programs and utilities. Without it we would not have the quality of our new release of NetWare.* ”
>
> *Jonathan Richey*
> *Director of Product Development*
> ***Novell***

Adopt PVCS & PolyMake On Existing Projects

You can obtain the benefits of configuration management for your current project without disrupting development, regardless of how long your project has been under way. You can build PVCS archives from revisions stored in your present archives or simply adopt PVCS from the current date.

PolyMake Works With PVCS

PolyMake understands the structure of PVCS logfiles and is able to correctly determine the time and date of any module revision. This prevents unnecessary operations that occur when the date and time of the revision archive file itself is used as with other Make utilities.

The Price AND Performance Leader

POLYTRON products are priced on a “Per User” basis. The price per user decreases as you add users. ■ ***MS-DOS, Macintosh MPW:*** **Personal PVCS** (for single programmer projects) $149 for single user. **Corporate PVCS** (has features for larger, more complex projects including unlimited levels of “branching”) $395 for single user. **Network PVCS** (includes file locking and security features for LAN use) $1,284 for 5 users. **PolyMake $149. Network PolyMake** $484 for 5 users. ■ *PVCS and PolyMake are packaged together on OS/2, Sun UNIX and VAX/VMS.* ■ ***OS/2:*** $695 single user, $2,259 for 5 users. ■ ***Sun UNIX:*** $795 single user. $2,584 for 5 users. ■ ***VAX/VMS*** any model: $995 single user. $3,233 for 5 users. ■ Call for price quotes.

**OS/2 & Sun UNIX versions available late 1988.*

30 Day Money Back Guarantee

TO ORDER: 1-800-547-4000
Ask for Dept. EMB. Send Checks and P.O.s To:
POLYTRON Corporation, 1700 NW 167th Place,
Beaverton, OR 97006. FAX (503) 645-4576,
TELEX 325800 POLYTRON

VAX/VMS Sun UNIX

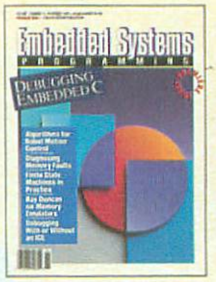
POLYTRON®

High Quality Software Since 1982

CIRCLE #203 ON READER SERVICE CARD

VOL.1NO.1> PREMIERE 1988

EMBEDDED SYSTEMS PROGRAMMING



On the cover: **Debugging takes place on the cool plane where software meets hardware. Photograph by David Bishop.**

Table of Contents

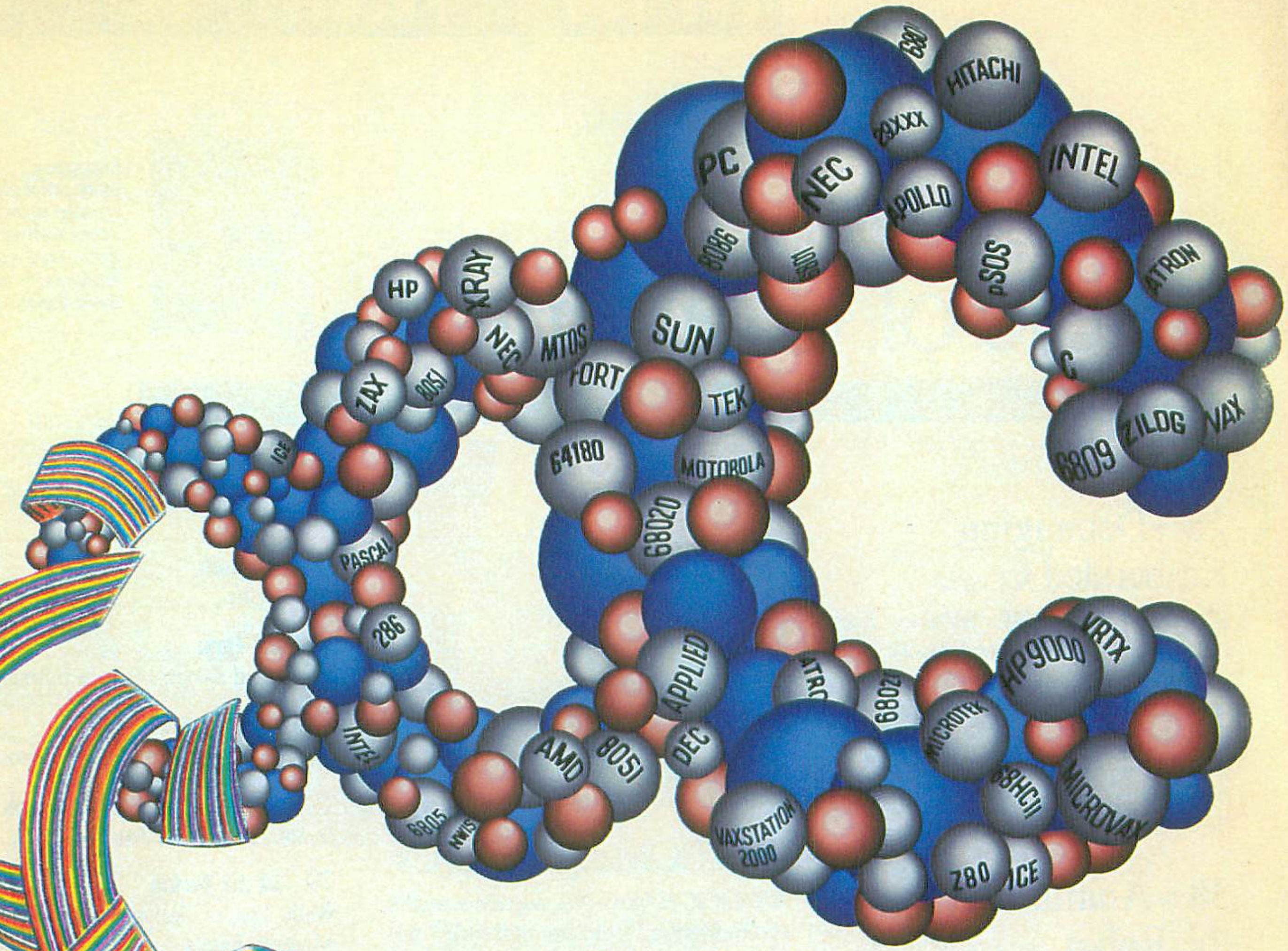
FEATURES

ANIMATING THE ROBOT **PAGE 38**

FINITE STATE MACHINES **PAGE 57**

REVIEWS

COLUMNS+ DEPARTMENTS



MICROTEC CROSS-COMPILERS DEBUGGERS ASSEMBLERS: THE DNA OF SUCCESS

Microtec Research has done a little genetic engineering on the cross-software development industry. Sure, our quality and customer support have become the industry cross-development standard. And what other company has the confidence to actually publish all fixes and ECOs for all products, for everyone to see? But our real legacy to you, our customer, is our commitment to providing *the same* cross-development software from more host processors to more target systems. From pSOS to VRTX to MS-DOS to UNIX to VMS. From assembler to C to Pascal to Fortran. From Motorola to Intel to Hitachi to AMD to Zilog. From IBM to DEC to HP to Apollo to Sun. From crossassemblers to cross compilers. From source debuggers running on hosts that are simulating target systems to the advanced in-circuit emulators from HP, TEK, Atron, Applied Microsystems, Sophia, ZAX and Taiwan's Microtek. All of which means you can spend your valuable time creating new silicon wizardry, rather than learning new cross-development user interfaces. Call Microtec at 800-551-5554 today (408-733-2919 in California). So you can do some genetic engineering on a competitor's physiology tomorrow.

MICROTEC®
RESEARCH

3930 Freedom Circle • Santa Clara, CA 95054 • 800-551-5554 or 408-733-2919 in California

Adv. by TRBA

CIRCLE #204 ON READER SERVICE CARD

Embedded Systems
PROGRAMMING

EDITOR
J.D. Hildebrand

CONSULTING TECHNICAL EDITOR
P.J. Plauger

MANAGING EDITOR
Linda S. Comer

EDITORIAL ASSISTANT
Christine A.T. Dunn

CONTRIBUTING EDITORS
Bruce Bergman, Ray Duncan, Ernest L. Meyer

DESIGNER
Brian Day, Fisher & Day

EXECUTIVE EDITOR
Regina Starr Ridley

PUBLISHER
Donald Pazour

ASSOCIATE PUBLISHER
Ted Bahr, (415) 995-2431

EAST COAST ADVERTISING
ACCOUNT MANAGER
Jo Ben-Atar, (212) 683-9294

ADVERTISING SALES ASSISTANTS
Robert Murphy, (415) 995-2475
Nancy J. Schnarr, (212) 683-9294

SALES/MARKETING ASSISTANT
Carol A. Frederico

ADVERTISING TRAFFIC
COORDINATORS
Dieter Tremp, Kathryn Bruin

DIRECTOR OF PRODUCTION
Andrew A. Mickus

DIRECTOR OF CIRCULATION
Wini D. Ragus

CIRCULATION MARKETING MANAGER
Jerry M. Okabe

ASSISTANT CIRCULATION
MARKETING MANAGER
Kathy L. Shay

mf MILLER FREEMAN publications

CHAIRMAN OF THE BOARD
Lord David R. Stevens

PRESIDENT
Marshall W. Freeman

VICE PRESIDENTS
Charles C. Baake
Leigh M. Freeman
Leonard E. Haas
Barbara M. Hampson
Thomas L. Kemp
Wini D. Ragus

CORPORATE FINANCIAL MANAGER
Charles H. Benz

SECRETARY/TREASURER
Miller Freeman III

ASSISTANT TREASURER
Douglas M. Denny

BPA Consumer Audit Membership
Applied for October 1988

include

by J.D. Hildebrand

Burning In

Launching a magazine, Ted Bahr and I have come to see, is not so different from designing and building an embedded computer system. Many of the challenges you face on the job, we have faced too. Many of the steps you take, we have taken. Your concerns have become our concerns.You may not see many similarities between programming and publishing, but similarities there are.

For us, requirements analysis began more than a year ago when readers of Miller Freeman Publications' *Computer Language, UNIX Review,* and other technical magazines requested more articles about programming embedded microprocessor systems. We investigated and agreed that there was a need for technical information about this increasingly important specialty. Programming magazines covered the topic now and then, and so did electronics magazines. No one devoted consistent monthly coverage to real-time development.

From the analysis came a specification...and a resolve: we would publish a monthly magazine and call it *Embedded Systems Programming.* We would meet the need with the kinds of hands-on, authoritative, technically challenging articles that characterize our other software development titles. And we would devote 100 percent of each issue to embedded systems development.

We got feedback along the way, surveying potential readers, studying other magazines, attending trade shows, and interviewing industry experts. P.J. Plauger's clear vision and quiet certainty were invaluable, and the project took on new life when he agreed to serve as consulting technical editor.

The implementation stage began last summer as we solicited articles, planned a direct-mail campaign, and secured the services of an art direction firm. More than a score of authors coded—whoops, wrote—submissions based on writers' guidelines and preliminary reader surveys. Like implementing an embedded system, launching a magazine requires the specialized contributions of a variety of experts.

The past several weeks, as I write this in mid-October, have been devoted to debugging. We've selected articles, a painful process; many that should have run will have to wait for next issue's consideration. We've edited text, commissioned art, and tested our ad-sales story against market realities.

On October 27 we completed a prototype and turned it over to production. So while this issue may be new to you, our involvement is over. We're moving on to issue two.

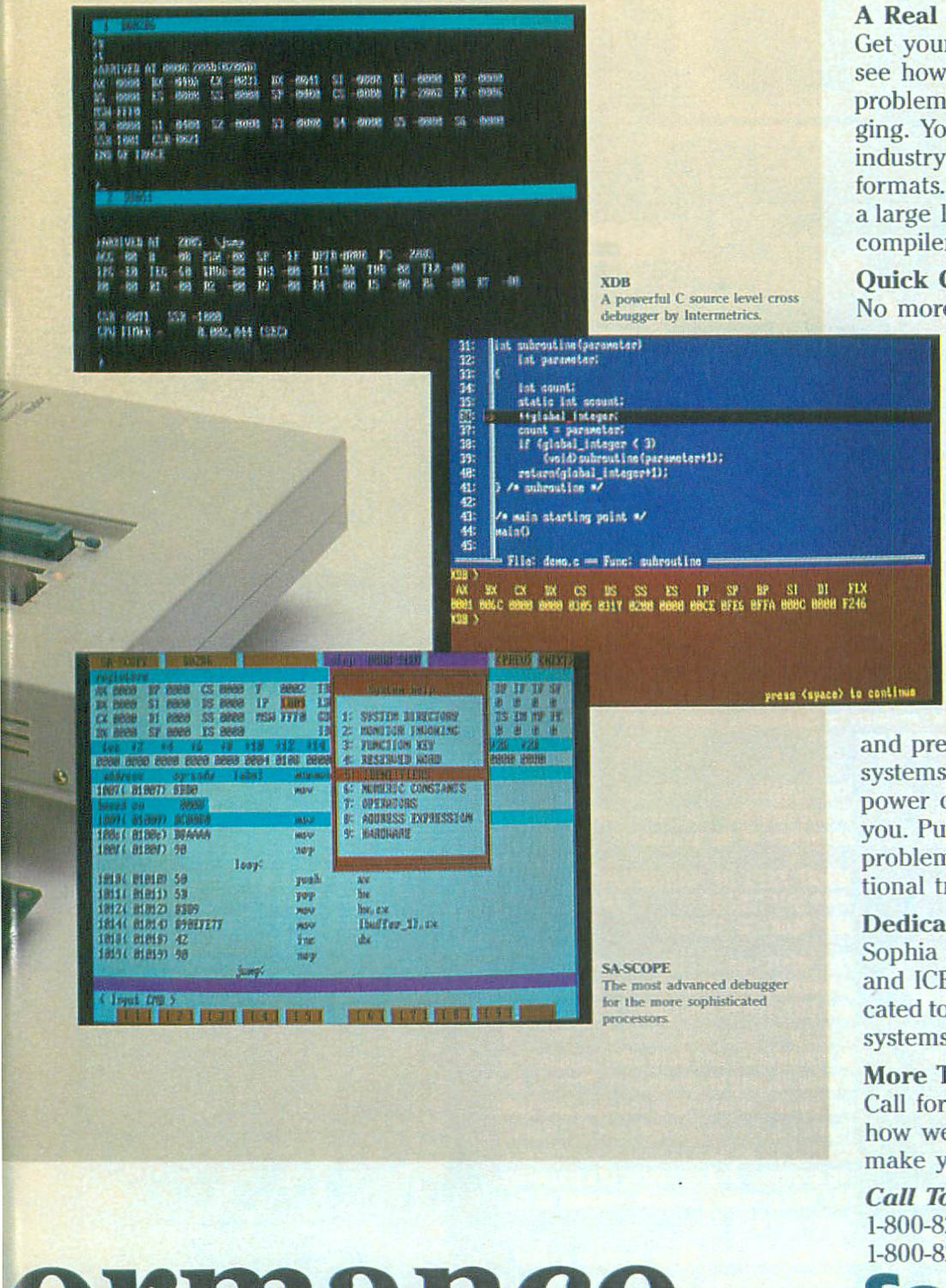
This cycle—analysis, specification, design, implementation, debugging, integration, and delivery of a prototype—is as characteristic of our business as of yours. Our job is to help you do your job better. Drop us a line and let us know how we did with this first effort.

EMBEDDED SYSTEMS PROGRAMMING (ISSN 1040-3272) is published monthly by Miller Freeman Publications, 500 Howard Street, San Francisco, CA 94105. (415) 397-1881. Please direct advertising and editorial inquiries to this address. SUBSCRIPTION RATE for the United States is $47 for 12 issues. Canadian/Mexican orders must be accompanied by payment in U.S. funds with additional postage of $6 per year. All other foreign subscriptions must be prepaid in U.S. funds with additional postage of $15 per year for surface mail or $40 per year for airmail. POSTMASTER: All subscription orders, inquiries, and address changes should be sent to EMBEDDED SYSTEMS PROGRAMMING, P.O. Box 52716, Boulder, Colo. 80322-2716. For quickest service, telephone toll-free 1-800-525-0643 (in Colorado 1-800-447-9300) Please allow four to six weeks for change of address to take effect. Third class postage paid at Long Praine, Minn. EMBEDDED SYSTEMS PROGRAMMING is a registered trademark owned by the parent company, Miller Freeman Publications. All material published in EMBEDDED SYSTEMS PROGRAMMING is copyright ©1988 by Miller Freeman Publications. All rights reserved. Reproduction of material appearing in EMBEDDED SYSTEMS PROGRAMMING is forbidden without permission. EMBEDDED SYSTEMS PROGRAMMING is available on microfilm/fiche from University Microfilms International, 300 N. Zeeb Rd., Ann Arbor, Mich. 48106. (313) 761-4700.

Sophia systems
SA98
Sophia systems
SA98
8051/C51
80286 PROBE

SA98

SADB
Real-time tracing of two separate processors at the same time.



```
ARRIVED AT [illegible]
[illegible]
END OF TRACE

ARRIVED AT  2005  \jump
[illegible]

CPU TIMER -      0.002,044 (SEC)
```

XDB
A powerful C source level cross debugger by Intermetrics.

```
31:  int subroutine(parameter)
32:      int parameter;
33:  {
34:      int count;
35:      static int scount;
36:      ++global_integer;
37:      count = parameter;
38:      if (global_integer < 3)
39:          (void)subroutine(parameter+1);
40:      return(global_integer+1);
41:  } /* subroutine */
42:
43:  /* main starting point */
44:  main()
45:
═══ File: demo.c ══ Func: subroutine ═══
XDB >
 AX   BX   CX   DX   CS   DS   SS   ES   IP   SP   BP   SI   DI   FLX
0001 006C 0000 0000 0105 031Y 0200 0000 00CE 0FE6 0FFA 000C 0000 F246
XDB >

                                          press <space> to continue
```

```
System help
1: SYSTEM DIRECTORY
2: MONITOR INVOKING
3: FUNCTION KEY
4: RESERVED WORD
5: IDENTIFIERS
6: NUMERIC CONSTANTS
7: OPERATORS
8: ADDRESS EXPRESSION
9: HARDWARE

        loop:
1010( 01010) 50          push   ax
1011( 01011) 5B          pop    bx
1012( 01012) 8BD9        mov    bx,cx
1014( 01014) 890EFE7F    mov    [buffer_1],cx
1018( 01018) 42          inc    dx
1019( 01019) 90          nop
        jump:
< Input CMD >
```

SA-SCOPE
The most advanced debugger for the more sophisticated processors.

An MDS/ICE system designed for power, growth, and ease of operation.

Powerful debug commands aren't worth much if they're too difficult to use. We've designed the SA98 to be fully compatible with IBM's PC-AT or XT. Now you can move quickly through the debug process with well-defined, detailed commands.

A Real Productivity Booster

Get your hands on an SA98 and see how fast you can zero-in on problems with full symbolic debugging. You get compatibility with industry standard symbol table formats. And you can pick from a large list of assemblers and compilers.

Quick Change Artistry

No more running up and down command trees to change or modify an emulator condition. You can change commands without stopping the emulator. Define in-depth breakpoints. Call up help menus and files as you need them. And you can go from 8- to 32-bit microprocessors quickly.

Powerful Commands

The SA98 command structure is so powerful and precise, comparing it to other systems is tough on them. Let the power of the SA98 go to work for you. Put it to work on your toughest problems and break out of conventional traps.

Dedicated to MDS/ICE

Sophia Systems is a pro in the MDS and ICE field. Founded and dedicated to supplying the most advanced systems. Keeping the user in mind.

More Than Expected

Call for a demonstration, and see how well-defined solutions can make you more productive.

Call Toll-Free Today
1-800-824-9294 (U.S.)
1-800-824-6706 (CA)

ormance

Sophia systems.

Dedicated to MDS/ICE Support.

CIRCLE #205 ON READER SERVICE CARD

ZAX Presents The Best Way To Develop, Program, Edit, Erase, Compile, Assemble, Debug And Compute

Along with everything else shown here, we offer emulators for the following processors: 8086/88, 186/188, 80286, 80386, 8085, 8048, V20/30, V40/50, 6301, 64180, 6809, 68000, 68020, 68030. And yes, more are on the way.

If you're dissatisfied with the formidable task of trying to assemble a suitable microprocessor development system from different vendors, take heart. Now with a simple phone call, you can receive *complete support* for all your development equipment needs from one supplier—ZAX Corporation!

WHY DOES SINGLE-VENDOR SUPPORT MAKE SENSE?

When you turn your development needs over to ZAX, you're assured that all hardware and software tools were conceived, designed and tested to work together reliably and efficiently. Both with your existing system or as a completely independent development system.

That coordination results in a complete turnkey development system instead of a collection of unmatched components. (Surprising as it seems, this modular approach to design tools still costs less than dedicated systems, yet offers more *flexibility!*) Also, by providing a package instead of a puzzle, you end up conserving another important resource: Time. One phone call. One purchase order. One solid commitment. No headaches.

WHAT TYPE OF HARDWARE AND SOFTWARE TOOLS ARE WE TALKING ABOUT?

ZAX offers you a choice of two different powerful emulation systems with the ICD- and ERX-series emulators. Both can be interfaced to a variety of hosts (from PC to mainframe) and both offer support for a wide variety of processors. There's also our universal interface chassis, the 300i, that's capable of linking our emulators to virtually all host computers and operating systems. And speaking of computers, ZAX can provide you with a model of its own—the BOX-ER.

ZAX can also furnish an array of useful support hardware, such as a line of PLD/EPROM programmers and erasers. Our ZP-series high-speed programmers interface to your PC for a powerful combination. And the ZE-series line of EPROM erasers include everything from an industrial-class, 200-chip model to the world's fastest eraser, the 5-second Quick-E II.

Chances are a broad choice of development software is paramount to your ability to work in a familiar environment. If so, ZAX is still your best source. We offer "C," Pascal, Ada, PL/M and Fortran compliers, assemblers/loaders, symbolic debuggers, source-level debuggers, and helpful menu-driven communications programs to get you up and running, fast.

Call ZAX today and get single-vendor support working for YOU! Our toll-free number is 1-800-421-0982 (800-233-9817 in CA). ZAX Corporation. 2572 White Road, Irvine, California, 92714.

In Europe, call United Kingdom: 0628 476 741, West Germany: 02162-3798-0, France: (03) 956-8142, Italy: (02) 688-2141.

ZAX
Zax Corporation

CIRCLE #206 ON READER SERVICE CARD

MAIL TODAY!

No Risk!

No Obligation!

100% Money-Back Guarantee!

Save 21%!

FREE TRIAL!

Charter Subscription Certificate

SEND ME PROOF! Please start sending these hands-on technical updates on embedded systems programming. I may examine my first issue with no obligation. If I decide to subscribe, I save $10. If I decide not to, I simply write "cancel" on my invoice and owe nothing.

1-Year (12 Issues) Charter Subscription .... ~~$47~~ $37*

SAVE $10

Name____

Company____

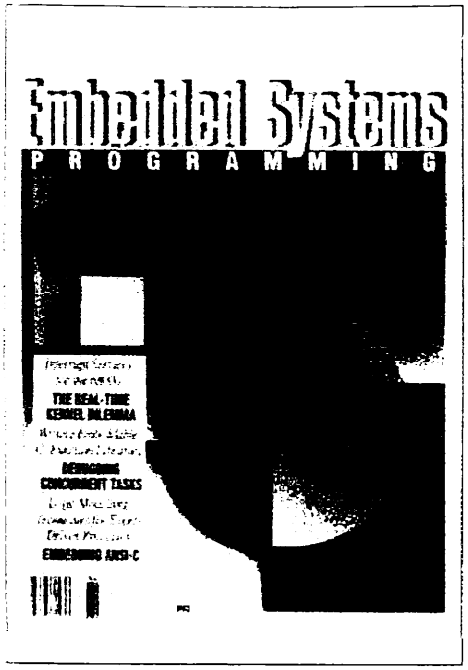
Address________________________________ M/S________

City, State______________________________ Zip________

* 21% discount off regular subscription price.

Please allow up to six weeks for delivery of the first issue. Foreign orders must be prepaid in U.S. funds. Canadian/Mexican orders add $6 per year for surface mail. All other foreign orders add $15 per year for surface mail or $40 for air mail.

48LS



FREE TRIAL!

Charter Subscription Certificate

SEND ME PROOF! Please start sending these hands-on technical updates on embedded systems programming. I may examine my first issue with no obligation. If I decide to subscribe, I save $10. If I decide not to, I simply write "cancel" on my invoice and owe nothing.

1-Year (12 Issues) Charter Subscription .... ~~$47~~ $37*

SAVE $10

Name____

Company____

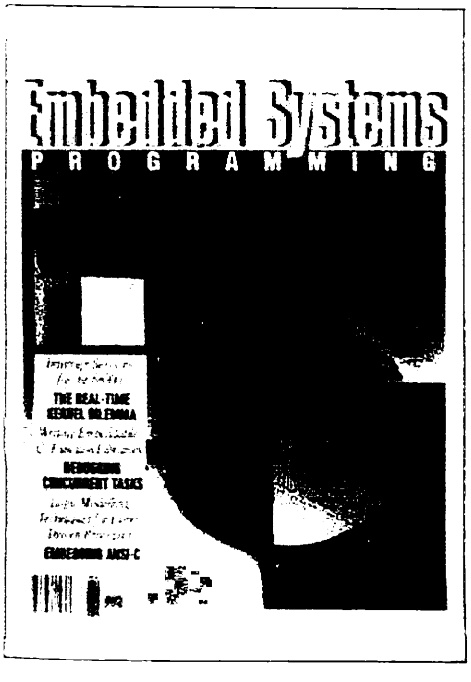
Address________________________________ M/S________

City, State______________________________ Zip________

* 21% discount off regular subscription price.

Please allow up to six weeks for delivery of the first issue. Foreign orders must be prepaid in U.S. funds. Canadian/Mexican orders add $6 per year for surface mail. All other foreign orders add $15 per year for surface mail or $40 for air mail.

48LS



FREE TRIAL!

Charter Subscription Certificate

SEND ME PROOF! Please start sending these hands-on technical updates on embedded systems programming. I may examine my first issue with no obligation. If I decide to subscribe, I save $10. If I decide not to, I simply write "cancel" on my invoice and owe nothing.

1-Year (12 Issues) Charter Subscription .... ~~$47~~ $37*

SAVE $10

Name____

Company____

Address________________________________ M/S________

City, State______________________________ Zip________

* 21% discount off regular subscription price.

Please allow up to six weeks for delivery of the first issue. Foreign orders must be prepaid in U.S. funds. Canadian/Mexican orders add $6 per year for surface mail. All other foreign orders add $15 per year for surface mail or $40 for air mail.

48LS

ATTENTION CIRCULATION MANAGER
Subscriber receives NO-RISK SAMPLE ISSUE
and $10 Charter Subscriber discount.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

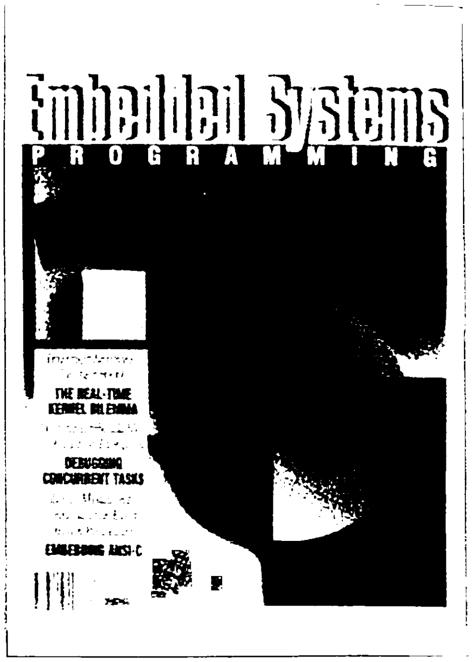
BUSINESS REPLY MAIL

First Class Permit No. 1286 Boulder Colorado

Postage will be paid by the addressee

Embedded Systems
PROGRAMMING

Miller Freeman Publications
P.O. Box 52716
Boulder, CO 80321-2716



ATTENTION CIRCULATION MANAGER
Subscriber receives NO-RISK SAMPLE ISSUE
and $10 Charter Subscriber discount.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

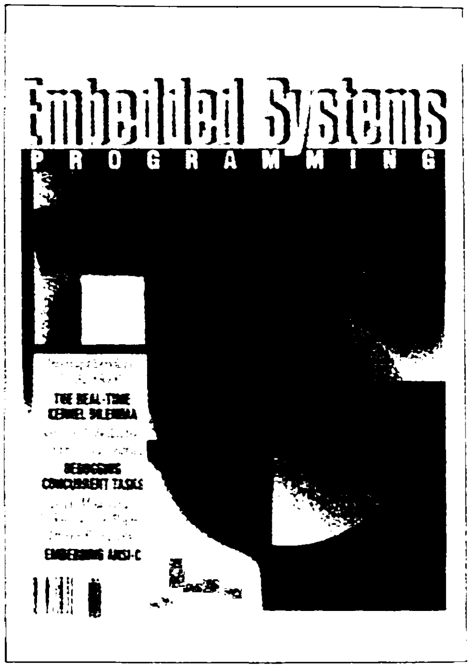
BUSINESS REPLY MAIL

First Class Permit No. 1286 Boulder Colorado

Postage will be paid by the addressee

Embedded Systems
PROGRAMMING

Miller Freeman Publications
P.O. Box 52716
Boulder, CO 80321-2716



ATTENTION CIRCULATION MANAGER
Subscriber receives NO-RISK SAMPLE ISSUE
and $10 Charter Subscriber discount.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

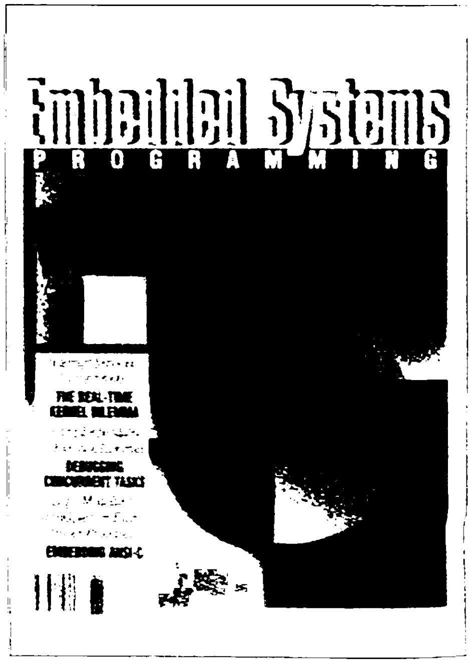
BUSINESS REPLY MAIL

First Class Permit No. 1286 Boulder Colorado

Postage will be paid by the addressee

Embedded Systems
PROGRAMMING

Miller Freeman Publications
P.O. Box 52716
Boulder, CO 80321-2716



by P.J. Plauger

Embedded Systems Programming

When J.D. Hildebrand told me of his plan to produce this new magazine, my immediate reactions were, in order:

Oh boy!

What can I do to help?

Can I please write the editorial introduction to the first issue?

You see, the realization has been growing within me for some time that programming for embedded systems was an important but sadly neglected discipline. I found myself lecturing more and more on the topic. My stack of essays and tutorials on the subject was beginning to teeter. My commercial interests in producing support software were drifting ever more in that direction. The time was clearly overripe to do something.

Nevertheless, I don't think to address a problem by starting a new magazine. The folks at Miller Freeman Publications do. In case you don't know, these are the people who bring you *Computer Language, UNIX Review, AI Expert,* and *Database Programming & Design*—all quality publications in a marketplace not always known for quality, accuracy, or integrity. We programmers can count ourselves lucky that someone went to the trouble of starting a new magazine aimed at embedded systems programming.

Just what is embedded systems programming? Let's dissect the phrase a word at a time.

EMBEDDED

The Oxford English Dictionary defines *embed* as "to fix firmly in a surrounding mass of some solid material." I'm sure that's how you feel at work sometimes, but that's not the basis for the use of *embedded* in this context. Rather, the word is intended to connote that the computer you're programming is stuck in the middle of something else. It's not a general-purpose computer that you can use to run an arbitrary program, but one dedicated to performing a particular task.

Embedded applications are microprocessors wired into the electronics of a host of varied thingamajigs.

A typical embedded application is a microprocessor wired into the electronics of some thingamajig. The micro and its firmware take the place of a fistful of hard-wired logic gates and flip-flops. It may even replace cams, cogs, springs, carburetors, or other bits of mechanical logic. Your microwave oven doubtless contains an embedded computer. Your new car may contain several. Today's laboratory instruments, weapon systems, and yuppie stereos are all built around embedded computers.

An embedded computer need not, however, be either small or intrinsically special-purpose. AT&T has for years used UNIX systems to monitor and control its extensive long-distance trunk network. Even if you *can* log on to one of those systems and compile new C programs or play Hunt the Wumpus, it's still an embedded system. Major airline reservation systems use large IBM mainframes that look for all the world like your standard-issue giant computer centers. However, they're still embedded systems.

Does this mean, then, that the term *embedded* is unselective? Not at all. Whatever their size, embedded computer systems have special requirements:

■ An embedded system typically has one or more special interfaces to nonstandard devices or to devices used in nonstandard ways.

■ An embedded system almost always has to respond to external events in a timely fashion. Often the performance requirements press the state of the art, and failure to perform adequately can cause serious harm or loss.

■ An embedded system frequently consists of two or more processors that must be synchronized.

These requirements are either nonexistent or far less important for general-purpose systems. So you can read *embedded* to mean both "not for general-purpose computing" and "dedicated to an application with special requirements." Either way, you as a programmer or system designer must learn some special skills. You won't necessarily pick up these skills writing programs that run under DOS, UNIX, or MVS.

SYSTEMS

Going back to the OED, we find that a system is "an organized or connected group of objects." A later definition is more to the point: "a group, set, or aggregate of things, natural or artificial, forming a connected or complex whole."

You need a whole system to support an embedded application. That system contains most or all of the parts of a general-purpose computer, to be sure. It also contains special transducers, interface chips, and power supplies. On the abstract side of the house, an embedded system often needs special timing, synchronizing, and error-recovery software. Someone has to understand how all the parts of the system play together and how to balance hardware, software, price, performance, and reliability considerations.

It would be very easy to concentrate exclusively on the CPU and memory chips that constitute an embedded computer. That's the part of an embedded system that most closely resembles the more familiar environment provided by a host operating system. That's the part a programmer of embedded systems often finds the most comfortable. But often it's the programmer who's the "someone" bringing together knowledge of the various system components. If you're that someone, rather than just another tame programmer, you'll be better at your profession and worth more on the job.

Plenty of publications out there tell you about the latest in computer hardware and support electronics. Those publications tend to take for granted that you know what to do with the latest gadgets. If they discuss programming issues at all, they tend to present artificially small examples. Or they present code samples that are laughably inadequate in real-world embedded applications. In short, hardware-oriented publications give short shrift to the software components of an embedded system.

Also, plenty of publications out there tell you about the latest in computer programming languages and support software. Those publications tend to take for granted that you know what to do with the latest software. If they discuss hardware at all, they tend to present gadgets as black boxes that are as easy to use as a library subroutine. Or they present circuits that are laughably inadequate in real-world embedded applications. In short, software-oriented publications give short shrift to the hardware components of an embedded system.

Embedded Systems Programming assumes that you'll have tough decisions to make regarding both the hardware and the software you bring together to make an embedded application. It strives to teach electronic engineers enough about modern programming and software engineers enough about modern electronic engineering that both can design safe, reliable, cost-effective systems. It aims to fill the gaps where pulses meet bits and blueprints meet data-flow diagrams.

PROGRAMMING

My edition of the OED doesn't include the modern supplement. It still thinks a computer is a person who does calculations for a bank or an insurance company. Nevertheless, I applaud its definition of *program* as "to scheme or plan definitely." That sounds like what I've been doing for a living for the past 25-odd years.

Programming is where the action is. That's where you make up for the inadequacies that creep into hardware designs. That's where you tailor the system to the application.

Even though an embedded system design must be concerned with both hardware and software, in many ways software is the more important concern. Software is the glue that holds all the parts together and makes them behave like a "connected group of objects." Software is the last malleable component you can shape to make a system meet the specs before you ship it. Software is often the repository of system complexity and hence the limiting factor in what you can do with the system.

This is not to say that the hardware is unimportant. Far from it. Were it not for the existence of computer hardware that's speedy, accurate, and ever more inexpensive, computer science would be a discipline of interest to just a handful of theoretical mathematicians. Indeed, one of the reasons we can worry less and less about the hardware components of a system is that the hardware designers and implementers have done their jobs so well. More and more, constructing embedded systems consists of pasting together off-the-shelf components and getting the application-specific part of the software right.

You can view the situation in either light. The fact remains, programming is where the action is. That's where you make up for any inadequacies in hardware design. That's where you tailor the system to the specific application. That's where you provide much of the value added in many of the sophisticated products turned out today.

So this magazine unabashedly calls itself *Embedded Systems Programming.* It confines itself to computer systems used in embedded applications. It deals with entire systems and how hardware and software components interact. It addresses the peculiar needs of the programmer of embedded systems.

That's the charter. The modus operandi is:

■ Present columns and articles aimed at practicing engineers and programmers. The focus is at all times practical.

■ Review products, services, books, and training aids. You won't find articles that are simply self-serving promotionals for commercial products.

■ Provide timely information on new technology and trends. You can expect more than just press releases and product announcements.

There's clearly much information and technology that we can share about programming embedded systems. If you have any personal contributions that you think fit the theme of this magazine, please submit them for consideration.

As for me, I've enjoyed my early involvement with *Embedded Systems Programming.* I plan to contribute to the editorial content of this magazine in various ways over the coming months as long as the editors tolerate my offerings. It's been a pure pleasure to read the numerous technical contributions that have already entered the pipeline. You'll get to share that pleasure in the next few issues.

I've learned a lot about a field I thought I knew well. I expect many of you will too. *Embedded Systems Programming* is the best forum for promoting an important technical discipline that has come along in some time. Please help make it a success.

P.J. Plauger is the consulting technical editor of Embedded Systems Programming. *He has coauthored several popular textbooks with Brian Kernighan, including* The Elements of Programming Style *and* Software Tools in Pascal. *Plauger serves as secretary of X3J11, the ANSI C standardization committee, and as president of Whitesmiths, Ltd.*

NEW! UDB Cross-Debugger

OASYS Solves the Cross-Development Puzzle

Every Piece is in Place

68030

with 68881 and 68851 Support

Oasys offers the complete development solution

Fast, Highly Optimized and Available

68030/20/00 Development Tools with 68881 and 68851 Support

- Cross and Native Compilers
  • C • C++ • Pascal • FORTRAN
- Macro Assembler/Linkers for 68000/10/20/30
- 68000/10/20 Simulator
- Real-Time Operating System Kernels
- Multi-Window Cross Debugger NEW!
- Language-Sensitive Editors
- C Run-time Analysis and Profiling Tools
- Communications/ Download Utilities

Plus over 120 other software development tools including:

- Other Complete Tool-Kits:
  - — **80386**
  - — **Microsoft Cross C** for 8086/286
- **C++:** Object-Oriented C++ Tool-Kit:
  - — Designer C++
  - — C++ Class Libraries for User Interface Development
  - — X-Windows/MS Windows Support
  - — Language Sensitive Editor (C++/EMACS)

TOOL-KITS AVAILABLE FOR:
VAX/VMS/ULTRIX
SUN
APOLLO
GOULD
IBM PC
IBM RT
. . . MANY MORE

OASYS SERVICES:

- *New ports easily arranged*
- *OEM, site and corporate licensing*
- *Training available*

Let us help you solve your puzzle

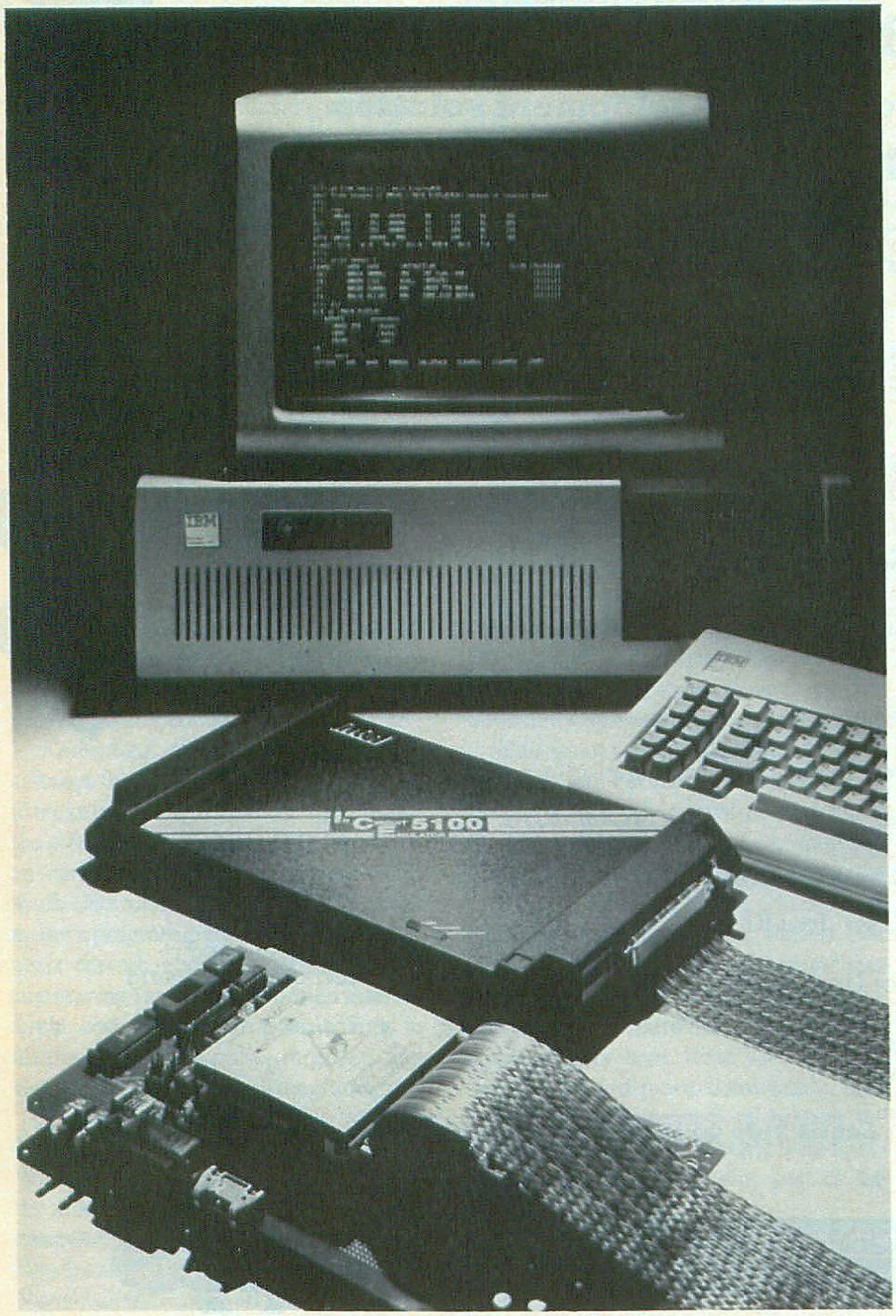
Oasys

230 Second Avenue, Waltham, MA 02154 **(617) 890-7889**

Trademarks are acknowledged to: DEC, UniPress Software, Inc., Microsoft, AT&T, CaseTools, Inc., Glockenspiel, Ltd., and XEL, Inc.

CIRCLE #207 ON READER SERVICE CARD

THE ORIGINAL IN-CIRCUIT EMULATOR FROM INTEL® EMULATES ITS COPYCATS IN ONE RESPECT



It emulates them on low price. In every other respect, Intel's in-circuit emulator sets the standard for speed, reliability, productivity and performance.

Our MCS®-51 emulators and high level language compilers can be run on IBM PC XT, AT or PS/2* computers or compatibles and offer full symbolic debugging.

The original in-circuit emulators were developed by Intel in 1975, so no one can match our wide experience in this technology, or in helping customers be successful in their design. Since you've already decided on an Intel MCS-51 microcontroller, it makes sense for you to buy your emulator and software from the same reliable company with the resources to help you at every phase of your design.

In fact, in all your development work, you'll find that no one can emulate Intel.

Get in touch with your local Intel sales office or authorized distributor to get your hands on the affordable original from Intel. Call 1-800-548-4725 and ask for Literature Department BA00.

$4995 complete

intel®

*IBM PC XT, AT and PS/2 are trademarks of International Business Machines Corp. ICE, MCS, and Intel are trademarks of Intel Corp. ©1988.

CIRCLE #208 ON READER SERVICE CARD

by Ray Duncan

First Thoughts: Memory Emulators

Greetings! You are fortunate to be present at the birth of a new magazine and a new column. Like all births, it's an event rich with hope and potential. I think we shall have many months of growing and coming to know one another. But first, allow me to introduce myself.

I've been programming—and writing about—microcomputers since the days of Imsai and Altair. When I got my first Imsai 8080 with 8 kbytes of RAM, a 70-kbyte ICOM floppy disk, and a Teletype ASR 33, I was euphoric. Up until then, I had been programming on minicomputers where I actually had to share the CPU and disk drives with other users! Since then I've worked with the 8080, Z-80, 6502, 1802, 80x86 family, 680x0 family, 8096/97, 8051/31, 6303, V25, 68HC11, HD64180, and TMS34010.

All these chips are fun (with the exception of the 1802 and 8051/31, which can only be described as endurable). The newer microcontrollers are impressive. The Intel 8096/97, for example, requires only a handful of support components but can blow many of the general-purpose microprocessors of a few years ago right out of the water. The chip is also quite pleasant to program at the assembly level; it has a nice, regular instruction set without any weird gaps (like the missing DEC DPTR on the 8051/31).

During the coming months, this column will cover everything from programming tools to new chips to assembly-language implementations of useful algorithms. (Your suggestions for column topics are welcome.) If I find a particular product especially useful (or especially dreadful) in the course of my work, you'll hear about it. You can choose to exploit or ignore such anecdotal material.

In its simplest form, a ROM/RAM emulator is a small circuit board with some static RAM, a handful of support chips, and a pair of connectors: one to the host and one to the ROM or RAM socket on a single-board computer.

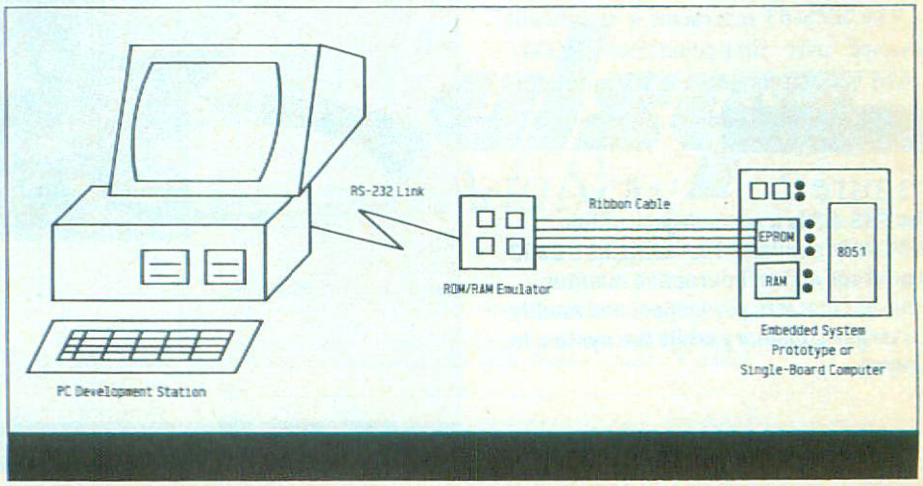
DOWN TO BUSINESS

Let's take a quick look at an as-yet little-known class of products for embedded systems development that are truly invaluable: the so-called ROM/RAM simulators or emulators (neither term is particularly accurate). Though conceptually quite simple, these products can save an astonishing amount of time. And since they cost only a few hundred dollars, they pay for themselves almost immediately.

In its simplest form, a ROM/RAM emulator is a small printed circuit board with some static RAM, a few support chips, and two connectors. The serial- or parallel-port connector goes to an IBM PC; the ribbon cable and DIP connector go to the ROM or RAM socket on a single-board computer. Simple emulators draw the power they need from the target system's socket, while the more elaborate ones may need an external power supply.

The ROM/RAM emulator is essentially a chunk of dual-ported RAM that both the PC and target system can access.

Figure 1
Here's a typical ROM/RAM emulator in use. As far as the target processor knows, the emulator is an EPROM.



When developing a ROMable application using a ROM/RAM emulator, you assemble or compile the code on the PC, then use a small utility program to load the executable image through the serial or parallel port into the emulator. The executable image appears in the target system's memory space as if ROM or EPROM had been plugged in (Figure 1).

The primary advantage of using a ROM/RAM emulator should be obvious. The need for unplugging, erasing, reburning, and resocketing EPROMs is eliminated, speeding up the edit-compile-test cycle enormously. Each time you make a change to your source code, you can run the improved code on the target within seconds after the assembly or compile rather than five or 10 minutes later. There's no need to burn an EPROM until the application has been exhaustively tested and polished.

A more subtle effect, one that you won't notice until you've used a ROM/RAM emulator for a few months, is that the quality of your code really improves. The turnaround time and the hassle of testing new code are so reduced that you'll feel free to optimize and even drastically restructure your code, things you never would have bothered with before.

For the last few months I've been using a second-generation ROM/RAM emulator called the SRS-63 (Figure 2). As far as I know, it's the first of its kind. Designed and built by my friend Klaus Flesch, who runs a hardware/software consulting firm in West Germany, the SRS-63 has a serial interface to the PC. It can emulate a ROM or RAM device of anywhere from 8 to 64 kbytes for the target system and can even (with a special cable) emulate a pair of memory devices for 16-bit micros such as the 8096/97.

The SRS-63 represents a significant advance over first-generation ROM/RAM simulators because it's an intelligent peripheral. Not only does it contain 64 kbytes of emulation memory for the target system, it contains a 6303 microprocessor with its own ROM, RAM, and interactive monitor program. This monitor allows you to inspect and modify the target's memory while the target system is running.

The primary advantage of using an emulator should be obvious. The need for unplugging, erasing, reburning, and resocketing EPROMs is eliminated, speeding up the edit-compile-test cycle enormously. There's no need to burn a ROM until the application has been tested and polished.

When an 8-, 16-, or 32-kbyte device is being emulated, the emulation memory can be divided into pages containing different versions of the same program. A modified executable image can also be uploaded to the PC and saved in a file.

INIT

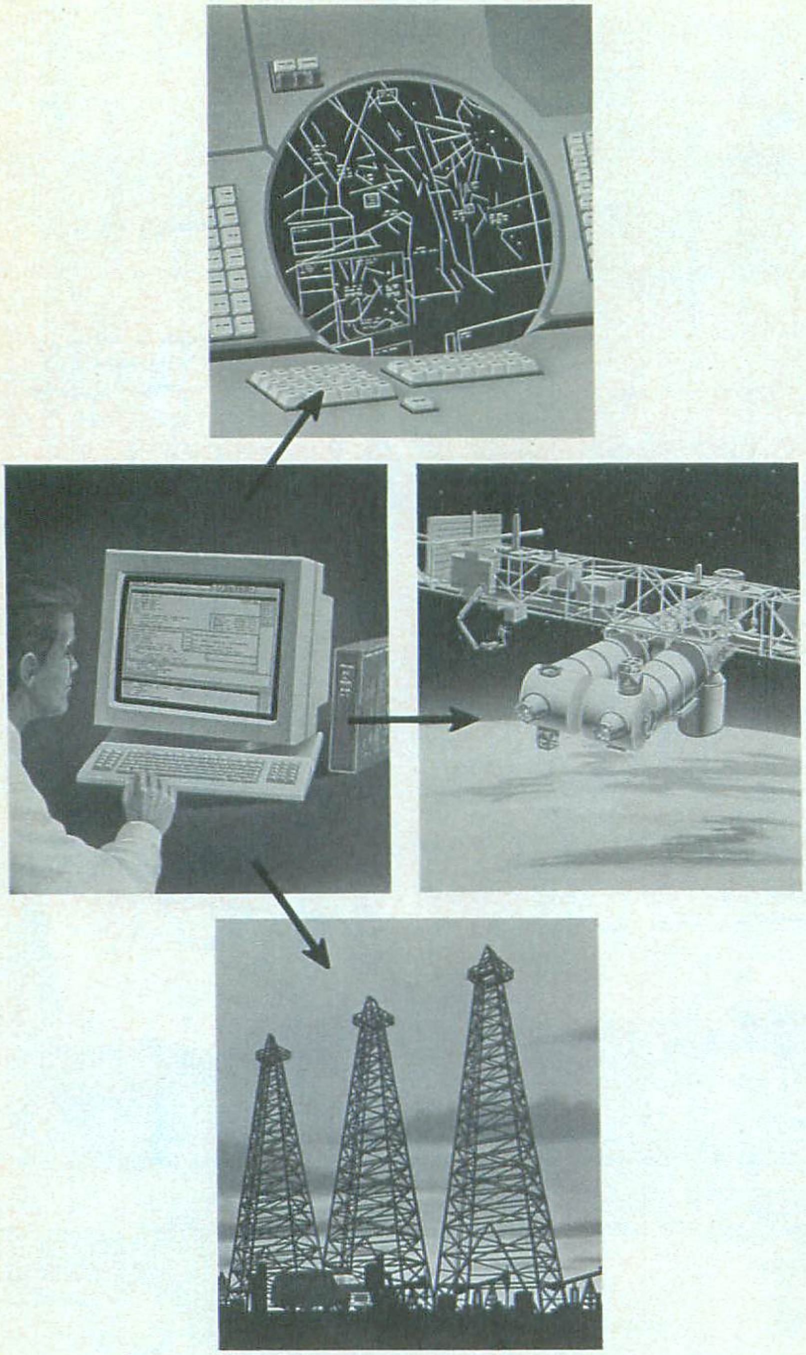
Since I'm cold-booting this column, I've not made any attempt to track down other manufacturers of ROM/RAM emulators, although I know that several exist because I've stumbled across them at Comdex and other trade shows. Companies that build and sell these devices are invited to contact me care of *Embedded Systems Programming*. I'll print a list of available products in a future issue.

Ray Duncan has written columns for Dr. Dobb's Journal, Softalk/PC, *and* PC Magazine. *He's the author of* Advanced MS-DOS Programming *(Redmond, Wash.: Microsoft Press, 1986) and* Advanced OS/2 *(Microsoft Press, 1989). He owns Laboratory Microsystems Inc., Marina del Rey, Calif., a software house specializing in Forth interpreters and compilers.*

Figure 2

The SRS-63 is a second-generation ROM/RAM emulator that contains a 6303 processor and an interactive monitor program that lets you inspect and modify the target's memory while the system is running.

Alsys Ada cross-compilers get you there in no time.



It's time you knew that Alsys, the premier Ada company, offers a range of powerful and flexible cross-compilers for all microprocessors in the Motorola MC680X0 and Intel i80X86 families* to get your applications up and running fast.

Part of the reason for this speed are powerful development tools such as AdaProbe, a source level debugger and program viewer providing facilities to address both the execution properties of a program and its static structure. In addition, there's support for placing program components into ROM, and the Alsys Multi-Library Environment allowing program units to be shared among libraries for team programming.

With Alsys' full line of cross-compilers you'll discover impressive flexibility and power. There's a configurable run-time system giving you full control over tasks, interrupts and all components of your application. The debugger and transfer utility are configurable. Best of all, it's easy to take advantage of all this power because there are only a few files to modify.

When you need to get from here to there without getting lost somewhere in between, use a cross-compiler that knows the shortest route.

alsys
AdaNow!

Check boxes that interest you. ESP 11/88

□ Please send me a Technical Summary on the Motorola MC680X0 cross-compilers.
□ Please send me a Technical Summary on the Intel i80X86 cross-compilers.
□ Please send me a copy of your "Many Facets of Quality" brochure.

Name ____________________

Address ____________________

City ____________ State ____________ Zip ________

Phone (   ) ____________________

Mail to: Alsys, 1432 Main St., Waltham, MA 02154

*The 680X0 cross-compilers are hosted on VAX, SUN 3, Apollo, HP 9000/300, IBM PS/2, PC AT, and Compaq 386. The Intel cross-compiler is hosted on the IBM PC AT and Compaq 386.

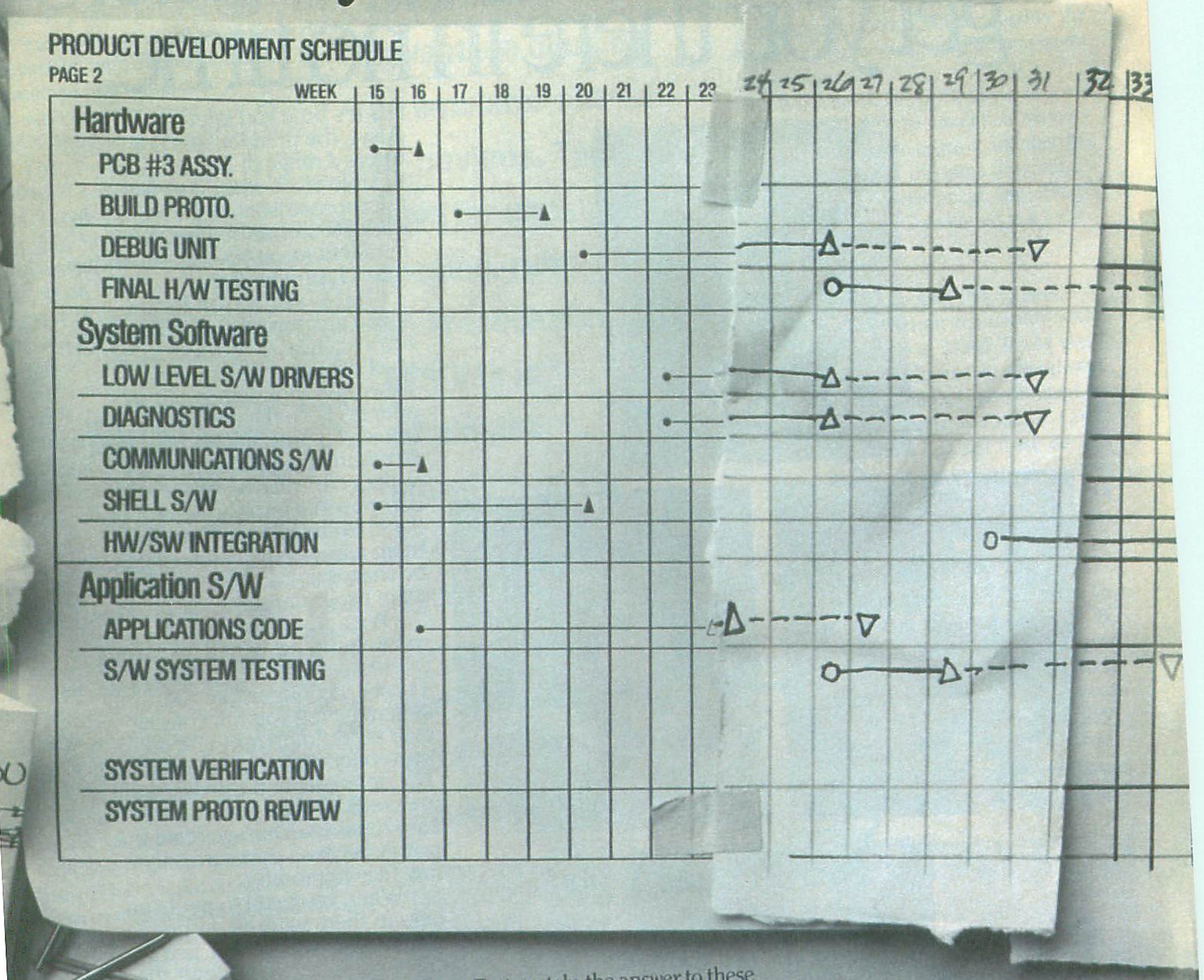
In the US: Alsys, 1432 Main St., Waltham, MA 02154 Tel: (617) 890-0030 Ext #323

In the UK: Alsys, Partridge House, Newtown Road, Henley-on-Thames, Oxon RG9 1EN Tel: 44(491)579090

Elsewhere: Alsys, 29 Avenue de Versailles, 78170 La Celle St. Cloud, France Tel: 33(1) 3918.12.44

CIRCLE #209 ON READER SERVICE CARD

It's your choice.



PRODUCT DEVELOPMENT SCHEDULE
PAGE 2

WEEK 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33

Hardware
PCB #3 ASSY.
BUILD PROTO.
DEBUG UNIT
FINAL H/W TESTING

System Software
LOW LEVEL S/W DRIVERS
DIAGNOSTICS
COMMUNICATIONS S/W
SHELL S/W
HW/SW INTEGRATION

Application S/W
APPLICATIONS CODE
S/W SYSTEM TESTING

SYSTEM VERIFICATION
SYSTEM PROTO REVIEW

Let's face it.

Slipped development schedules and budget overruns can mean lost opportunities. Yet many traps that seriously delay a development schedule are quite complex, especially when they are compounded by problems that arise in cross development work.

Like not knowing whether the errors you are getting from your prototype processor are real. Or losing bugs in the cracks between your development system and the prototype.

Fortunately, the answer to these complex problems is simpler than you might think. Because now Applied Microsystems offers what we call performance packages: complete, fully integrated development solutions, designed to meet your development requirements and to detect even subtle problems quickly.

Performance Packages that Live Up to Their Name.

Each package includes a powerful in-circuit emulator, the only tool that can successfully bridge the gap between host computer and prototype. With features like complex triggering, reliable memory, built-in target diagnostics, I/O simulation, and special interrupt handling.

And to complement the power of our emulators, we provide software tools that work with a variety of platforms and languages.

Whichever package you choose, you're getting the highest performance

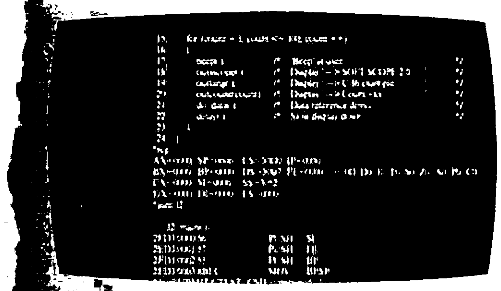
Invest now or pay later.

development tools available.

Source Level Debugging for Intel Microprocessors

Our VALIDATE/Soft-Scope and VALIDATE/Soft-Scope 286 packages are designed for any language producing complete Intel OMF information.

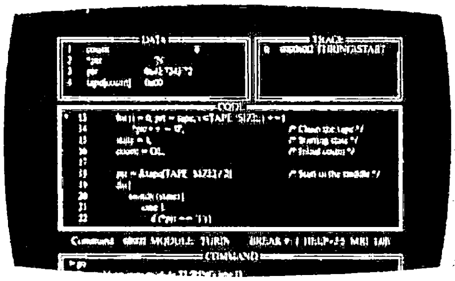
A PC-based, in-circuit source level debugger and simulator are closely coupled with our ES 1800 emulator. You can use commands to examine variables on the fly, check contents of registers, and determine current position in code. And real-time trace is displayed as source level statements, machine instructions or bus cycles.



The packages also include a logic state analyzer probe, and provide up to 2 Megabytes of overlay memory plus full protect mode support for the 80286.

Source Level Debugging for Motorola Microprocessors

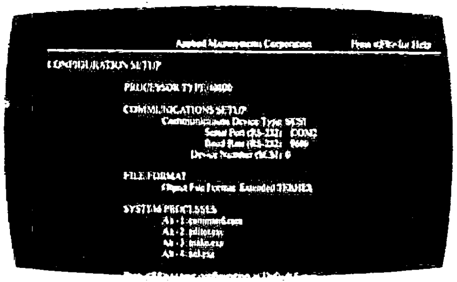
The window-oriented VALIDATE/XEL package combines our XEI source-level debugger, a simulator and the MCC68K compiler with our ES 1800 emulator. The package also includes a logic state analyzer probe and our well-known SCSI interface option, that significantly decreases download time.



In addition to up to 2 Megabytes of overlay memory, you get target control from your source code; powerful "C" language macros for code patching, remote control and simulation of I/O; plus user-definable windows for viewing registers, stacks and variables

High-speed Symbolic Debugging for Intel, Motorola and Zilog Microprocessors

Our VALIDATE/ES DRIVER package includes easy-to-use (menu-driven and remote control) software that smoothly links the host functions to the ES 1800 emulator. This allows the upload and download of programs, symbol tables and command files.



Also included are a logic state analyzer probe; the SCSI option for increasing download speeds by up to 30 times; plus up to 2 Megabytes of overlay memory.

To find out more about 8, 16 or 32-bit development solutions that save money in the long run, write Applied Microsystems Corp., P.O. Box 97002, Redmond, WA 98073-9702. Or call 1-800-426-3925 (In Washington, call 206-882-2000).

In Europe, contact Applied Microsystems Corporation Ltd., Chiltern Court, High Street, Wendover, Aylesbury, Bucks, HP22 6EP, United Kingdom. Call 44-(0)-296-625462.

Applied
Microsystems
Corporation

CIRCLE #210 ON READER SERVICE CARD

12 REASONS WHY

You Should Do Your 80x86 ROM Development With **Microsoft C and C_thru_ROM**

1. You can do it all at your PC
C_thru_ROM enables you to accomplish start-to-finish 80x86 ROM application development in the familiar MS-DOS environment on your PC.

2. You'll have a complete tool set
C_thru_ROM, teamed with your compiler and linker, gives you a complete, full-featured 80x86 ROM development environment.

3. You'll work with the ideal language
C_thru_ROM works with C, which is clearly the language of choice for professional ROM developers.

4. You'll be teamed with the best
C_thru_ROM works hand-in-hand with Microsoft C, the implementation of choice of the C language. And for those who prefer to work in assembly language, the package works with MASM, too.

5. You can stick with your linker
C_thru_ROM doesn't burden you with having to buy, learn and debug a new and unfamiliar linker. It works with the tried and true MS linker you already have.

6. You'll get a great debugger
C_thru_ROM has a comprehensive debugger that allows you to debug in source, assembly, or mixed mode. And it gives excellent visibility through its windows for source, commands, registers, and expressions.

7. You'll be able to debug remotely
C_thru_ROM enables you to sit at your PC and debug your program as it resides and runs on the target system - in source! Working through user-defined communication via a small kernel on the embedded ROM, you can download, modify, execute and debug your application — even access global, local, static, and register variables on the target system.

8. You'll have full code location
C_thru_ROM's flexible locator gives you complete control of code location; providing you full location of segments and classes, and allowing you to place code and data anywhere you'd like in 80x86 memory.

9. You'll get pre-written startup code and ROMable library
C_thru_ROM will save you a lot of time and trouble by giving you pre-written, bugfree startup code (in source) which works with Microsoft-C and the Microsoft-C Library. And C_thru_ROM comes with its own library of useful ROMable functions — an additional timesaving feature.

10. You'll get access to updates
C_thru_ROM's bulletin board is always standing by to offer you the latest updates and rapid bug fixes.

11. You get true ROM support
C_thru_ROM is specifically designed, documented, and supported for the professional ROM developer. Datalight is fully committed to your success as a ROM developer. Our unsurpassed phone support will help you through the roughest storms of ROM development.

12. You can't go wrong
C_thru_ROM comes with the added assurance of a thirty-day, money back guarantee. Full refund. No hassle.

C_thru_ROM............................$495
30-Day Money Back Guarantee

For ordering or additional information
Call Today Toll-Free

1-800-221-6630

And when you call us be sure to ask
for your complimentary copy
of our new guide
"The Essentials of ROM Development"

Datalight

17505 - 68th Avenue N.E., Suite 304
Bothell, Washington 98011 USA
(206) 486-8086

Microsoft and CodeView ar registered trademarks
of the Microsoft Corporation.

by Bruce A. Bergman

Dynamic Interrupt Attachment

Welcome to Programmer's Sourcebook, a column dedicated to providing you with interesting and useful code fragments and programming suggestions aimed at embedded systems development. I'll provide some of these, but our aim is to get *you* involved. This month we'll look at a method for dynamically attaching interrupts to hardware routines, review some pointers for debugging Ada code, and discover a way to reduce stack overhead when calling procedures.

The 1750A is powerful, but its most useful capabilities can also be the most difficult to use effectively.

The MIL-STD-1750A processor is inarguably a powerful chip. However, as some programmers would point out, its most useful capabilities can also be the most difficult to use effectively. One of the more complex elements of this processor is the interrupt-handling scheme.

The 1750A has 16 interrupts, one of which is a form of "software interrupt" or executive call. By carefully planning interrupt use and layout, you can create a simple, powerful interrupt-attachment mechanism that allows a program to dynamically attach an interrupt to a corresponding handler routine at run time.

As discussed in the MIL-STD-1750A ISA manual, each of the 16 interrupts has a corresponding entry in the vector table in low memory. This table contains paired pointers, two for each interrupt, telling the hardware where to store old interrupt information and where to go to handle the current interrupt. The storage location is the linkage pointer (LP) area; the interrupt handler location is the service pointer (SP) area (see Figure 1).

These pairs of LPs and SPs occupy memory locations 20 hex through 3F hex. Each LP and SP must point to another location within page 0 (address state 0) of the 1750A's memory. The LP area is an open area of memory where three words of information for each interrupt—the old interrupt mask, status word, and instruction counter—are stored. Once the interrupt has been handled, this information is used to return control of the processor to the address one past where the interrupt occurred.

The SP area is the heart of the interrupt scheme because it's where the interrupt handlers are defined. As in the LP area, each interrupt has a three-word area where a new interrupt mask, status word, and instruction counter exist.

When an interrupt occurs, the 1750A stores the current (old) state of the processor in the LP area defined for the current interrupt. The processor then loads

Listing 1

1750A assembly routine to attach an interrupt to a handler.

```
:: DYNATT-Dynamic Interrupt Attachment
:: This routine is used to dynamically attach an interrupt or BEX to a
:: routine. It replaces the IM. SW. and IC for the specified interrupt.
:: If an ordinary interrupt should be replaced. R0 should contain zero.
:: If a BEX interrupt should be replaced. R0 should contain a nonzero value.
:: Interrupts are enabled upon exit of this routine.
:: Inputs:
::      R0    0=Attaching interrupt handler. NOT 0=Attaching BEX handler
::      R1    Interrupt/BEX number to attach to (0-15)
::      R2    New interrupt mask (IM)
::      R3    New status word (SW)
::      R4    Handler address (IC)
:: Outputs:
::      None.  Old service pointer information is overwritten with new.
:: Destroys:
::      R1
   REFER  SPTBL       : Should point to start of service pointer areas
   REFER  BEXTBL      : Should point to start of BEX table
   REFER  BEXLP       : Should point to linkage pointer for interrupt 5
DYNATT    EQU    $    : Handle as ordinary interrupt replacement
   LR     R0.R0              : Is this a request for an interrupt?
   BNZ    BEXATT             : NO. replace BEX information
   MISP   R1.3               : YES. index into proper table location
   EFST   R2.SPTBL.R1        : Store IM. SW. and IC at table offset
   BR     LEAVE              : Time to leave
BEXATT    EQU    $    : Handle as BEX replacement
   DST    R2.BEXTBL          : Store IM and SW at table top
   ST     R4.BEXTBL+2.R1     : Store BEX IC at proper location
LEAVE     EQU    $    : Clean up and return
   XIO    R0.ENBL            : Reenable interrupts
   LST    BEXLP              : Continue where left off
   END
```

the new information and begins execution at the address defined in the SP area. When the interrupt handler is done, it usually returns control to the location just past where the interrupt occurred by performing an LST operation on the old state information. By arranging all the LP areas in a consecutive block of memory and the SP areas in the next consecutive block of memory, you can easily identify the LP and SP for any given interrupt.

There is a catch to this idea, though: interrupt 5, the software interrupt mentioned earlier, has the added ability to call any of 16 additional handler addresses using the 1750A BEX instruction. BEX is a special instruction that activates interrupt 5 and dispatches it to one of the handler addresses based on the operand passed to it. It's a novel way of creating software interrupts on the 1750A.

The SP area is slightly different. Instead of 16 additional three-word areas, there's one 18-word area containing the BEX's new interrupt mask, the new status word, and 16 instruction counters in consecutive memory locations. When a BEX instruction is issued, the new information is taken from the top of the table while the

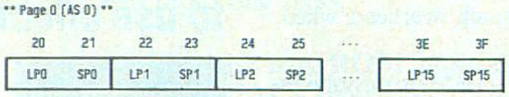
Listing 2

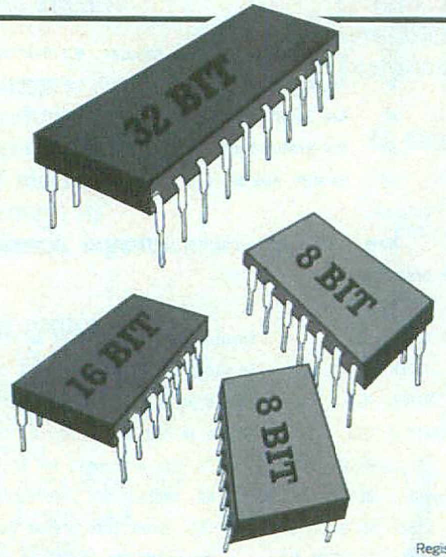
Use of dynamic interrupt attachment routine at the assembly level.

```
:: This code assumes the Dynamic Interrupt Attachment BEX is 0
DYNATT  EQU  0  ; BEX number to use for the DYNATT routine
:: For ordinary interrupt attachment, R0 should contain 0.
:: If BEX interrupt is to be attached, use nonzero value for R0.
  REFER   FLTEXP       ; Floating-point exception handler address
  XORR    R0,R0        ; This is to be an interrupt (rather than BEX) attachment
  LISP    R1,3         ; Attach to floating-point overflow interrupt (3)
  LIM     R2,IMASK3    ; Use interrupt mask IMASK3 (could be any mask)
  LIM     R3,SWORD3    ; Use status word SWORD3 (could be any status)
  LIM     R4,FLTEXP    ; Handler starts at FLTEXP
  BEX     DYNATT       ; Do the attachment now
  END
```

Figure 1

1750A vector table.





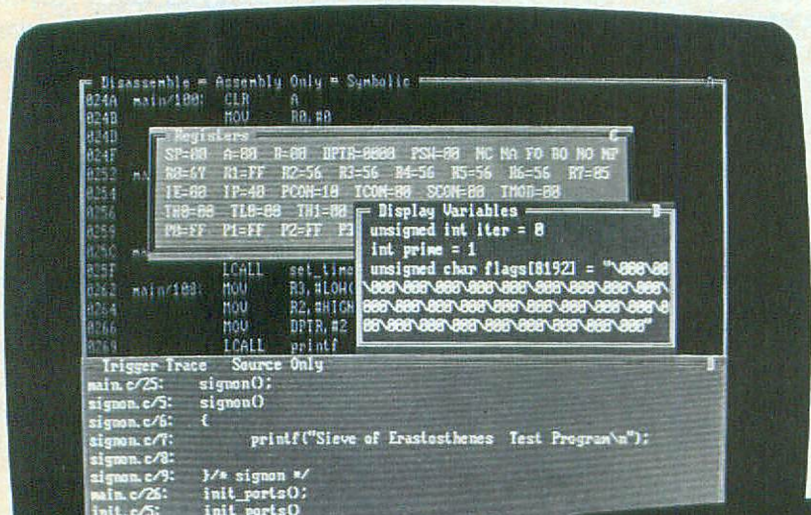
WRITE IN C ? THEN YOU SHOULD DEBUG IN C !

The ultimate in time savings is obtained when you debug your code in the same language it was written. Code development is accelerated as constant program printouts are no longer necessary. All displays of your program, including the real-time trace buffer, are in the form you specify, with options for Source only, Source and assembly or assembly only. Use your favorite C or PL/M compiler with our emulation system and SourceGate™ to enhance productivity of your engineering department. If you are working with different microprocessors, SourceGate provides the same interface for each, so learning curves are almost nonexistent when switching between projects or processors.

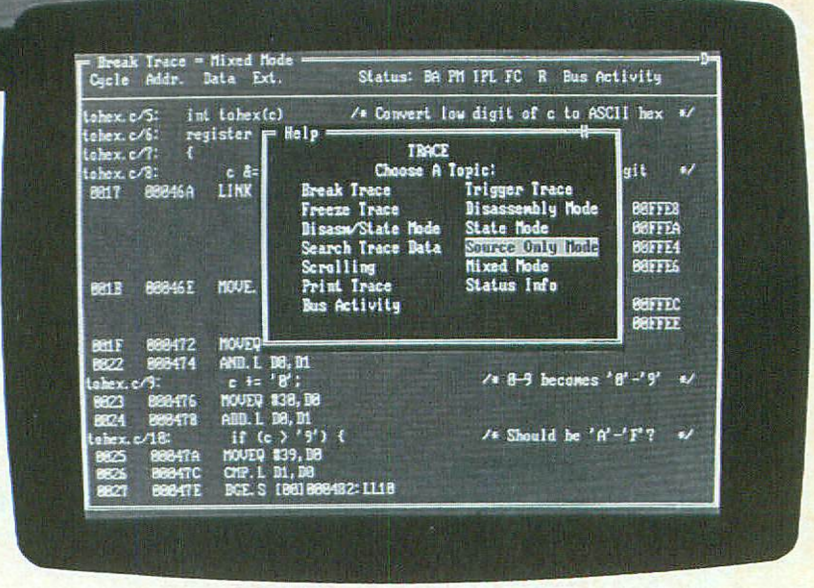
SourceGate was written from the beginning to enhance the power of our 200 Series emulators with an advanced source level debugger. This total integration assures that the emulator capability is utilized and not masked as in afterthought debuggers.

HMI enhances this software capability with the most advanced line of in-circuit emulators on the market today. Current support is available for the 8051 family, 68HC11, 64180/Z180, Z80, 68000 family, 6809 and 8085. SourceGate runs on all IBM PC family computers, Sun Workstations and many Unix systems. For complete details, contact:

Huntsville Microsystems, Inc.
P.O. Box 12415
4040 South Memorial Parkway
Huntsville, AL 35802
(205) 881-6005
TWX: 510-600-8258
FAX: 205-882-6701



```
Disassemble = Assembly Only = Symbolic
024A  main/100:   CLR     A
024B              MOV     R0,#0
024D    Registers
024F    SP=00  A=00  B=00  DPTR=0000  PSW=00  MC MA F0 B0 N0 MP
0252 m  R0=67  R1=FF  R2=56  R3=56  R4=56  R5=56  R6=56  R7=05
0254    IE=00  IP=40  PCON=10  ICON=00  SCON=00  TMOD=00
0256    TH0=00  TL0=00  TH1=00   Display Variables
0259    P0=FF  P1=FF  P2=FF  P3  unsigned int iter = 0
025C m                           int prime = 1
025F              LCALL   set_time unsigned char flags[8192] = "\000\00
0262  main/100:   MOV     R3,#LOH( \000\000\000\000\000\000\000\000\000\
0264              MOV     R2,#HIGH 000\000\000\000\000\000\000\000\000\0
0266              MOV     DPTR,#2  00\000\000\000\000\000\000\000\000"
0269              LCALL   printf
 Trigger Trace   Source Only
main.c/25:    signon();
signon.c/5:   signon()
signon.c/6:   {
signon.c/7:          printf("Sieve of Erastosthenes  Test Program\n");
signon.c/8:
signon.c/9:   }/* signon */
main.c/26:    init_ports();
init.c/5:     init_ports()
```



```
Break Trace = Mixed Mode
Cycle  Addr.  Data  Ext.          Status: BA FM IPL FC  R  Bus Activity

tohex.c/5:   int tohex(c)          /* Convert low digit of c to ASCII hex  */
tohex.c/6:   register   Help
tohex.c/7:   {                          TRACE
tohex.c/8:      c &=               Choose A Topic:                  git    */
0017  00046A  LINK    Break Trace         Trigger Trace
                      Freeze Trace        Disassembly Mode      00FFE8
                      Disasm/State Mode   State Mode            00FFEA
                      Search Trace Data   Source Only Mode      00FFE4
                      Scrolling           Mixed Mode            00FFE6
001E  00046E  MOVE.   Print Trace         Status Info
                      Bus Activity                              00FFEC
                                                                00FFEE
001F  000472  MOVEQ
0022  000474  AND.L D0,D1
tohex.c/9:      c += '0';                    /* 0-9 becomes '0'-'9'  */
0023  000476  MOVEQ #30,D0
0024  000478  ADD.L D0,D1
tohex.c/10:     if (c > '9') {               /* Should be 'A'-'F'?  */
0025  00047A  MOVEQ #39,D0
0026  00047C  CMP.L D1,D0
0027  00047E  BGE.S [00]000482:LL10
```

Supported Processors:

8051 Family	
68HC11	**64180/Z180**
Z80	**68000** Family
6809	**8085**

Sun Workstations is reg. T.M., Sun Microsystems, Inc. IBM is reg. T.M., International Business Machines, Inc. Unix is reg. T.M., Bell Laboratories, Inc.

HMI

CIRCLE #213 ON READER SERVICE CARD

new instruction counter is taken from the appropriate slot in the IC table.

The BEX table should be placed in the next consecutive memory location available. Once these three tables are correctly set up and pointed to in the vector table, it's a simple matter to change interrupt information. The tables *must* remain contiguous for this method to work, however.

Connecting an interrupt handler to an interrupt used to mean you had to know the location of the SP area and move the information into it. The code in Listing 1 allows you to specify the interrupt you wish to attach a routine to, the address of that routine, and a new interrupt mask and status word. This routine needs to know only three things: (1) the location of the start of the SP table (with 16 consecutive SP areas), (2) the location of the start of the BEX table (with all 18 consecutive locations), and (3) the location of the LP area for interrupt 5 (the BEX interrupt).

Since the address of the SP for interrupt 5 must point to the start of the BEX table, it's impossible to change the interrupt handler for interrupt 5 unless you address it as BEX 0. (Note that the routine in Listing 1 assumes BEX 0 is free. If it isn't, use an open BEX interrupt and be sure to call the routine with that BEX number instead.) An example of this routine, assuming the memory layout mentioned, is shown in Listing 2.

Listing 3

Use of elaborated constants to debug algorithms in Ada.

```
package DEBUG_CONSTANTS is
   -- GET_A_VALUE takes the (optional) name of the constant that will
   -- be receiving the initial value as a result of this function.
   function GET_A_VALUE(name : in string := "") return integer;
end DEBUG_CONSTANTS;
with TEXT_IO;
package body DEBUG_CONSTANTS is
   -- GET_A_VALUE takes the (optional) name of the constant that will
   -- be receiving the initial value as a result of this function.
   function GET_A_VALUE(name : in string := "") return integer is
      -- Instantiate integer text i/o
      package IIO is new TEXT_IO.INTEGER_IO(integer);
      value : integer;
   begin
      -- If a name was provided, display it.  Otherwise, just get
      -- the initial value and return it to the constant.
      if (name /= "") then
         TEXT_IO.put("Enter a value for '" & name & "': ");
      else
         TEXT_IO.put("Enter a value: ");
      end if;
      IIO.get(value);
      return value;
   end GET_A_VALUE;
end DEBUG_CONSTANTS;
with DEBUG_CONSTANTS;
with TEXT_IO;
procedure EXAMPLE is
  use DEBUG_CONSTANTS;
  use TEXT_IO;
  -- Get the initial value of a_constant from user
  -- and make b_constant and c_constant dependent on that value.
  a_constant : constant integer := get_a_value("a_constant");
  b_constant : constant integer := a_constant * 4;
  c_constant : constant integer := get_a_value + b_constant;
begin
  put_line("A_constant =" & integer'image(a_constant));
  put_line("B_constant =" & integer'image(b_constant));
  put_line("C_constant =" & integer'image(c_constant));
end EXAMPLE;
```

ELABORATED CONSTANTS

When debugging Ada source code, try to eliminate as many potential algorithmic problems as possible by using constants instead of variables. Using constant values makes debugging easier when changing values is the problem. However, if your program relies on having a wide range of values for a particular variable, you must come up with other solutions—unless you're using Ada, that is. Ada constants have the unique ability to change their initial value and then become unchangeable for the remainder of the program. Take the following Ada constant declaration:

```
a_constant : constant integer := 10;
```

When the resulting program is executed, the value of a_constant is set to 10 during program elaboration. If the initial value of the constant depended upon a value defined elsewhere, you could simply change the declaration to reflect that:

```
a_constant : constant integer :=
 other_value;
```

Note that this kind of declaration is only valid when the base type of the constant is explicitly declared. For instance, the declaration

```
a_constant : constant := other_value;
```

is invalid because the base type of the constant must be part of the declaration.

Taking this one step further, it's possible for the program to ask the user for the initial value of the constant or even calculate the initial value of the constant before using it. Since the result of the constant assignment can be the return value of a function, other_value could very well be a function call that queries the user for the initial value of the constant.

Listing 3 illustrates the use of elaborated constants when the assignment is a calculated value or function.

Assemblers/Simulators/Compilers

Series 3

Series 3 runs under CP/M 80, CP/M 86 and MSDOS. This Series has Full Listing Control, Conditional Assembly & Built in Cross Reference. There is Unlimited Program Size, Number of Symbols and Macros. The Linkers output: Intel Hex, Extended Intel Hex, Tektronix Hex, and Motorola S19, S28 and S37 formats.

Series 4

Series 4 requires at least 640k of memory. These products have all the features of Series 3 plus: 32 Character Labels, User Defined Sections, Local Labels and a Librarian. The Linker outputs Intel Hex, Extended Intel Hex, Tektronix Hex, and Motorola S19, S28 and S37 formats as well as Global, Abbreviated Global, Microtek and Zax Symbol Table Formats. Listings can be relocated to reflect run-time addresses.

Simulator-Debuggers

The Simulator has 16 Breakpoints with optional Counter Field. The Symbol Table is limited only by the amount of memory. Buffers of 256 bytes may be opened for I/O ports. The Simulator-Debuggers run with 512k of memory.

C Compilers

The C Compilers support in-line assembly language and ROMable code, and includes the following: Macro Processor, full Floating Point support, complete Assembler, Linker, Librarian and Libraries. Library Source Code Purchases require a signed license agreement.

	Macro Cross Assemblers			*Simulator-Debuggers*		*C Compilers*		*C Librarys*	*C Library Source*
	Series 3	**Series 4**							
Software Product	**CP/M 80** **CP/M 86**	**MSDOS**	**VMS** **ULTRIX** **SUN**	**MSDOS**	**VMS** **ULTRIX** **SUN**	**MSDOS**	**VMS** **ULTRIX** **SUN**	**All-Systems**	
Super 8		199.50	995.00	149.50	499.50				
Z-8	199.50	199.50	995.00	149.50	499.50				
Z-80	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
Z-280		299.50	1250.00			600.00	2250.00	150.00	250.00
Z-8000	299.50	299.50	1250.00						
1802	199.50	199.50	995.00						
6301	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
64180	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
6501	199.50	199.50	995.00						
6502	199.50	199.50	995.00						
65c02	199.50	199.50	995.00						
65816		299.50	1250.00						
6800,2,8	199.50	199.50	995.00	149.50	499.50				
6801,3	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
6804	199.50	199.50	995.00						
6805	199.50	199.50	995.00						
6809	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
68c11	199.50	199.50	995.00			500.00	2000.00	150.00	250.00
68000,8,10	299.50	299.50	1250.00						
68020		399.50	1500.00			700.00	2500.00	150.00	250.00
8400/c00	199.50	199.50	995.00						
8044/51/52	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
80451	199.50	199.50	995.00			500.00	2000.00	150.00	250.00
80515	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
8080	199.50	199.50	995.00	149.50	499.50				
8085	199.50	199.50	995.00	149.50	499.50				
8086/88	99.50	99.50	1250.00						
8096	199.50	199.50	1250.00						
80186/286	199.50	199.50	1250.00						
80386		299.50	1250.00						
83351	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
8748	199.50	199.50	995.00	149.50	499.50				
740	199.50	199.50	995.00						
Cops 400	199.50	199.50							
F8/3870	199.50	199.50							
NCR32	199.50	199.50							
NEC7500	199.50	199.50							
NSC800	199.50	199.50	995.00	149.50	499.50	500.00	2000.00	150.00	250.00
32000	199.50	199.50							

More formats available

Name ______________________________
(Please Print)

Company ______________________________

Address ______________________________

City ______________ State ________ Zip ________

Phone ______________________________

Product ______________ Operating System ________

Series ______________ Amount $ ______________

Shipping $ ______________ Total $ ______________

Signature ______________________________

Check □ COD (U.S. Only) □

MC/VISA # ______________________________

Expiration Date ______________________________

Educational discount available.

To order, call toll free in U.S. (including HI, PR and VI):

1 800 843-8144

In Colorado: (719) 395-8683

TELEX: 752659/AD

For more details, ask for a free brochure.

(Shipping is $31.00 per unit for overseas orders. Toll Free number does not apply to overseas. 2500AD pays COD charges.)

2500AD SOFTWARE INC
109 Brookdale Avenue
P.O. Box 480
Buena Vista, Colorado 81211

REDUCING STACK OVERHEAD

Unless you have access to a C compiler that runs on a RISC-based computer, you'd be wise to optimize your programs to take advantage of the best parameter-passing scheme possible.

When generating parameter-passing code for an application, compilers try to pass as many parameters in registers as possible. This practice not only improves the access speed for the called routine but reduces the amount of stack space required by the application.

Small objects that aren't composites or aggregates can be passed most easily by placing them directly in a register before the call is made. However, if the number of objects to be passed exceeds the number of registers available or if numerous aggregate objects are to be passed, the compiler must start passing the objects on the stack. In a recursive or reentrant application, heavy use of the stack can have a significant impact.

Successful embedded systems programming requires application of obscure and hard-won knowledge.

[redacted]

So how do you reduce the chances of having objects passed on the stack?

Try placing the parameters in a record or structure and passing the address of the structure instead. The amount of information that must be passed can then be reduced to just one item.

For instance, if you have a procedure declaration in C that looks like this:

```
int junk_call(a, b, c, d, e, f, g, h,
 i, j, k, l, m, n, o, p)
int a, b, c, d, e, f, g, h;
char *i, *j;
long k, l, m, n, o, p;
```

you can reduce the overhead on the stack by placing all the elements of the call into a structure and passing the address of that structure, as follows:

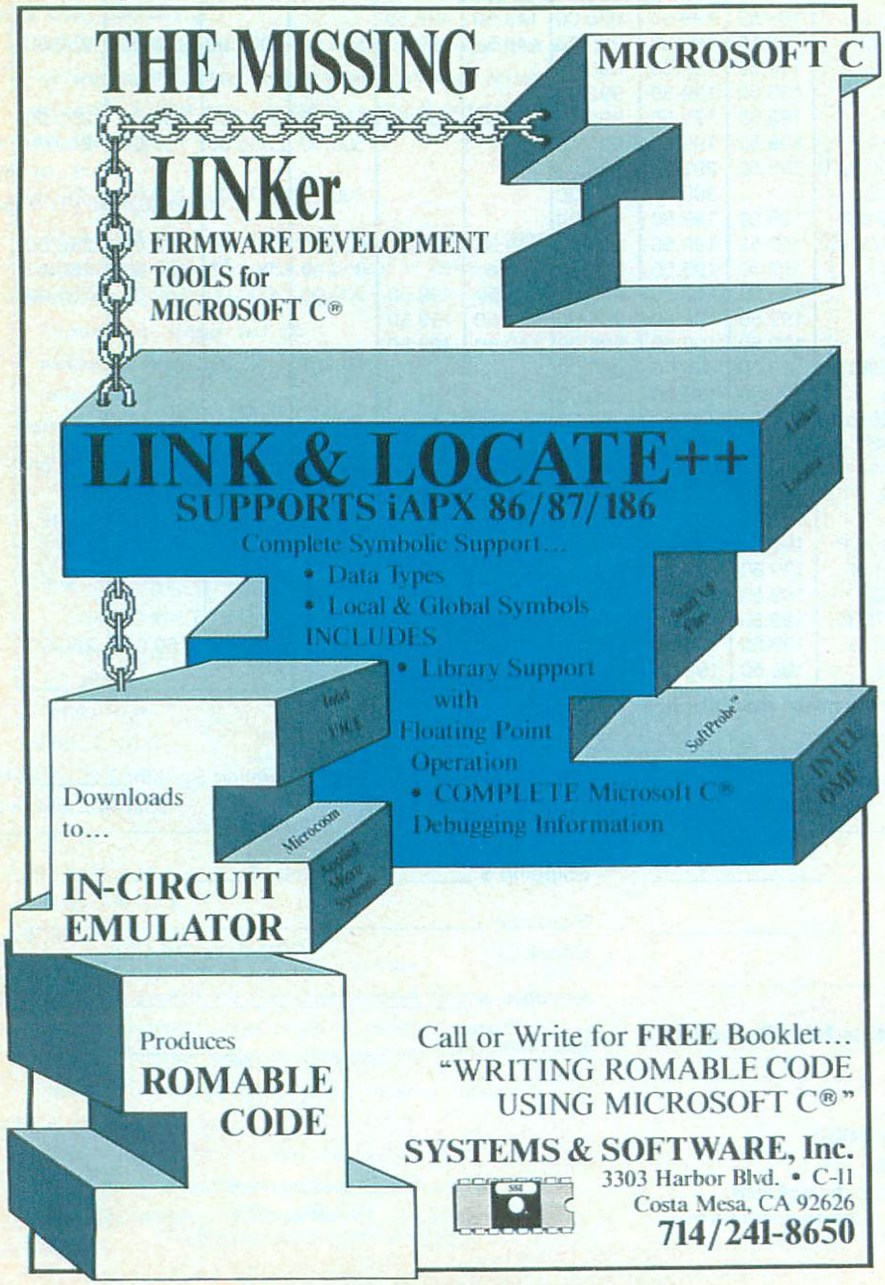
```
struct params_struct {
    int a, b, c, d, e, f, g, h;
    char *i, *j;
    long k, l, m, n, o, p;
} call_params;
int junk_call(&params)
struct params_struct *params;
```

The elements can be manipulated with about the same level of ease, and the advantages of reducing stack space far outweigh the cost of typing a few extra characters.

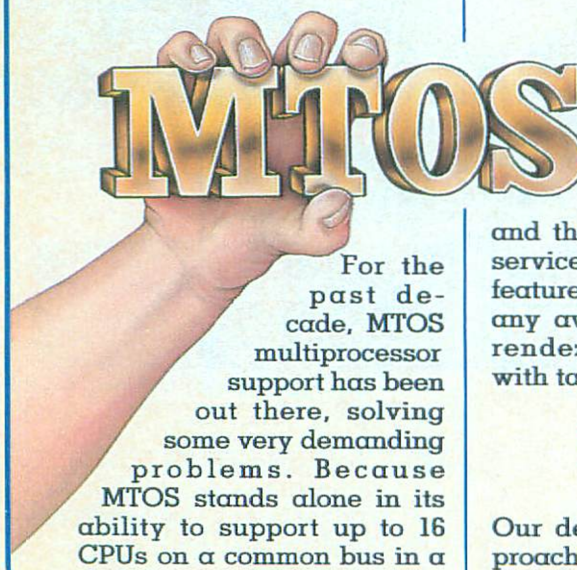
Embedded systems require intimate knowledge of the systems and tools available to the programmer; often this knowledge is obscure and hard-won. If you have programming tips you'd like to share, send them to me care of the editors of *Embedded Systems Programming*, 500 Howard St., San Francisco, Calif. 94105. We prefer tips to be as machine-independent as possible and primarily in C, Ada, Forth, or assembly. By taking advantage of the breadth of knowledge of our readership, we hope to disseminate this information to you every month.

[redacted]

Bruce Bergman is a software engineer at TeleSoft, San Diego, Calif., specializing in Ada compilers targeted to the 1750A.



MTOS-UX, THE SYMMETRICAL, TRANSPARENT MULTIPROCESSOR SYSTEM...BALANCES COMPUTING LOAD... MINIMIZES BUS CONTENTION... AND RENDEZVOUS WITH Ada



For the past decade, MTOS multiprocessor support has been out there, solving some very demanding problems. Because MTOS stands alone in its ability to support up to 16 CPUs on a common bus in a tightly-coupled configuration. MTOS balances the computing load and optimizes the use of available computing resources. And since the number of CPUs is transparent to application code, programmers write tasks as if there's only one CPU. In fact, an application runs in the same way with one or many CPUS — with additional CPUs offering additional throughput.

Ada RUN-TIME PACKAGE

MTOS-UX/Ada is a complete run-time system for TeleSoft™ Ada. It supports the Ada tasking model, all Ada primitives and the full range of MTOS services. The multiprocessor feature allows tasks to run on any available CPU, and to rendezvous transparently with tasks on other CPUs.

HOW MTOS-UX GOT RICH

Our designers took the approach that "given the two alternatives, rich is better than poor." So MTOS-UX has a very complete range of coordination and other facilities: event flags for broadcasting the occurrence of multiple asynchronous events, semaphores, message exchanges, signals, freespace managers, controlled shared variables (an implementation of Per Brinch Hansen's "conditional critical regions"), and much more. And it's all put together in a way that's simple, elegant and highly efficient.

ipi Industrial Programming Inc.

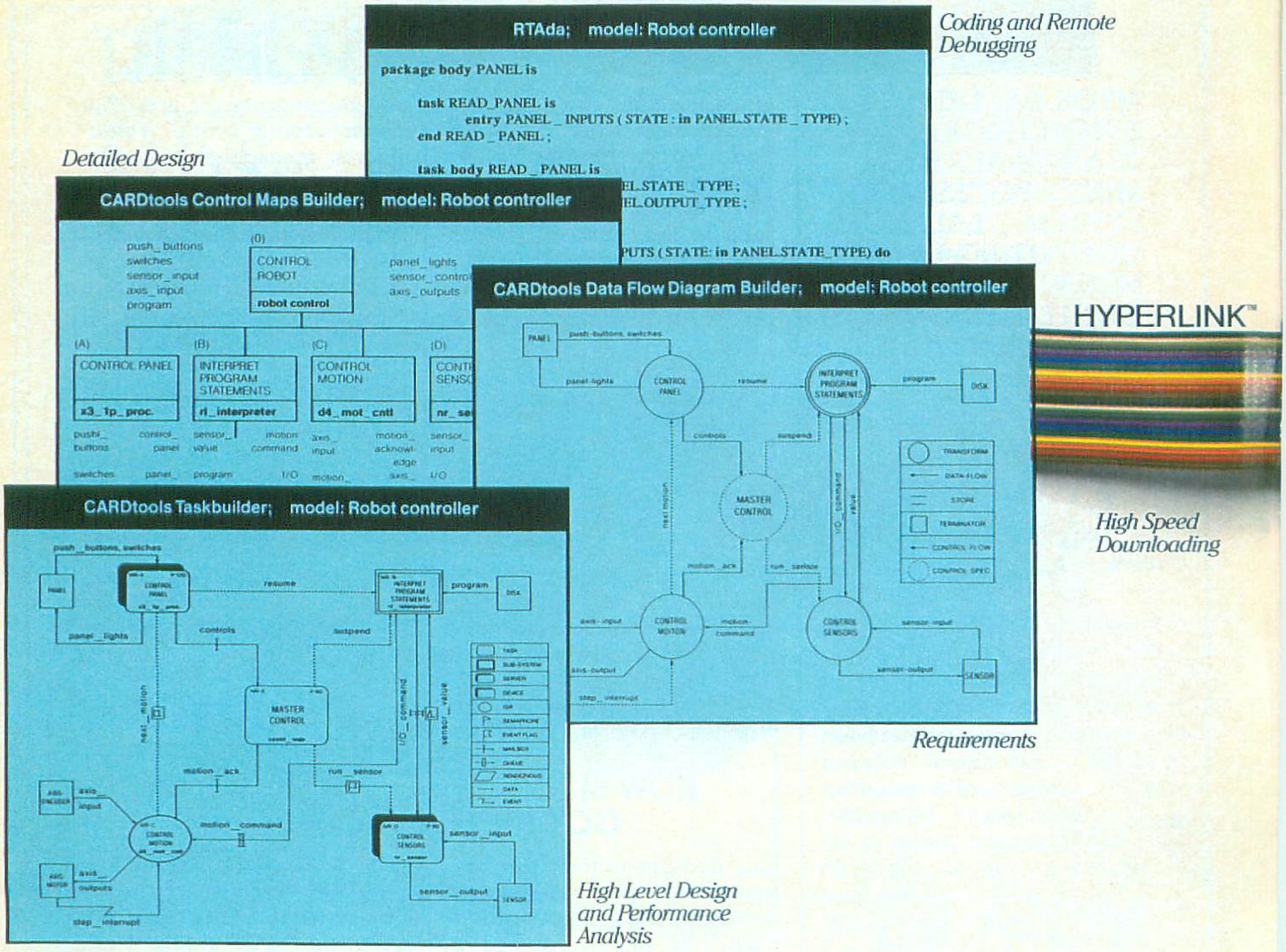
"The Standard Setter"

100 Jericho Quadrangle
Jericho, NY 11753
Phone: (800) 228-MTOS
NY: (516) 938-6600
Fax: (516) 938-6609
Telex: 429808

For detailed information about all MTOS products, or to obtain a free copy of our booklet "On Operating Systems," call **(800) 228-MTOS.**

CIRCLE #216 ON READER SERVICE CARD

Do you have to design in



Coding and Remote Debugging

RTAda; model: Robot controller

```
package body PANEL is

        task READ_PANEL is
                entry PANEL _ INPUTS ( STATE : in PANEL.STATE _ TYPE) ;
        end READ _ PANEL ;

        task body READ _ PANEL is
                                        EL.STATE _ TYPE ;
                                        EL.OUTPUT_TYPE ;


                                        PUTS ( STATE: in PANEL.STATE_TYPE) do
```

Detailed Design

CARDtools Control Maps Builder; model: Robot controller

CARDtools Data Flow Diagram Builder; model: Robot controller

HYPERLINK™

High Speed Downloading

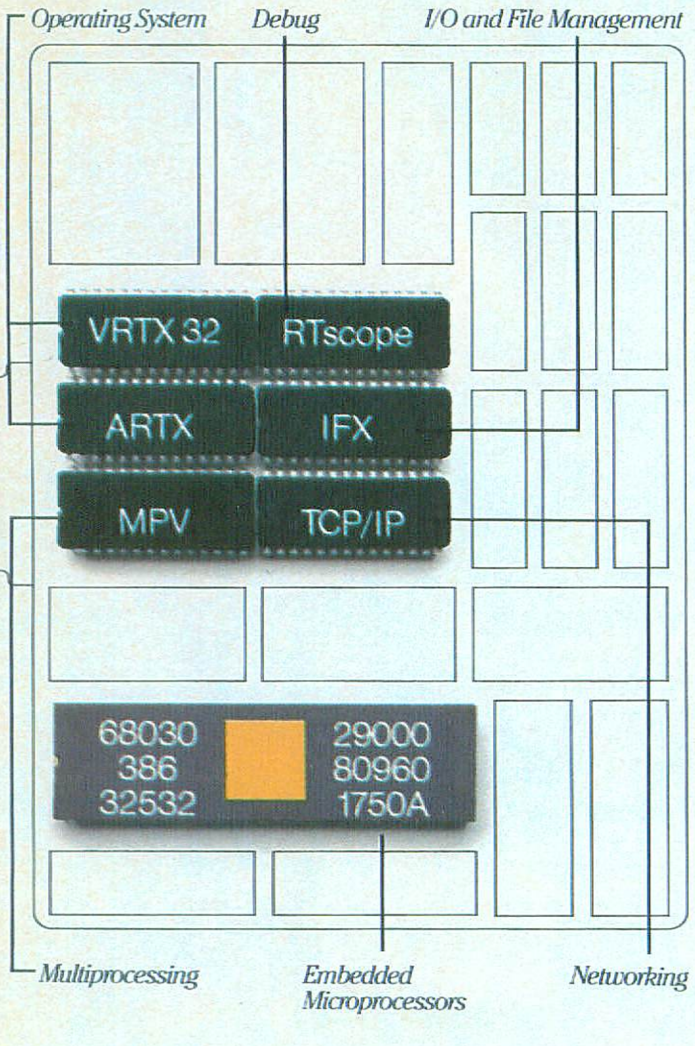
Requirements

CARDtools Taskbuilder; model: Robot controller

High Level Design and Performance Analysis

HOST

what it takes real-time?



TARGET

Successful real-time design for embedded systems takes more than extraordinary helpings of craft and creativity. It also takes the right tools. Which may explain why our CARD (Computer-aided Real-time Design) technology is behind the development of well over fifty million lines of real-time code.

We're not talking about ordinary tools spruced up with a few real-time extensions. CARD technology was created exclusively for the real-time world specifically for integrating run-time and software development environments.

We are talking about tools such as reusable software components like VRTX,® the real-time operating systems standard. Real-time implementations of C and Ada. Automated analysis tools for verifying system performance. Our CARDtools™ product gives you everything you need including specifically tailored design aids to manage software organization and data flow.

And automated documentation so complete, so accurate, it satisfies even the DoD's rigid 2167A specs. Not to mention the demands of our hundreds of exacting commercial customers.

So if you'd like to have what it takes to make your next real-time project run more efficiently call us, toll free, at (800) 228-1249. Or (214) 661-9526 in Texas.

Because you can only be a success in real-time design if you've got the right stuff.

READY
SYSTEMS

CIRCLE #217 ON READER SERVICE CARD

BY ROBIN KNOKE

Debugging Embedded

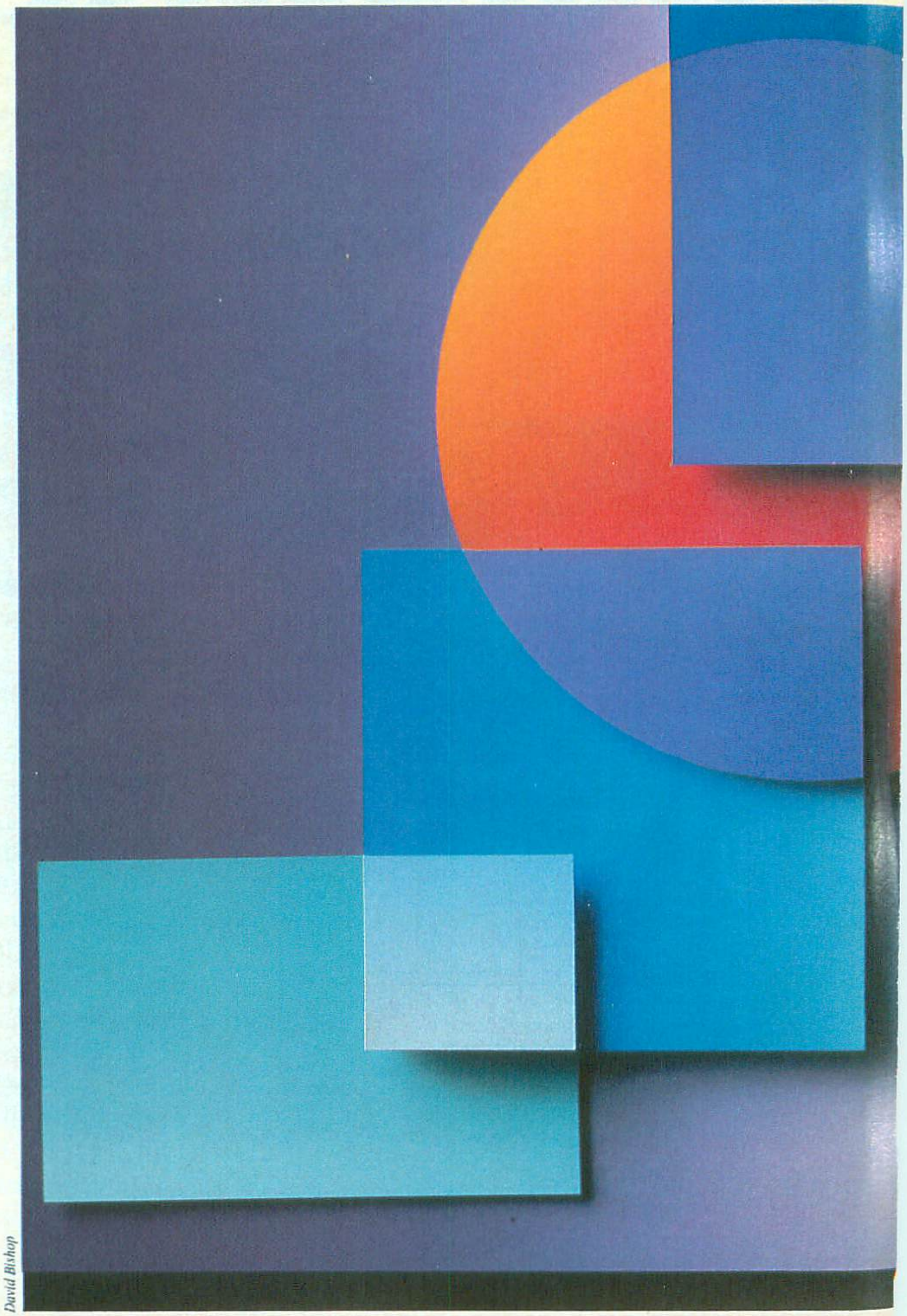
Debugging is one of a series of steps necessary to produce quality software. It consumes much of a programmer's time, yet is one of the least discussed and studied tasks in software development. The process of debugging, as described by Robert Ward in his book *Debugging C* (Que Corp., 1986), involves four phases: testing, stabilization, localization, and correction.

Testing exercises the capabilities of a program by stimulating it with a wide range of input values. First, the program is tested under normal conditions. If it appears to work, its handling of special cases and boundary conditions is tested. Tests should be carefully engineered to force execution of all program branches and thus ensure that every decision node is executed correctly. Any peculiar performance by the program during testing is considered a potential bug and should be investigated.

Stabilization, the second phase, is an attempt to control conditions to the extent that a specific bug can be generated at will. Usually a given set of test conditions will cause a bug to appear, and the bug will remain even when statements are added in the source code. As we'll see, however, certain classes of bugs typical in embedded C programs are difficult to stabilize; any change in the source code or linking process can significantly alter the bugs' behavior or even make them appear to go away.

In the localization stage, the programmer moves in for the kill. Localization involves narrowing the range of possibilities until the bug can be isolated to a specific variable or segment of code. There are three general approaches to this problem.

One approach is to construct a hypothesis to explain how such data might



David Bishop

be created, then modify the experiment to test the validity of the hypothesis. This process of localization employs the scientific method of problem solving, requiring skills quite different from those needed to generate the code in the first place. Modifying the experiment itself will also cause some bugs to behave differently.

Another way to localize a bug is to single-step through the suspect code, watching carefully for the first sign of abnormal behavior. Since the programmer knows what's supposed to happen, the problem can often be pinpointed the first time through the program. The problems with this technique are that it tends to be tedious when the program contains loops or complex structures and that quite often the bug won't manifest itself during single-step execution.

Bugs can also be localized by examining a trace history of the executed code. Microprocessor emulators can be used to capture a trace of the program as it executes, and hardware breakpoints can be used to stop execution where desired. The trace history is then used to reconstruct what happened when the bug occurred.

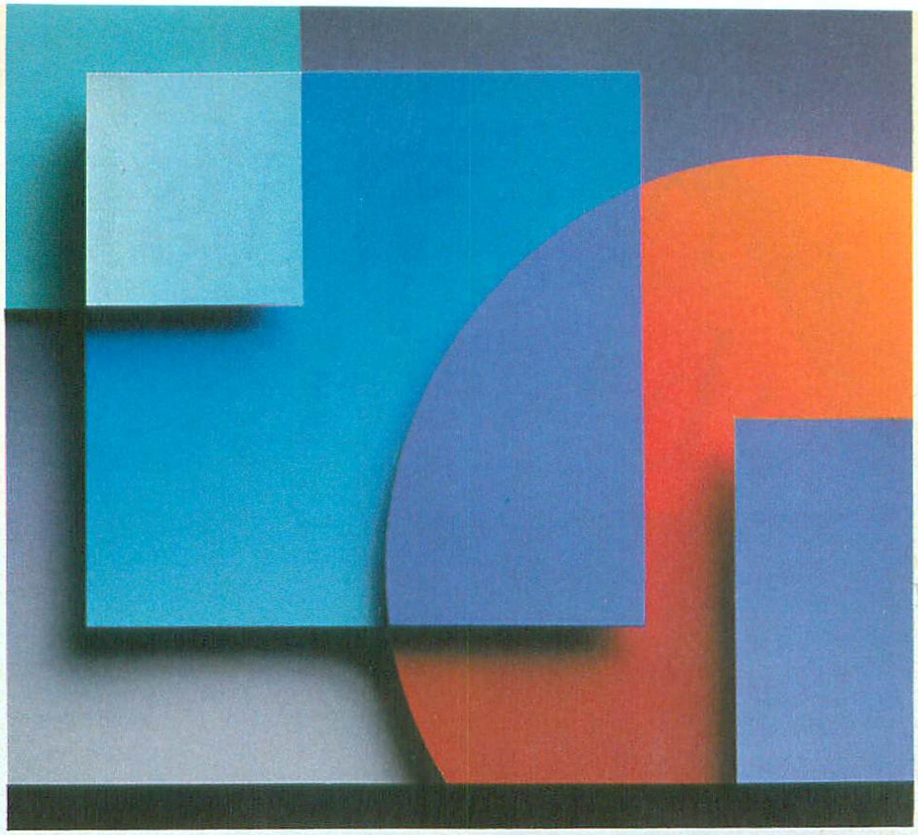
Correction is the final step. After a bug is located, it must be eradicated. Often, correction is straightforward; sometimes, however, a bug reflects a conceptual design flaw. In any event, after the bug is corrected, the process starts over from the testing phase.

Correction is the final step of any debugging exercise. Sometimes bug eradication is straightforward; sometimes a bug reflects a conceptual design flaw.

Debugging a program running under the supervision of an operating system can be quite different from debugging a program in an embedded system. The primary distinction is that the available tools are different. An operating system environment may support native debuggers, where an embedded system may not. Each system has its advantages.

In the native environment, the compiler, source code, debugger, and target pro-

Debugging Embedded C



gram are all together in one place. Cross-debugging—debugging programs in a separate target system—requires a monitor or an emulator. The code must be moved back and forth between the development system and the target. The advantage is that emulators provide hardware specifically designed to facilitate the debugging process.

The following process is typical of a software development effort once the specification for an embedded systems program has been determined:

1. Design the program.
2. Code and edit the program (during this step we may learn how to do step 1.)
3. Compile the program. If compile-time errors occur, localize to find the errors, then return to step 2.
4. Link the program. If link-time errors occur, modify the link commands and relink. If necessary, return to step 2.
5. Test the program. If run-time errors occur, stabilize and localize the bugs, then return to step 2. If necessary, go back to step 1.

To evaluate debugging tools, we need to understand the sources of potential bugs. In *Karel the Robot* (John Wiley & Sons, 1981), Richard Pattis cites four classes of bugs: lexical, syntactic, intent, and execution. Lexical and syntactic bugs are identified by the compiler at compile time; intent and execution bugs are identified by testing the program at run time.

COMPILE-TIME ERRORS

The compiler makes several passes through the code during compilation. The actual number of passes depends on the compiler, but most compilers make three or four. Bugs are usually discovered only in the first two passes. The first pass, made by the preprocessor, expands macros and reads included header files or other source files. In the next pass, the parser and lexical analyzer attempt to understand and produce code from the source statements. Most of the error messages are generated during this second pass.

The following are simple examples of compile-time errors:

- Invalid preprocessor directives (`#page = 7`)
- Illegal operator use (`&a=123;`)
- Illegal symbol or identifier name (`byte j;`)
- Illegal punctuation or character (`j=0:`)
- Illegal language grammar (`if j==0 then j=3;`)
- Incompatible type operation (`*r = r;`)
- Invalid symbol or number (`int 23skidoo;`)

For the most part, these bugs result from typos or errors made by a programmer still learning C and are classified as syntactical bugs. Stabilizing these bugs isn't an issue since the bug recurs each time the file is compiled.

Localization of a compile-time error is usually fairly easy, although occasionally a bug (such as an open comment) may require a little work to localize. Once the bug is localized, it's usually simple to correct.

LINK-TIME ERRORS

The linker's job is to connect other (hopefully tested) modules to the program and build an executable entity. It's possible that one or more of the modules linked to the target program will be either test routines capable of exercising the target program or stub modules simulating a section of code yet to be written.

If link-time errors are detected, most likely the linker couldn't find all the parts

Lexical and syntactic bugs are identified by the compiler; intent and execution bugs are identified by testing the program at run time.

or libraries required to build the final program or couldn't understand the object modules representing the program due to incompatible format. (These problems are usually also reported by the compiler.) In either case, the error is in the commands to the linker or the format of the object module, not bugs in the program. Only rarely, such as when a library function name is misspelled, can a bug cause link errors.

RUN-TIME ERRORS

At this point, the programmer has succeeded in getting the program to compile. This accomplishment doesn't mean that the program is free of typos or incorrectly formed statements; it simply means the compiler didn't detect them.

Now we come to the difficult bugs. Run-time bugs that exist without catastrophic results—the program will run but does the wrong things—are intent errors; those that cause the program to terminate abnormally are execution errors.

An intent error occurs whenever a program runs to normal completion but produces incorrect results. Examples of intent errors are one-liner, typematch, boundary, macro, and design bugs.

The simplest intent error is the one-liner, a syntactically correct statement that has an error in it. These errors are usually caused by an incorrect assumption regarding operator precedence, an incorrect choice of operator, or misplaced or missing punctuation. These errors, which I call "awshitical" bugs (based on mutterings from programmers who have spent considerable time staring right at the erroneous statement without seeing the bug), are a subclass of intent bugs. Here are a few:

```
if (i=1) {...}
```

Dang, we wanted to compare i, not assign to it. The if condition will always be true.

```
if (i==1); {...}
```

Oops, we put a semicolon after the if statement. The if has no effect on the body of statements that follow.

```
while (c = getchar() != 0) {...}
```

We forgot operator precedence; c will be 1 or 0. What we meant to write is ((c=getchar()) != 0).

The simplest intent bug is the one-liner, a correct statement but for misplaced punctuation or an incorrect assumption about operator precedence.

Boundary bugs show up when test inputs are designed to test the boundary (or beyond-boundary) conditions of the program. When dealing with arrays, it's easy to create a boundary bug by forgetting that the 10th element in a 10-element array is actually at array[9]. It's not uncommon for even the most seasoned programmer to occasionally generate invalid array indexes within while and for loops, especially if the loops are at all complex. Out-of-bounds array indexing can also cause viral bugs (described later).

Boundary bugs don't necessarily involve arrays. Any variable has a limited range of values, so tests at (and, if allowed, beyond) these limits should be run. Listing 1 contains two potential boundary bugs.

On machines that implement 16-bit ints, requesting more than 32,767 bytes will produce undesired results due to the signed comparison i<n. Also, if zero bytes are requested, the function can never return an EOF, even if no characters are available. This exception brings up an interesting point: if the programmer can guarantee that the calling function will never ask for zero characters, then this boundary need not be tested. Indeed, by definition the zero-length buffer bug doesn't exist.

A more complex form of bug is the type mismatch, or typematch, bug. It occurs when the programmer attempts to pass arguments to a function but the called function expects arguments of a different type. Although some of the newer compilers will catch this and some lint utilities are designed to look for precisely this type of error, typematch bugs still seem to crop up now and then. Here are two examples:

```
double nbr = 5.0;
int x;
sscanf("123","%d",x);
printf("%f",nbr);
```

In the third line, x should be &x; in the fourth line, the argument type is incorrect. The function argument prototyping in newer compilers may catch these errors for library calls, but programmers can call their own functions and may not have included argument prototypes in their header files.

Macro bugs are errors that are inadvertently caused and camouflaged by macro expansion. When the preprocessor expands macros, it substitutes the macro definition anywhere the macro name appears in the body of the program. The programmer must be aware of what the code will actually look like after the macro expansion. If the macro contains parameters or other macros, there are even more things to consider. In Listing 2, both expansions of the macro will cause intent errors.

In the first expansion, nbr is incre-

Listing 1

Code containing two potential boundary bugs.

```
/* Read n characters from stdin to buffer.
** Return EOF if end of file, otherwise return OK.
*/
int getinput(n,buffer)
int n;
char *buffer;
{
  int i, c;
  for (i=0; i<n; i++) {
    c = fgetc(stdin);
    if (c==EOF)
      return(EOF);
    *buffer++ = c;
  } return(OK);
}
```

mented twice; in the second, the macro expands to (nbr&0x7f)'0'&&nbr&0x7f<='9') ?1:0) and doesn't perform at all as might be expected. (My compiler concludes that the statement is always false and optimizes it to a jump instruction.)

Intent bugs characterized by flaws in the design approach are called design bugs and result from incomplete comprehension of the problem. Some problems are so complex that it's hard to comprehend the entire problem at once. Sometimes the insight needed to solve the problem comes from trying to program it. Design bugs are often the result of some simple oversight by the programmer. Computers are more exact than we humans and sometimes embarrass us with their explicit logic.

In the alphabetical sorting function in Listing 3, tests were run using all uppercase or all lowercase strings and the function worked flawlessly. When uppercase strings were compared with the lowercase the function claimed the uppercase string should come first, regardless of the characters. This wasn't expected.

Design bugs like these can be local to a function or result from the interface of two or more functions. As design errors span a larger scope, they are considered to be integration bugs.

Programs that terminate abnormally contain execution errors. These bugs may be detected by run-time type checks, bound checks, or hardware-fault detection mechanisms in native environments, but in an embedded application they may simply cause the system to crash. Examples of execution bugs are division by zero, running a program with link-time errors, incorrectly implemented interrupts or assembly code, out-of-bounds arrays, and assignment to an invalid (uninitialized) pointer.

Simple execution errors such as division by zero can be easy to stabilize, provided the zero is a direct result of a controlled test case. Otherwise, they can be hard to localize. For example, some processors issue an interrupt when an illegal operation is attempted. If this interrupt wasn't expected, the programmer may not have set a valid vector in the interrupt vector table. In this case, the control flow may go "into the weeds" and the original cause may not be readily apparent.

Many execution bugs are the result of an errant store through an incorrectly initialized pointer or at an out-of-bounds array index. These bugs belong to a very nasty class called viral bugs and can be extremely hard to stabilize and localize. They crop up in C programs because of the unrestricted run-time use of pointers, array indexing, and casting. The effect of a viral bug—corrupted code, data, or stack—is usually not apparent until several (million) instructions later. Even then the infected data might not prove catastrophic but may cause something else to become infected. At any rate, when the bug eventually surfaces, the symptom usually has nothing to do with the original bug.

Ordinarily, uninitialized pointers are more insidious than out-of-bounds arrays. An out-of-bounds array reference usually attacks a stack frame or data area adjacent to the location of the array. An uninitialized pointer can attack anywhere and is very often inconsiderate as to whether it attacks code or data. Furthermore, the pointer will probably contain a different initial value each time the program is run, causing an entirely different symptom each time.

Listing 2

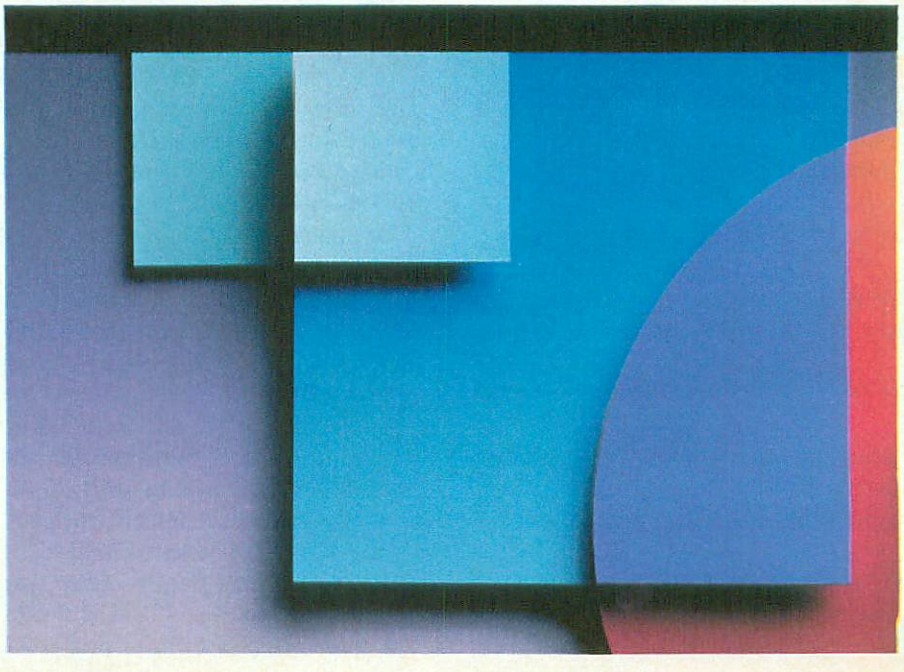
Expansions of this macro will cause intent errors.

```
#define is_ascii_digit(x)  ((x)'0'&&x<='9')?1:0)
func(nbr)
int nbr;
{
  if (is_ascii_digit(++nbr))
    printf("%c ",nbr);
  if (is_ascii_digit(nbr&0x7f))
    printf("%c ", nbr&0x7f);
}
```

Listing 3

An alphabetical sorting function containing design errors.

```
/* An ASCII collating function -
** return YES if s1 to follow s2, else return NO.
*/
alphasort(s1, s2)
char *s1, *s2;
{
  while (*s1!=0 && *s2!=0)
  {
    if (*s1 < *s2)   return(NO);
    if (*s1 > *s2)   return(YES);
    /* strings are equal so far */
    s1++;     /* try next char in string */
    s2++;
  }
  /* we reached the end of one string */
  if (*s1)   return(YES);
  return(NO);
```



Caution: A MicroSCOPE debugger may create the need for additional hardware.

Give yourself the luxury of choice with MicroSCOPE.™ The symbolic cross debugger for 80x86 embedded systems.

MicroSCOPE speaks C, PL/M, FORTRAN, Pascal, Ada and Assembler. It has windows, full source display and immediate evaluation of any expression. Instant help is even supported by a quick "examples only" feature. So you can do your job faster.

And with advanced features like movie mode, 100 conditional breakpoints and very tight host-target integration, you'll have all the power you need.

It works on a PC or VAX with the best compilers (Intel, Microsoft, etc.), enhancing emulators by adding source display and symbolic evaluation. Or you can add new debug stations without an emulator, using Direct Connect.™ That saves you money.

Call us for a free demo diskette. Then make your life easier with MicroSCOPE and rediscover something everyone needs. A day off.

MicroSCOPE™

For a free demo diskette, call

1-800-242-5566

Trademarks of First Systems Corporation: MicroSCOPE, Direct Connect.
Copyright 1988 First Systems Corporation.

First Systems

CIRCLE #218 ON READER SERVICE CARD

SUBTLER BUGS

In addition to compile-time, link-time, and run-time errors, bugs resulting from integration, portability, and compiler errors may occur.

Integration bugs form a huge class of programming errors. These errors manifest themselves when two or more modules are combined to form a program. The bugs may not cause an error when the modules are tested separately but may show up as the system becomes integrated into one complex program. During integration, function return-value typematch bugs can become apparent. A function may return an error status if the data it processed is invalid. If the caller fails to check the error status, it may inadvertently continue processing with bogus data.

The opposite may also occur: a function that's supposed to return a value may instead contain a void return. In this case, the returned value is undefined and may appear to work during initial testing. Once integrated into a program, though, it creates an unexpected bug.

Global variables often cause problems that surface during integration. These variables are like salt: they should be used sparingly lest they spoil the stew. One module can change the currency of a global variable and cause another module to do something unexpected later.

These types of errors require the programmer to rethink the layering of the program at a modular level. A source-level debugger may be needed to understand how the interactions occur and to hack in a fix, but the real solution might well lie in the program architecture.

Interrupts can present their own set of difficult bugs. An obvious example of an interrupt bug occurs when the interrupt neglects to save or restore the entire status of the machine before returning, as often happens when modifications have been made to an interrupt routine. If the modification uses a register that wasn't saved during the interrupt prologue, then any routine in the foreground program that also uses that register is vulnerable to the interrupt. This oversight can cause previously working code to break.

An interrupt routine may call a function that isn't reentrant (a problem if the foreground program also uses that function). Library routines, particularly in floating-point math libraries, may be reentrant for one brand of compiler but not for another.

Generally, interrupt routines require that a debugger be able to deal with code at the assembly language level. In addition, the debugger must be able to operate transparently to the interrupt and vice versa. In multithreaded or multitasking systems, integration bugs can breed and multiply. Shared resources must either have lock semaphores or be designed to be reentrant.

Processes that `malloc` memory or open files must free the resource when they're done with it; otherwise, the system will eventually—maybe even two or three days later—run out of memory or file handles and lock up. Programs employing `setjump/longjump` and `goto` statements must be carefully designed to avoid abnormal control flow that may leave these resources tied up.

A cousin of the typematch bug is the portability bug. This bug surfaces during porting from one machine to another and can cause both intent and execution errors. Since C is implemented in different ways on different machines, tricks that work on one machine may not work on another.

Variations in the size and alignment of various objects, particularly differences in the implementation of types `float` and `double`, can also cause problems. Segmented microprocessors may treat pointers differently from nonsegmented machines; stack-addressing direction and byte ordering can also be different. Programmers with limited experience writing portable C code will undoubtedly discover these pitfalls the first time they compile their old programs on a new machine.

Compiler bugs are rare, but they do occur. All too often, the bug isn't really the fault of the compiler but of poor coding practice. Many of the newer compilers on the market perform code optimization, and the optimizer may make assumptions about the program that aren't desired. This problem is further complicated by the fact that debugging is usually performed on unoptimized objects, while the debugged program is compiled using the optimize mode. If bugs show up after optimizing, then the optimizer is probably the culprit.

Optimized code, especially globally optimized code, is tougher to debug since the object program may not match the source code line for line. Also, some source-level debuggers won't even allow optimized code to be debugged, forcing us to retreat to our assembly-level debugger for support.

The most common optimizer bug is caused by memory-mapped I/O. Consider an I/O device memory-mapped at address `data_port`. The function in Listing 4 waits for ready status, writes a sequence of characters to the device, then returns the new ready status to the caller.

An optimizer could just have a ball with this. First, it might consider that the `while` loop was a redundant read of the same address and replace it with a single read of the location. Next, it might determine that there was nothing to do in the body of the `while` anyway and decide that the entire `while` statement wasn't even needed. It would probably assume that the `A` and `C` assigned to the address were dead stores and replace them with the single assignment `*data_port = 'K'`. As for

Global variables are like salt: they should be used sparingly lest they spoil the stew. They often cause problems that surface during integration.

WRITING MICROCODE ISN'T CHILD'S PLAY. . .

The speed and flexibility available from microcode architectures carry a price which must be paid by designers—complexity. But even complexity can be moderated with the right tools, and HILEVEL has them.

We've designed an array of microcode development systems and software to suit projects ranging from simple device controllers to complex real-time radar processors.

Write your programs in C using our new, reconfigurable cross-compiler, or use our meta-assembler to write time-critical programs in assembly code. Modules from either can be linked together, and there is a comprehensive source code editor and disassembler to simplify integration and debugging.

To support your hardware, we offer the only control-store emulator with memory in its pods. This dramatically improves access times. But more important, the memory configuration is determined in software. Now you can use this system for a variety of projects, with control stores up to 512 bits x 64K, without replacing pods or other hardware.

You can monitor all critical activity up to 256 bits at full speed, using the logic state analyzer in the development system. And, our performance measurement features will even help you find ways of improving your design.

Don't make your project more difficult than it has to be. Call HILEVEL today and let us show you how writing microcode just became easier!

DIAL TOLL FREE:
1-800-HILEVEL
In California, 1-800-541-ASIC

HILEVEL
TECHNOLOGY, INC.

At The Leading Edge Of Microcode Development

31 Technology Drive
Irvine, CA 92718
(714) 727-2100

FOR LITERATURE CIRCLE #219 ON READER SERVICE CARD
FOR A DEMONSTRATION CIRCLE #220 ON READER SERVICE CARD

the return, it might assume that the last thing written to the address, a K, should still be there and simply return K instead of reading the status. Now the code, if represented as C source, looks like this:

```
int sendack()
{
  extern char *data_port;
  *data_port = 'K';
  return((int)('K'));
}
```

where *data_port is the address of the device.

Because optimizers may rearrange the code to minimize computation, the only safe way to avoid optimizer errors when dealing with memory-mapped I/O is to compile the driver with optimization turned off. This practice is usually safe but in special situations will cause bugs. When optimizing the function in Listing 5, for example, an optimizer might make two assumptions: that the ratio x/y could be done one time, and hence calculated before the loop, and that the multiply operation in the array index computation i*4 could be avoided if the loop were written differently. The optimizer may produce something equivalent to Listing 6.

The function now has two bugs! First, the loop will never terminate because i is an unsigned character and can't reach 256; second, a divide error will occur if y==0. Some compiler optimizers are smart enough to detect these situations and avoid producing code with these bugs.

SUGGESTIONS FOR WRITING BETTER C

The quickest way to debug a program is to write a program that has no bugs. Software that's modular and nicely layered will usually have fewer integration bugs. Here are some suggestions for producing code with a minimum of problems.

- Be extremely careful when using pointers; uninitialized-pointer bugs and boundary bugs can be very time consuming to correct.
- Look for typematch bugs by using lint or other utilities, or simply do a "paper debug" to check each function call for proper argument and return types.
- Be careful when using macros, especially those containing parameter substitutions. Capitalizing all macro names can serve as a reminder that they're macros, not function calls.
- Use parentheses liberally to guarantee associativity.
- Think about portability while writing code. If necessary, include a header file containing typedefs for basic types (BYTE, WORD, etc.), then use these instead of char and int. Programs can then be ported to another machine or compiler by modifying the typedefs in the header file.
- Use casting when converting types; don't expect the compiler to do it for you.
- Avoid global variables.
- Use header files for function prototyping and argument definition.
- Use return codes for modules that interface with each other.
- Above all, design the tests to exercise all branches in the program. Include tests for boundary conditions, especially for code using pointers. If possible, keep a test suite for future use, should the module ever be modified or ported to another environment.

Writing software is a complicated and tedious puzzle. Even with a concerted effort by the programmer, it's nearly impossible to produce a nontrivial program that's bug-free the first time. Knowing the potential causes of bugs allows us to adopt disciplines to minimize their occurrence and guides our efforts to stabilize, localize, and correct them.

Robin Knoke cofounded Applied Microsystems Corp., Redmond, Wash. He's currently involved in the specification and design of productivity tools for creating and debugging embedded systems software.

Listing 4

Code containing an optimizer bug caused by memory-mapped I/O.

```
int sendack()
{
  /* address of device */
  extern char *data_port;
  while(*data_port == 0)
   /* wait here for 'ready' status */
   ;
  /* send 'ACK' and sequence */
  *data_port = 'A';
  *data_port = 'C';
  *data_port = 'K';
  /* return status to caller */
  return((int)(*data_port));
}
```

Listing 5

An optimizer might make incorrect assumptions regarding *x/y* and *i*4*.

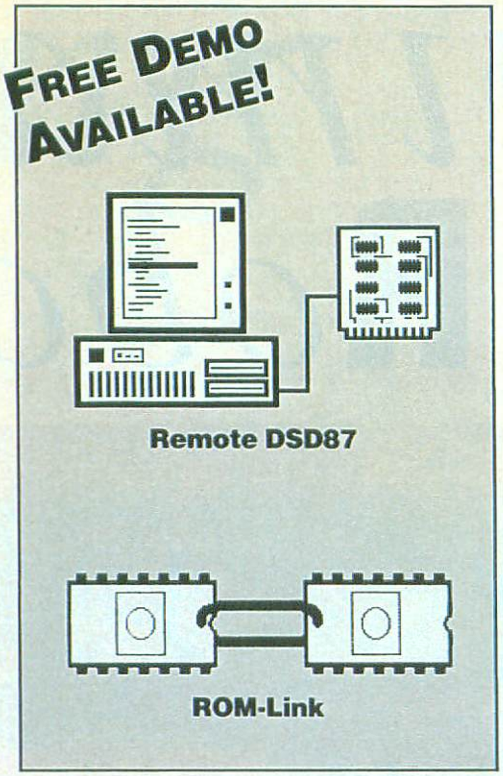
```
float array[256];
calcarray(x,y)
float x,y;
{
  unsigned char i;
  /* put ratio in every 4th cell in array */
  for (i=0; i<64; i++)
  {
    if (y != 0)
      array[i*4] = (x/y);
  }
}
```

Listing 6

A possible result of optimization of the Listing 5 code: an endless loop and a divide error.

```
float array[256];
calcarray(x,y)
float x,y;
{
  unsigned char i;
  register float tmp;
  tmp = x/y;
  /* put ratio in every 4th cell in array */
  for (i=0; i<256; i+=4)
  {
    if (y != 0)
      array[i] = tmp;
  }
}
```

For the Ultimate in Embedded Systems Programming.



Debug Debug Debug Debug Debug

Remote DSD87 is the best source debugger for embedded development and the only product offering high performance and advanced software In Circuit Emulator capabilities. This package and your compiler form a complete solution for the development of embedded programs. Order now and receive a FREE copy of Native DSD87, a $125 value.

Remote DSD87 (supports both Microsoft and Turbo C) . . . $395
Remote DSD87 OMF (includes Intel OMF support) $795

Link Link Link Link Link Link Link Link

ROM-Link is the first integrated link and locate utility to directly generate COM, EXE, SYS, HEX, or Intel OMF files. The only linker ever guaranteed faster than any present or future competitors. Supports Microsoft debugging information. Receive a FREE librarian and cross reference utility with your order!

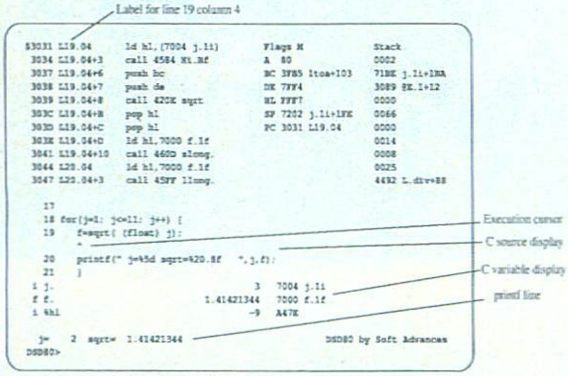
ROM-Link $395

Soft Advances
10811 Washington Bl., Ste. 205 • Culver City, CA 90232 • (213) 559-7015

CIRCLE #221 ON READER SERVICE CARD

Embedded C Debugging.

We have it now for Z80 and HD64180 family microprocessors! *Z-World C* and Z-World's *Remote Debugger* make it easy to utilize C, or C mixed with assembly language, for programming embedded systems. We have created a C language programming system designed for the practical microprocessor engineer. We do give you the library source. We don't change all the names between the C source level and the assembly language level. We do support high-speed, IEEE format floating point. You can mix in assembly language and you can easily access all variables from that assembly code. *Z-World C* is only $495.00 and it comes with an unconditional, 100%, 30-day, money-back guarantee. The *Remote Debugger* is $195.00.



```
$3031 L19.04      ld hl,(7004 j.1i)     Flags M             Stack
 3034 L19.04+3    call 4584 Xt.Rf       A  80               0002
 3037 L19.04+6    push bc               BC 3FB5 Itoa+103    71BE j.1i+1BA
 3038 L19.04+7    push de               DE 7FF4             3089 fE.I+12
 3039 L19.04+8    call 420E sqrt        HL FFF7             0000
 303C L19.04+B    pop hl                SP 7202 j.1i+1FE    0066
 303D L19.04+C    pop hl                PC 3031 L19.04      0000
 303E L19.04+D    ld hl,7000 f.1f                           0014
 3041 L19.04+10   call 460D along.                          0008
 3044 L20.04      ld hl,7000 f.1f                           0025
 3047 L20.04+3    call 45FF llong.                          4432 L.div+88

   17
   18 for(j=1; j<=11; j++) {
   19    f=sqrt( (float) j);
         *
   20    printf(" j=%5d sqrt=%20.8f   ",j,f);
   21    }
 i j.                             3   7004 j.1i
 f f.                    1.41421344   7000 f.1f
 i %hl                           -9   A47E

  j=    2  sqrt=  1.41421344                  DSD80 by Soft Advances
DSD80>
```

Typical Debugging Screen as seen on an IBM-PC while connected to a remote target.

Need Hardware Aids?

Our ic80 in-circuit emulator makes it easy to do source level C debugging on your Z80 or HD64180 target. Only $995.00. Our SBC100 HD64180 single board computer and prototyping platform provides a quick and low-cost development system for $295.00. Please call for complete information.

ic80 In-circuit Emulator

Z-World, 1772 Picasso Ave, Davis, CA 95616
(916) 753-3722 Fax: 753-5141
"Z80 Family Specialists"

CIRCLE #222 ON READER SERVICE CARD

BY BOB ZORICH

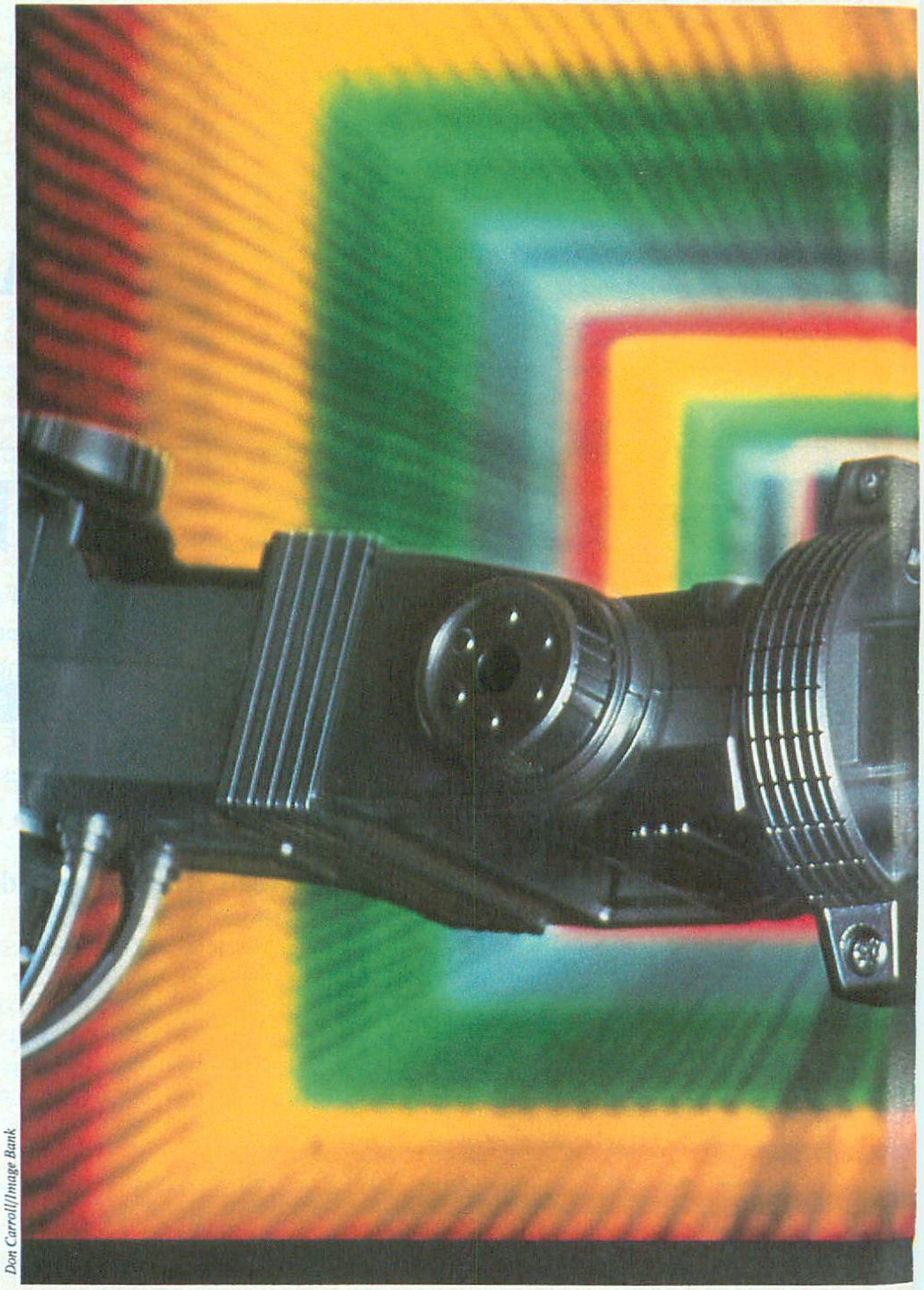
Animat the Robot

American industry is increasingly looking to automation as a means of improving productivity and reducing cost. Full-time operation, lower tolerance for overhead, and increasingly stringent precision and cleanliness requirements have led to a wide variety of approaches to robot construction. While many of these approaches are incompatible with one another and divergent in design, some features are common to all robots.

These baseline standards accommodate the features that are usually desired or required in an independently acting robot. For these features to be properly implemented in the robot, the hardware and software must be integrated at a very early stage.

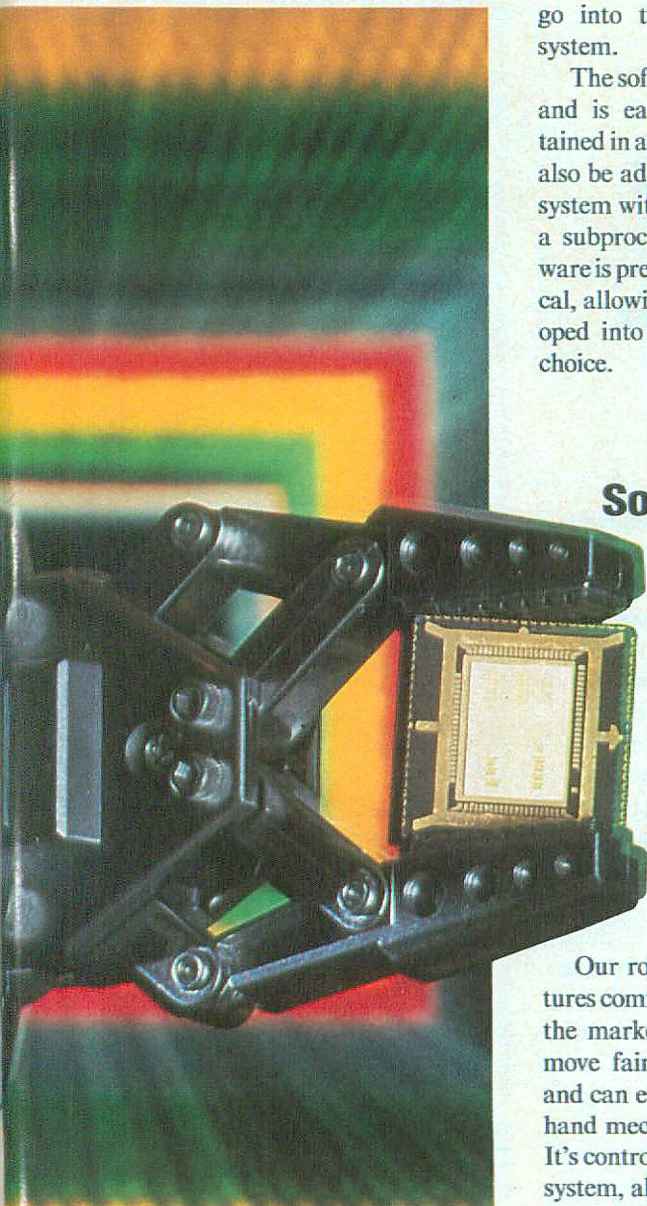
Too many design teams have made the mistake of assuming that either hardware or software solutions can address all the requirements of the system. In the first case, a machine may be hard-wired to be good at its particular task but incapable of adapting to new requirements. In the second, control resides in software run as an external computer, maximizing flexibility at the expense of performance and feedback bandwidth.

The embedded CPU approach resolves the dilemma by allowing specific aspects of a robot to be run independent of a host computer, combining the power of a hard-wired robot with the flexibility of a software-controlled robot. The application of this approach to motion-control algorithms and basic control systems is of particular interest to systems designers and is covered here using a simple robot as an example. This robot isn't particularly complex but provides a good demonstration platform without requiring us to



Don Carroll/Image Bank

ing



go into the specifics of a particular system.

The software outlined here is modular and is easily implemented and maintained in any number of languages. It can also be adapted for use in a single-CPU system with the robot control running as a subprocess. Pseudocode for the software is presented in a form similar to Pascal, allowing the algorithms to be developed into the source language of your choice.

Software should be modular, with simple routines defined, debugged, and combined as necessary.

Our robot has a broad range of features common to most of the machines on the market today. It has the ability to move fairly freely in three dimensions and can easily be fitted with a variety of hand mechanisms for greater flexibility. It's controlled by an embedded computer system, allowing it to run independently from the host computer. This embedded computer also allows the robot to be self-diagnosing, with some capacity to maintain itself during normal operation.

In the process of building first the primitive software routines, then the composite motion algorithm, we'll design such features as self-diagnostics and self-maintenance capabilities, feedback-control loops, and exception handling into the robot. The exception code should anticipate as many aspects of potential situations as possible to prevent damage. Warning: Murphy must have known about automatons when he formulated his law; robots will continually surprise and amaze you with their behavior.

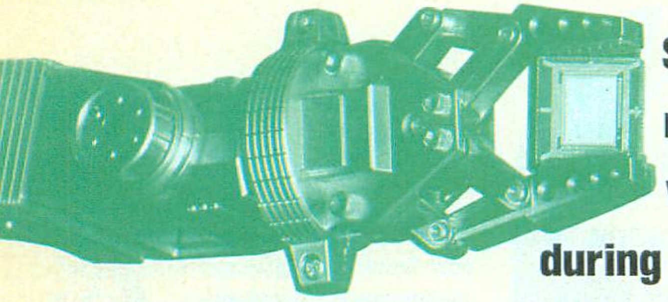
SOFTWARE SETUP ROUTINES

System software for the robot should be designed in a manageable, modular format. The basic routines for simple motions should be determined and written, then combined as necessary. These basic routines include axis acceleration, deceleration, and steady-state motion operation.

After each motion is defined, the larger, more complex motions can be formed. These composite motions perform such routines as exception handling and picking up, moving, and placing the cargo. Finally, these are all integrated to form a single, smooth operation.

The first primitive routine we'll need is a motion initiation sequence (see the pseudocode in Listing 1). This motion can take place in any of the three axes and can involve rotation of one of the axes. The axis of travel is specified first, followed by the direction in which the robot must travel. If the motor controller is being controlled directly by the main CPU, the software must know which direction the motor travels in as a function of voltage applied. While this may sound trivial, more than one robot has crashed or experienced development slowdowns due to details such as these.

Animating the Robot



Stability must be verified during the acceleration routine. This could involve a range of simple and complex position sensors.

We'll assume that the positive directions of motion are from the pickup point to the target on the x-axis and from the x-axis toward the target on the y-axis. The positive z-axis is up, while the positive direction of rotation is counterclockwise. Note the use of the "right-hand rule" in defining the coordinate system.

After the direction and axis of travel have been defined, we need to send the motor controller the velocity and acceleration values. These numbers must be verified for performance and can only be characterized depending on the application. Certain robots move much faster than others depending on the load, construction materials, etc. These data points should be stored in data tables and used to determine the optimum rates of travel. If you have a particularly powerful CPU, appropriate sensors, extra RAM storage, and some good programmers, you can make the system self-analyzing and able to modify its velocity and acceleration files occasionally.

In any event, these values must be prepared for transfer to the motor controller. The data shouldn't be sent until all other setup items are complete, so until that time either the data should be stored or the "go" signal withheld, depending on the type of controller. We'll assume that the controller has a separate go signal, allowing us to transmit the velocity and acceleration directly to the controller. This usually involves sending a reset signal to the motor controller to stop all current activity. If activity is detected or the robot's location is improper, we must go to the exception-handling routines before executing the move. There may be dire consequences if the robot is moved while in an illegal state.

When the robot is in the correct state, the motor controller is sent the go signal. There may be a variety of things to check at this point depending on the sensor setup. If there's a current-monitoring device, we may wish to observe it for jerky motion or jamming of the axis in the direction of travel. The voltage and current driving the motor should be changing at a smooth, consistent rate until the robot reaches cruising velocity.

During acceleration, the CPU must count the number of steps seen by the optical encoder on the servo motor to control the exact position of the robot and keep track of the acceleration rate. Acceleration is easily checked by setting a timer, counting a certain (small) number of steps from the optical encoder, and comparing that against the time elapsed. Additional effort may be required if the optical encoder has no method of determining the direction of travel.

One last point that could be important when moving very heavy or very light loads is that the cargo stability must be verified during the acceleration routine. This could involve a range of sensors from simple switches for confirming the position of the load to complex accelerometers and laser-positioning systems that measure all the load characteristics. If problems are observed in the acceleration phase (a fairly common occurrence), the software must flag this and set lower ac-

Listing 1
Axis acceleration control routine.

```
Procedure  Start_Axis_Motion;
  Begin
    Verify_All_Previous_Errors_Corrected;
    { x-, y-, or z-axis or z-rotation}
    Specify_Axis_of_Control;
    Specify_Direction_of_Travel;
    Verify_Current_Location_of_Robot;
    Get_Velocity_&_Acceleration_from_Table/File;
    Send_Velocity_&_Acceleration_to_Motor_Controller;
    Send_"Go"_to_Motor_Controller;
    While Current_Velocity < Set_Velocity do { Accelerating }
       Begin
         Watch_for_Encoder_Signal; { This section can be }
         When Encoder_Signal seen  { used as a procedure }
            Begin                { (e.g., Check_Velocity) }
              Start_Timer;
              Watch_for_Next_Encoder_Signal;
              When Encoder_Signal seen
                 Begin
                   Stop_Timer;
                   Calculate_Current_Velocity;
                   Calculate_Current_Acceleration;
                   Verify_Correct_Velocity_&_Acceleration;
                   If Velocity_or_Acceleration incorrect
                     Begin
                       If Problem_is_Serious then
                         Invoke_Exception_Handler
                       Else
                         Make_Adjustments_as_Required;
                     End;
                 End;
            End;
       End;
     When Current_Velocity = Set_Velocity
       Begin
         Check_Velocity;
         Check_System_Status;
         If (Velocity_OK and Status_OK) then
           Leave_Routine
         Else
           Invoke_Exception_Handler;
       End;
  End;
```

Listing 2

Axis steady-state motion control routine.

```
Procedure  Continue_Axis_Motion;
  Begin
    Verify_All_Previous_Errors_Corrected;
    { x-, y-, or z-axis or z-rotation}
    Specify_Axis_of_Control;
    Specify_Direction_of_Travel;
    Verify_Current_Location_of_Robot;
    Verify_Current_Velocity_of_Robot;
    Get_Velocity_from_Table;
    Compare_Expected_Velocity_to_Real_Velocity;
    Adjust_Robot_Velocity_as_Required;
    Repeat
      Check_Velocity_of_Robot;
      Check_Status_of_Robot;
      If (Velocity_Incorrect or Status_Shows_Problem) then
        Begin
          Check_Seriousness_of_Problem;
          If Problem_Serious then
            Begin
              Invoke_Exception_Handler;
              Exit_Procedure;
            End
          Else
            Adjust_Robot_&_Reset_Flags;
        End;
    Until Robot_Has_Reached_Deceleration_Point;
  End;
```

The exception-handling routines must have high priority in any multiprocessing operating system's scheduling process.

Listing 3

Axis deceleration control routine.

```
Procedure  Slow_Axis_Motion;
  Begin
    Verify_All_Previous_Errors_Corrected;
    { x-, y-, or z-axis or z-rotation}
    Specify_Axis_of_Control;
    Verify_Axis_&_Direction_of_Travel;
    Verify_Current_Location_of_Robot;
    Get_Velocity_&_Deceleration_from_Table/File;
    Send_Velocity_&_Deceleration_to_Motor_Controller;
    Send_"Go"_to_Motor_Controller;
    While Current_Velocity > Set_Velocity do { Decelerating }
      Begin
        Watch_for_Encoder_Signal; { This section can be }
        When Encoder_Signal seen  { used as a procedure }
          Begin                   { (e.g., Check_Velocity) }
            Start_Timer;
            Watch_for_Next_Encoder_Signal;
            When Encoder_Signal seen
              Begin
                Stop_Timer;
                Calculate_Current_Velocity;
                Calculate_Current_Deceleration;
                Verify_Correct_Velocity_&_Deceleration;
                If Velocity_or_Deceleration incorrect
                  Begin
                    If Problem_is_Serious then
                      Invoke_Exception_Handler
                    Else
                      Make_Adjustments_as_Required;
                  End;
              End;
          End;
      End;
    When Current_Velocity = Set_Velocity
      Begin
        Check_Velocity;
        Check_System_Status;
        If (Velocity_OK and Status_OK) then
          Leave_Routine
        Else
          Invoke_Exception_Handler;
      End;
  End;
```

Listing 4

Example of composite motion.

```
Procedure  Move_From_PickUp_To_Target;
  Begin
    Home_Robot_Positions;
    Select_Z_Axis;
    Rotate_to_Correct_Angle;
    Set_Z_Axis_Height_to_Fit_Under_Cargo;
    Select_Y_Axis;
    Extend_Y_Axis;
    Verify_Hand_is_Under_Cargo;
    Select_Z_Axis;
    Raise_Cargo_on_Z_Axis;
    Select_Y_Axis;
    Retract_Y_Axis;
    Verify_Cargo_Properly_Loaded;
    Select_Z_Axis;
    Rotate_to_Transport_Angle;
    Select_X_Axis;
    Accelerate_X_Axis;
    Move_Robot_to_Transfer_Position;
    Stop_X_Axis;
    Select_Z_Axis;
    Rotate_to_Placement_Position;
    Adjust_Height_to_Target_Placement__Height;
    Select_Y_Axis;
    Extend_Y_Axis;
    Verify_Cargo_Over_Target;
    Select_Z_Axis;
    Lower_Cargo_onto_Target_Platform;
    Release_Hand_from_Cargo;
    Select_Y_Axis;
    Retract_Y_Axis;
    Reset_&_Home_All_Axes;
    Complete_Routine;
  End;
```

celeration values. Like it or not, Newton's laws are here to stay, and momentum and inertia must be considered.

Any problems discovered during acceleration must cause control to go to the exception-handling routines; a failure or problem at this point may be amplified later. However, since the motion has already been started, it's up to the exception handler to decide whether to continue the action, compensate for the problem, or abort the process. For instance, a rapid, severe rise in current draw, associated with a sudden slowing of the acceleration rate, indicates a mechanical problem that may necessitate immediate shutdown. The discovery of a miscount in the number of steps taken to reach a certain point, on the other hand, may simply require that the robot move to a new position at some point in its travels.

The exception routines must have a very high priority in the scheduling process of any multiprocessing operating system. We'll assume that error handling will control system resources from the moment of discovery.

Animating the Robot

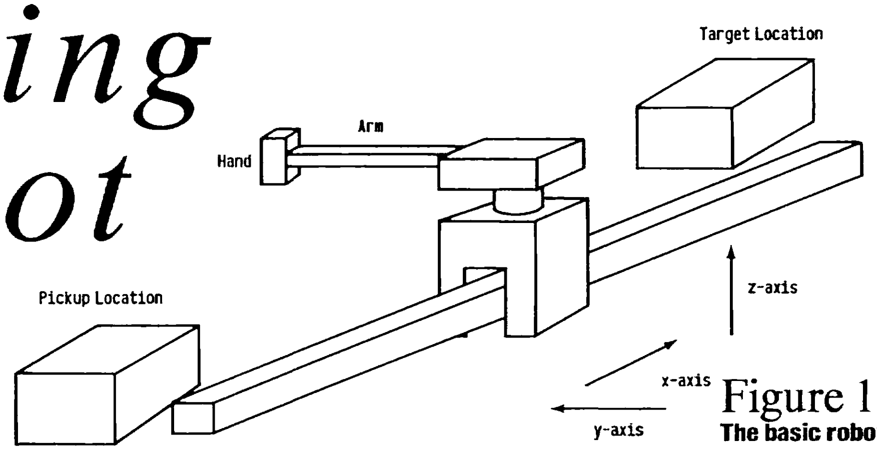


Figure 1
The basic robot.

STEADY-STATE MOTION

After the robot has successfully reached the velocity setpoint, program control goes to another of the primitive routines—steady-state operation (see Listing 2). Since most problems occur at the beginning or end of the trip—during acceleration or deceleration—this phase should be relatively uneventful unless something breaks or gets in the way of the robot.

However, there's always work to do. In most cases, it involves constantly analyzing and verifying the current position and velocity. If external devices measure location, they should be referenced against the expected position as determined by the optical encoder counter. If any discrepancies are detected, the routine must determine their magnitude. If they're minor, the routine makes the required adjustments and goes on about its business. Major discrepancies require that control be passed to the exception-handling routines.

We need to monitor constantly for sudden stops, contact with objects, and cyclic changes in velocities or other parameters. Contact with objects may be sensed directly with contact circuitry or indirectly by observing locations of optical encoders and checking for relative changes. Cyclic changes may point out areas of friction or stress in the robot drive mechanism or heavy cargos swinging around, possibly indicating excessive acceleration or speed. If cyclic changes are seen, the host computer or operator should be notified.

We also want to watch for contact with limit switches, endpoint switches or detectors, and overtravel switches. If a limit or endpoint switch is triggered, program control should go to the deceleration routine; if a problem is discovered, control should pass to the exception-handling routines.

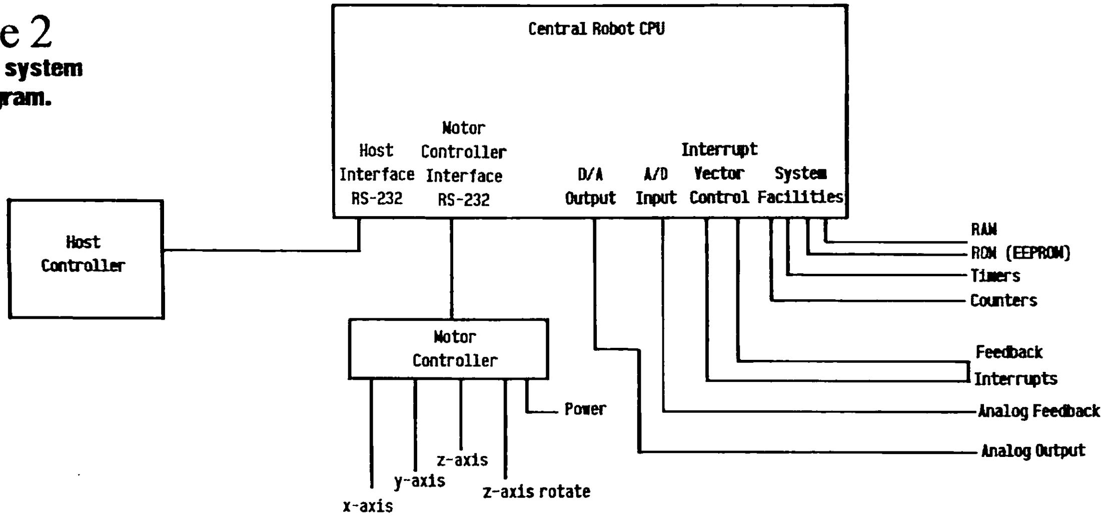
ROBOT DECELERATION

Finally, when all systems indicate that the cargo is near its destination, the routine enters the deceleration phase. This phase is initiated by sending the motor controller the signal to decelerate (or, in some controllers, accelerate with a negative value) with an ultimate velocity of zero, followed by a go signal (see Listing 3). The CPU must then verify deceleration rates and exact position using timers and encoders, as outlined earlier for the acceleration phase. This action verifies that the position of the robot on that axis is correct as specified.

One of the key functions of the deceleration phase is to anticipate the point at which the robot must initiate the next action. This requires that the current parameters be evaluated in real time, especially if the system has self-diagnostic abilities. If the CPU realigns the robot in real time, a degree of flexibility should be built into the hardware for the adjustment tables (for example, the point of initiation into the deceleration phase). This can be accomplished through the use of EEPROMs or nonvolatile RAM chips.

It's quite easy to be slightly off-target when the robot stops, even after all our attempts to prevent problems. There's a certain amount of hysteresis in real devices, especially belt or chain drives. Because set screws become loose and other unpredictable and unpreventable events

Figure 2
Computer system block diagram.



occur that cause deviations, a routine should be added to the code to handle such problems. Even though this adjustment routine may not be necessary when the robot is brand new, we'll be happy to have the flexibility as these slight deviations occur.

One way to minimize the impact of deviations is to use the setpoint or endpoint as the prime reference to the desired final location for the robot. In most cases, the closer to the actual stop point the reference locator is placed, the more accurate the actual position will be. For maximum precision, both the deceleration point and the actual endpoint should be referenced by the CPU, with discrepancies adjusted by moving along the axis the required distance.

While it's possible to include these routines only in the exception handler, self-correcting software should have the adjustment routines built into or referenced by the deceleration routines to allow the robot to come as close as possible to the desired endpoint.

If any exceptions are observed during deceleration, the program should continue with the stopping procedure. If the problem reaches crisis proportions, of course, control should be transferred to an exception routine. Sudden changes in motion during this phase could cause a number of problems, so continuity is probably the best goal.

COMPOSITE SOFTWARE ROUTINES

Now that we've defined the major functions the software must perform, we can start putting the pieces together to form an integrated system. In constructing integrated motions, we'll use a modular design structure to keep the program clean and ensure that the integration routines are relatively easy to construct.

In almost all cases, the composite routines consist entirely of the basic steps outlined here. Each axis can be controlled by the same basic routine. Rotational motion is similar in concept, though somewhat different in orientation.

As an example of composite motion, let's consider the case of moving an object from a stand to a target location (see Listing 4). The routine starts by resetting the robot to verify that it's in the correct start-up position. This involves moving axes to the *home*, or zero-state, condition (all axes at 0). When the robot reaches the reference point, the CPU is notified and the robot is reset.

The robot must then get into the correct position to pick up the object. First, the x-axis is adjusted so that the bulk of the robot is in the correct position for activity. Next, the appropriate hand angle is determined and the hand rotated to that position. The z-axis is adjusted to verify that the hand will slide under the cargo, and the y-axis is extended until it's under the cargo.

The robot then raises the cargo off its stand by moving the z-axis upward. The z-axis should raise only high enough to allow the cargo to move; raising it too high increases the likelihood of damage. If necessary, the y-axis should retract to place the cargo over the robot's center of gravity. This is especially important when the robot has to carry heavy loads or when it has insufficient space available for counterbalancing.

The arm may need to release its cargo very precisely on the target platform to avoid imbalances.

Note that in each of these cases, the sequence of events is essentially the same:

1. Select axis.
2. Verify current location on axis.
3. Calculate distance to travel and appropriate acceleration rates.
4. Accelerate axis.
5. Move to correct location.
6. Decelerate axis.
7. Verify endpoint was reached; adjust if required.
8. Continue to next action.

The second motion in the composite routine is from the pickup point to the target point. We must verify that the y-axis has moved so that the cargo is properly placed for transport and that the z-axis has the cargo properly rotated. We then accelerate the x-axis to the main transport speed and traverse the x-axis, observing orientation, location, velocity, and other parameters.

The system must determine the point at which deceleration should begin. When that point is reached, the deceleration routine is initiated to bring the robot to a stop. The stopping point on the x-axis is then adjusted and the cargo is rotated until it's aligned for placement on the z-axis.

Listing 5

Exception-handling routine.

```
Procedure  Exception_Handler;
  Begin
    Read_Current_CPU_Status;
    Read_Current_Robot_Status;
    Check_for_Emergency_Situations; { e.g., impact }
    If Emergency then
      Begin
        Execute_Emergency_Action; { may be emergency stop
        Return_Data_to_Host_Computer;
        Exit_or_Halt;
      End;
    Prioritize_Problems_Found;
    Repeat
      Place_Problem_in_Category;
      Check_for_Common_Categories;
      Based_on_Common_Traits_Choose_Potential_Problem;
      Based_on_Problem_Determine_Adjustment_to_Make;
      Execute_Adjustment_to_Robot;
      Check_Effectiveness_of_Adjustment;
      If Adjustment_Worked then
        Begin
          Reset_Errors;
          Continue_Routine;
        End
      Else
        Begin
          Look_for_Other_Potential_Solutions;
        End;
    Until All_Problems_Analyzed_and_Fixed;
  End;
```

The final portion of the composite motion involves placing the cargo on the target platform. This action can be quite critical, as the object may have to be placed very precisely to avoid imbalances and other problems. This section must first verify that the cargo is correctly oriented for placement. The rotation around the z-axis should be checked, followed by a check of the x-axis. (This should have been checked at the end of the x-axis motion section but should be redone at this point for good measure.)

The height of the cargo on the z-axis must then be adjusted to the proper height for placing the cargo on the target. As in the pickup section, it's important not to lift the object too high, as the potential for damage from an accident increases as a function of the distance the object falls. When "landing" the robot, assume the worst-case scenario; a significant proportion of problems with this type of transport robot occur at this point.

We now extend the y-axis out from the robot until the cargo is positioned over the landing target. Assuming that no problems have occurred, we can lower the cargo along the z-axis until it's aligned properly on the target. The next step is to continue lowering the hand until it's freed from the cargo. We can then retract the y-axis and return all the axes to their home positions.

EXCEPTION HANDLING

Inevitably, there will be times when the robot seems to have a mind of its own and wanders into territory forbidden to it. If something is forbidden, it's for a good reason—either the robot, the cargo, the operator, or something nearby would be damaged otherwise. Other conditions can be hazardous to the robot as well; for example, friction on the axis rods can cause the motors to pull excess power, which could cause overheating problems.

Another example is the possibility of a slight change in the shape or tension of various parts over time. This could create warpage or areas of contact between the cargo and the surroundings, possibly causing damage to the cargo or exacerbating the warpage problem. To detect these problems and minimize their impact, the software must include exception-handling code (such as that given in Listing 5) that responds appropriately to a given situation.

This section of the program should be as unified as possible and built as a separate module, not spread throughout the primitive routines. This is because many of the routines are common to all facets of robot operation and because there are likely to be frequent changes to these routines as development progresses.

The exception code is called whenever a major problem is sensed in the robot. The level you wish to call "major" may vary depending on your application but in most cases will involve significant changes in parameters (expected vs. actual positions, velocities, etc.), contact with obstacles, overheating or drawing excessive current, or activation of a limit switch. The primary routine should send information to the exception routines to identify what has happened. This can involve looking at the interrupt vectors or examining the registers or memory addresses that contain information regarding the nature of the problem. The primary routine leaves the status untouched so the exception handler can determine what happened.

If a cyclic error or other subtle problem is noted, this information should be sent to the host computer immediately.

After the exception handler has gathered the required information, the next step is to assign a priority to the problem. This is especially important if, as is likely to happen, a number of problems are identified simultaneously. In determining the seriousness of the incident, the routine should test all available sensor inputs to verify current conditions and test for commonalities. In most cases, this must be done quickly. Each priority code generated should be specific to a particular type of failure so that appropriate reactions can be formulated.

After the software has identified the problem and verified the consequences, the CPU should look through a table of known or expected symptoms, identify both the problem and the probable system failure, and determine the action to be taken. Based on the problem identified, the potential reactions can fall into a number of broad areas:

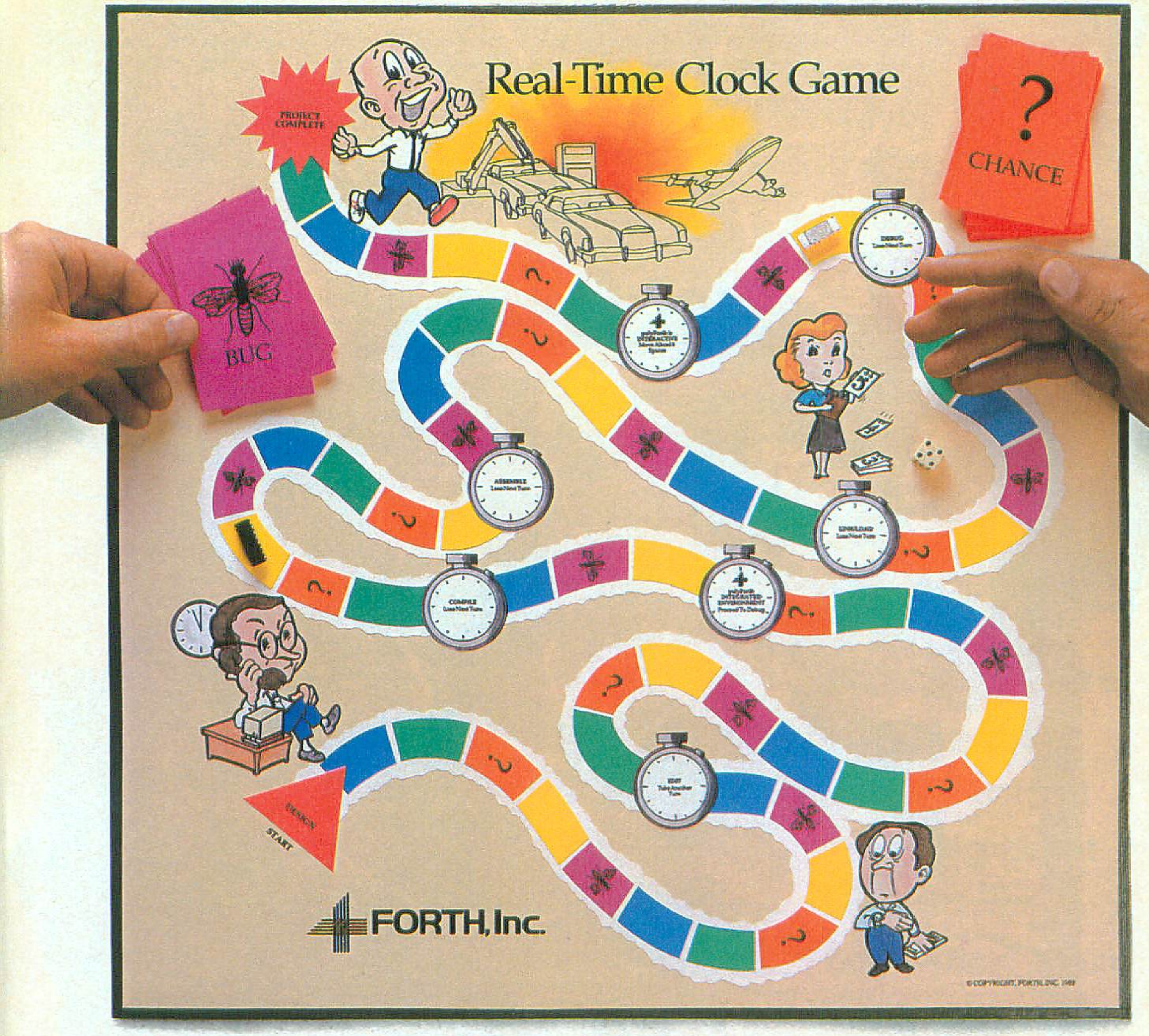
- Simple motor voltage adjustments in the case of improper velocities.
- Adjustment of reference points necessitated by a slight mechanical offset.
- Resets or rehomes for each axis.
- Entry into deceleration phase.
- Entry into sequence to reverse direction.
- Emergency stop.
- Emergency power-down.

After the system has reacted to the problem, it must observe the final status of the robot. In many cases, operator intervention is necessary to tell the robot whether to continue moving, go home, or just wait and do nothing. In any event, the robot CPU then reports the status of the robot to the host and adjusts the alignment tables or other setpoint files in EEPROM or nonvolatile RAM. If a cyclic error or other subtle problem is noted, this information should be sent to the host computer as soon as the problem is discovered.

SELF-CONTROLLING AUTOMATION

Industry continues to demand increasing quality and productivity from its factories and is the driving force behind the push to automation. As this trend continues, the techniques of robotics control and embedded independent CPU control become increasingly important. Such issues as maintainability, reliability, and ease of upgrades are of major consequence in an industrial robot; the greater the robot's ability to be self-controlling, the better. For all these attributes to be obtained, the robot must be developed as an integrated system that will meet industry's needs today and in the future.

Bob Zorich is a senior engineer at a major semiconductor manufacturing company and president of Acro Technology.



How To Win The Real-Time Clock Game

Developing software for real-time applications can be a challenging game.

Dealing with a separate compiler, assembler, linker, loader, executive and debugger, each with its own procedures, can really slow you down. Just think of all the time you spend accessing files and conversing with all those tools, plus the time waiting for them to run.

There's a better solution—polyFORTH® It's a complete, fully integrated, interactive programming environment that's designed specifically for real-time applications. It supports single-chip microprocessors, 16- and 32-bit processors, DSP processors, and minis like the VAX.

The editor, compiler, assembler and debugger are always resident and ready to use. There's only one program to learn and no separate procedures to bog you down. You can move directly from thinking and editing to testing, with no distractions. It's an integrated approach that can dramatically improve your productivity and creativity.

Fact is, polyFORTH is far and away the most consistent, effective way to go from "design" to "done". Whether you do it yourself or have us design a turnkey system for you, expect to finish your real-time embedded applications 4 to 10 times faster than ever before.

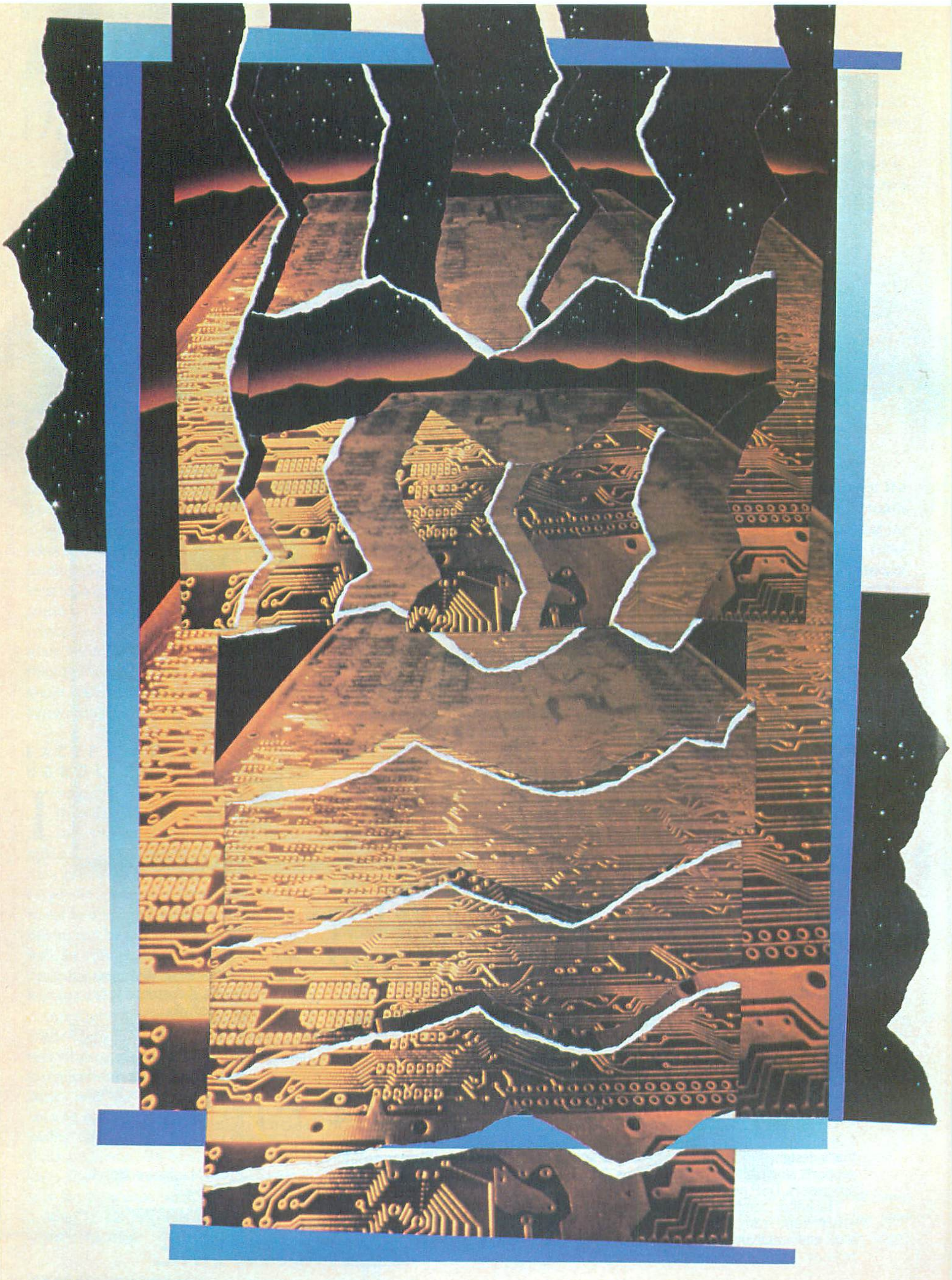
After all, isn't that the name of the game?

Call 1-800-55-FORTH today for your free-for-the-asking "Real-Time Clock Game" poster and brochure with real-life programming solutions from FORTH.

FORTH, Inc., 111 N. Sepulveda Blvd., Manhattan Beach, CA 90266, 1-800-55-FORTH, (213) 372-8493

VAX is a trademark of Digital Equipment Corporation polyFORTH is a trademark of FORTH, Inc.

FORTH, Inc.



BY RICK NARO

ROM*ing* DOS C*s*

So you've decided your next design requires the throughput of a 16-bit controller. This is a golden opportunity for the software development team assigned to the project to wield some clout and influence the choice of microcontroller. Selecting one of the popular 8086-compatible Intel or NEC processors can lower the cost of software development and have a significant impact on the project's bottom line.

The reason is simple: powerful software development tools are available for PCs in a wide variety of languages. If you forget for the moment about microcontrollers and instead focus your attention on the PC software development marketplace, you'll find an exceptionally competitive environment for software development tools. Microsoft, Borland, and dozens of other companies are constantly trying to gain the advantage by providing fast compilers that produce highly optimized code and that come with an impressive set of run-time libraries and utilities. A complete kit of the most powerful tools costs just a few hundred dollars, allowing each team member to have a set of tools and documentation.

Designs based on Intel 8086-compatible microcontrollers (Intel 8086/88 and 80186/188; NEC V20/V30, V40/V50, V33, and V25/V35) can take advantage of powerful PC-based tools from the early design stages through the production release of the software. Popular ROMable assemblers include Microsoft's MASM and SLR Systems' OPTASM; ROMable compilers include Microsoft C, Turbo C, Lattice C, Microsoft Pascal, Manx C, and JPI Modula-2.

These tools tremendously improve productivity by allowing short compile/link turnaround times and the use of popular run-time library routines. Even better, they let the software development team prototype and test algorithm design and preliminary versions of software on standard-issue PCs long before the first hardware prototypes are available. This feature alone greatly reduces the time required to perform the final hardware/software integration. This becomes increasingly critical as more and more hardware designs are simulated rather than prototyped.

While the application of PC software development tools to the embedded system environment is useful, it's not without problems. As compiler vendors have pushed to distinguish themselves from one another in the marketplace, the compatibility that once existed among different compilers has been lost. While this trend is beneficial for PC users, it does create problems when multiple tools are used in a ROM environment.

But the biggest potential downside to these tools is also the reason they're so popular to begin with—they're designed specifically for the development of PC applications. It seems that embedded applications take a back seat when it comes to obtaining vendor support. For this reason, it's important that you become an expert in the ROM compatibility issues of a particular compiler or linker.

Selecting an 8086-compatible processor from NEC or Intel can lower the cost and pain of software development and have a significant impact on the bottom line...if you know how to adapt native tools to development of ROMable code.

GETTING STARTED

To develop an embedded application, we'll need an assembler, compiler, linker, and locator. Run-time libraries are time-savers and should be used whenever possible. While other tools may be useful, these five components are critical. They must work together seamlessly to get us from program source to an executable, binary-format program suitable for a PROM programmer or in-circuit emulator.

The application can usually be developed with little attention paid to whether or not the final code will execute from ROM. The important considerations during initial development are the memory model to be used, the run-time library

Don Carroll/Image Bank

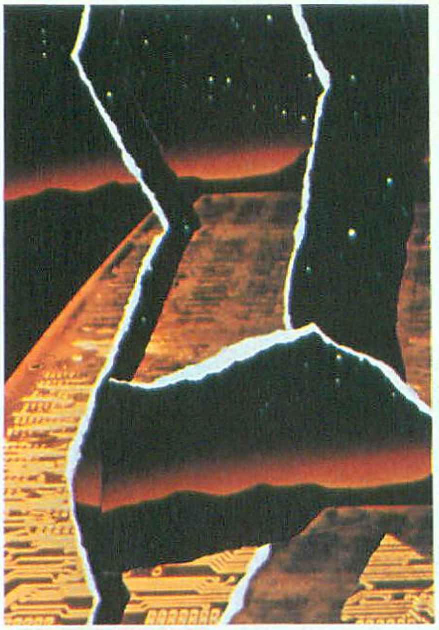
ROM*ing* DOS C*s*

routines, and any restrictions that the hardware will impose on the application address space.

Rather than explain the process in theoretical terms, I'll use a simple application to demonstrate the process. Assume we've been given the task of building the electronic egg timer shown in Figure 1. Based on discussions with the hardware department, we know the unit has a start switch, an LED display, and a small buzzer. At the zero count, the buzzer is to provide an audio cue to remove the eggs from the water. The tools we'll use in this example are MASM, Turbo C, and a Borland linker. We'll use a `locate` utility to place the code properly in ROM.

Because the hardware prototype isn't available at this early stage, we can proceed to prototyping the software directly on the PC. By properly partitioning the software, we can use the PC hardware (keyboard, display, and speaker) to emulate the hardware in our egg timer. Because the system-dependent code is modular, only the I/O drivers for the start switch, LED display, and buzzer will require modification when the time comes to bring up the software on the prototype.

The egg timer is designed to run in an endless loop until the start button is pushed. When it detects the start event, it writes the initial count to the display and decrements the count for each elapsed second. When the count reaches zero, the speaker emits three beeps and the cycle begins anew. Listing 1 shows the code needed to implement these specifications.



Listing 1
Example of code needed to run egg timer.

```
while (1) {
   /* Wait for start switch to be depressed */
   while (start_depressed() == FALSE)
     :
   /* Display the count continuously,
      decrementing once per second */
   write_LED_display(INITIAL_COUNT) :
   for (i = INITIAL_COUNT: i > 0: ) {
     delay(ONE_SECOND) :
     write_LED_display(--i) :
   }
   /* Beep three times */
   for (i = 0: i < BEEP_LIMIT: i++) {
     beep() :
     delay(BEEP_INTERVAL) :
   }
}
```

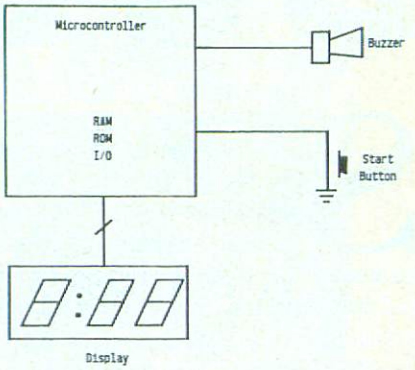
All that remains is to implement the code that checks for the start signal, displays the time, measures the elapsed time, and activates the buzzer. During testing on the PC, run-time library routines that delay for a predetermined time, cause a speaker to emit a beep, and wait for a character from the keyboard are used to simulate the egg timer hardware. By using these functions, we'll have much of our egg timer code debugged and ready for the prototype hardware.

Listing 2 provides an MS-DOS version of these routines.

START-UP CODE

With the prototype hardware available, new start-up code for the compiler becomes a necessity. ROM-based start-up code receives control following a reset. It sets up the segment registers and stack, then initializes any data structures required by the program. It may also initialize the hardware and interrupt vectors, although this can often be handled by a high-level-language initialization routine.

Figure 1
A simple electronic egg timer.



Most compilers divide the program data into two groups: `DATA` (initialized) and `BSS` (uninitialized, or, more accurately, initialized to zero). Because C requires that the former be initialized before `main()` receives control, the start-up code keeps a copy of the initialized data in ROM and copies it to its final destination in RAM. The `BSS` data area can be initialized by writing zeros to all locations.

Since a DOS linker is used, the start-up code is also used to define the order of segments in the .EXE file. In an MS-DOS application, segments are contiguous; in an embedded application, they're found in the noncontiguous address spaces of ROM, RAM, and I/O. When the start-up code controls segment order and classes in the .EXE file, related segments are grouped automatically.

The start-up code also defines placeholder segments to serve as handles for extracting and moving unrelated segments. `Locate` utility code, for example, copies the initialized data and places it in a segment after the executable code.

LINKING

By default, MS-DOS linkers place the segments in the order in which they're encountered in the various object modules. By having the

Code and debug micro-controllers in C without ever leaving your PC

Now you can run, debug, and test Archimedes Microcontroller C code right on your PC, and you don't even need any prototype hardware. Combined with Archimedes C, SimCASE allows you to speed up software development. You can test-run your software ideas before you even commit to a microcontroller design. It's like having a microcontroller built into your PC.

You'll have every traditional debugging tool at your fingertips, including trace, step and breakpoints. So you can fully debug microcontroller code at the C source level. Of course, you can use SimCASE to debug at the Assembly level too, if necessary.

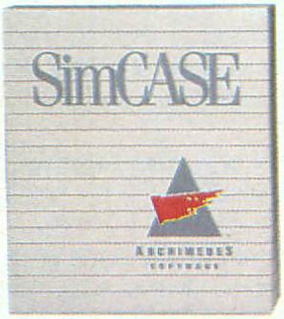
Speed up software development on all of today's most popular microcontrollers. Archimedes Microcontroller C and SimCASE are available for a wide variety of microcontrollers, including: Motorola's 6801 and 68HC11, Intel's 8051 and 8096/196, Zilog's Z80/Z180, Hitachi's 6301 and 64180.

Simulate and test your designs without hardware. At the heart of SimCASE is the Microcontroller Simulator Engine. Use it to simulate every part of your chip on your PC. Then use the various modules to control and analyze your simulation.

With the Input Stimulus Generator you can simulate real-time I/O intensive applications right on your PC.

Then use the Performance Analysis Tool to get the execution time of every block and line of code and identify any performance bottlenecks in your design. You can run this assessment for worst-case scenarios, including hardware tolerances.

All before you even commit to hardware.



Get your free demo diskette and see SimCASE in action. Get a taste of the full speed and power of Archimedes C and SimCASE. Order your free demo diskette and product guide today by calling 1-800-338-1453. In California call 415-567-4010.

Archimedes Microcontroller C and SimCASE. They set the standard by giving you fast, fully-featured C compiling, C-source level debugging and simulation of real-time microcontroller designs.

ARCHIMEDES™
SOFTWARE

Archimedes Software Inc.
2159 Union Street
San Francisco, CA 94123
415-567-4010
800-338-1453

Archimedes, Archimedes C, Microcontroller C, and SimCASE are trademarks of Archimedes Software, Inc.

CIRCLE #223 ON READER SERVICE CARD

ROM*ing* DOS C*s*

Don't overlook the vendor's run-time libraries. Depending on the compiler, you can ROM string, memory, search, sort, classification, and conversion routines.

start-up module declare the segments ahead of the other object modules, we can control the segment order and alignment for the next stage—locating the executable image in the target hardware address space. Next, the `locate` utility will need to tear apart the .EXE file using information in the segment map.

Other than requiring us to specify the start-up module first, the method we'll use to link our egg timer application is no different from in an MS-DOS application. Keep in mind that even if we didn't explicitly specify any run-time library routines in the application, the appropriate libraries may still need to be searched. Most compiler run-time libraries contain a large number of helper routines used by the compiler and by other run-time library routines. An example of these is a routine that performs 32-bit integer math but occupies too much space on a 16-bit processor.

The executable file is built using the following linker command line:

```
tlink /m /c tc eggtimer, eggtimer, egg
  timer, \tc\lib\cs
```

Don't overlook the power of the vendor's run-time libraries. Many standard run-time library routines require MS-DOS, particularly for file I/O. Nevertheless, many important routines are ROMable and should be used; they're highly optimized and will save precious development time. For example, an application can easily use string or memory functions because they're optimized for speed and don't rely on the presence of DOS.

Depending on the compiler, we can also use search, sort, classification, and conversion routines in an embedded application. If the dynamic range of data in an application requires the use of floating point values, the floating-point emulation and math run-time library routines can also be used in place of an expensive numeric coprocessor.

At last we're ready to prepare the ROMs for our egg timer. To successfully split the .EXE file, we need to supply information on the organization of the hardware and address spaces using the configuraticn file shown in Listing 3. The `locate` utility uses the configuration file to relocate the segments so that they match the physical configuration of the target system. The .EXE file contains both the executable code and a header with relocation pointers to segment fixups. Segment fixups are physical segment refer-

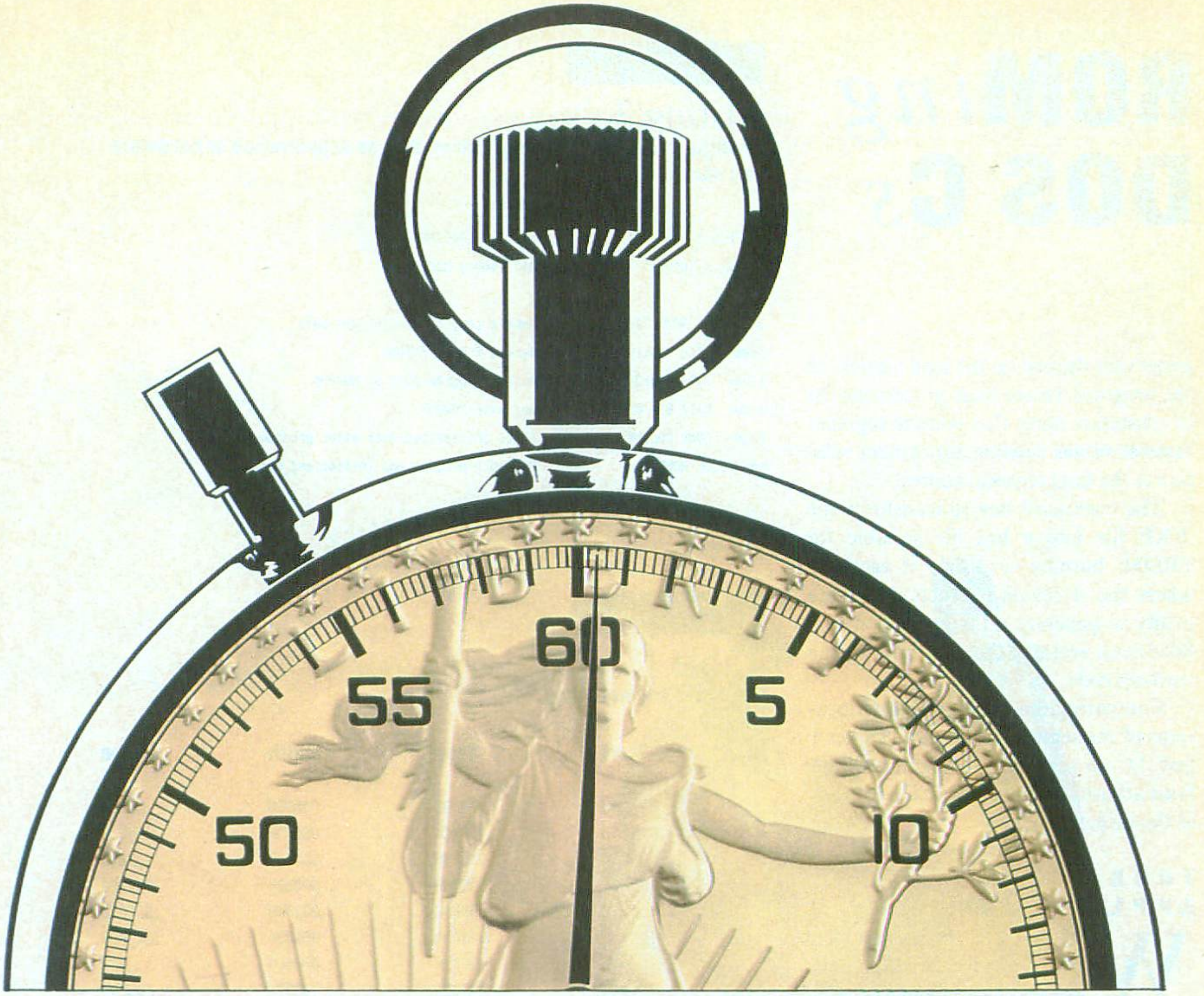
Listing 2

MS-DOS version of egg timer routines.

```
#include  <stdio.h>
#include  <conio.h>
#include  <dos.h>
#define   FALSE  0
#define   TRUE   1
#define   INITIAL_COUNT  3 * 60
#define   ONE_SECOND     1000
#define   BEEP_INTERVAL  300
#define   BEEP_LIMIT     3

void  write_LED_display(unsigned) ;
void  beep(void) ;
int  start_depressed(void) ;

void  main()
{
  int  i ;
  while (1)  {
   /* Wait for start switch to be depressed */
   while (start_depressed() == FALSE)
     ;
   /* Display count continuously,
     decrementing once per second */
   write_LED_display(INITIAL_COUNT) ;
   for (i = INITIAL_COUNT; i > 0; )  {
     delay(ONE_SECOND) ;
     write_LED_display(--i) ;
   }
   /* Beep three times */
   for (i = 0; i < BEEP_LIMIT; i++)  {
     beep() ;
     delay(BEEP_INTERVAL) ;
   }
  }
}
void  write_LED_display(count)
unsigned  count ;
{
  /* Convert to minutes:seconds format
     and display */
  printf("%u:%02u\r", count / 60, count % 60) ;
}
void  beep()
{
  /* Print BELL character */
  printf("\a") ;
}
int  start_depressed()
{
  int  c ;
  /* Test for depressed key and read it
     if necessary */
  if (kbhit())  {
   c = getch() ;
   if (c == 0)
     c = getch() ;
   return  TRUE ;
  }
  else
   return  FALSE ;
}
```



If Time Is Money, PDOS® Can Make You Rich.

PDOS puts speed and profits back into your realtime applications. Program development is done directly on your target hardware saving downloading time and debugging costs. A high priority program can respond to an external event in less than 27 micro seconds. The result is greater productivity and profits. Plus, PDOS comes ported to the top-line VME systems. Choose from over 200 hardware products and from more than 20 manufacturers.

PDOS is as complete as it is fast. The PDOS modular operating system includes a kernel, file manager, monitor, debugger and other support utilities. The PDOS editor, assembler, and linker are standard. Language compilers and tools for multi-processing and symbolic debugging are optional.

When you need support, we won't let you down. We provide concise documentation and hotline support. Because time is money, you can't afford to be without PDOS. Here's where to contact us: **Eyring Systems Software Division, 1450 West 820 North, Provo, Utah 84601. Telephone 801-375-2434 or Telefax 801-374-8339.**

PDOS Realtime Operating System

®PDOS is a registered trademark of Eyring

EYRING Creating Solutions™

PDOS Distributors In Benelux:
Interay BV
Lageweg 2A
9251 GM Bergum
The Netherlands
Telephone 05116/4052

In Germany, Switzerland and Austria:
Systrix GmbH
Hindenburgring 31
D-7900 Ulm/Donau
West Germany
Telephone 0731/37515

In United Kingdom:
Eyrisoft Ltd.
Etwall Street
Derby DE3 3DT
England
Telephone 0332/384978

CIRCLE #224 ON READER SERVICE CARD

ences that depend on the load address of the program. Locate must process the list of relocation items that point to segment register values needing adjustment relative to the final physical address.

The command line to transform the .EXE file into a hex file suitable for PROM burning is `locate -b eggtimer`, where the `-b` option instructs the `locate` utility to generate a far jump from the 8086 reset vector to the entry point of the start-up code.

Comparing the before and after locations of the segment addresses (Tables 1 and 2), we see that the segments have been adjusted to their physical addresses in the target system.

YOUR NEXT APPLICATION

While the process of using MS-DOS tools to develop embedded systems was demonstrated here using a simple application, the same steps can be used in larger programs. As the size and number of program source modules grow, the number of combine classes (see Table 1) in the .EXE file stays constant.

Other than assembling, compiling, and linking these modules, little additional effort is required to burn a more complex application in one or more PROMs. By following the simple procedures outlined above, we can develop a new generation of sophisticated applications for 8086-compatible microcontrollers using popular PC compilers.

[redacted]

Rick Naro is the president and founder of Paradigm Systems Inc., a developer of embedded system development tools for Intel and NEC microprocessors. Rick has been building embedded systems for Intel and NEC microprocessors for nine years and is also the author of INSIDE!, *a software performance analysis utility for PC compilers.*

[redacted]

Listing 3

Configuration file supplying information on organization of hardware and address spaces.

```
;
; This is the configuration file for EGGTIMER.EXE
; using Turbo C 1.5 in the small memory model.
;
dup  DATA ROMDATA;            Make a copy of initialized data
class  CODE = 0xfc00;         Assume ROM at FC000h
class  DATA = 0x0040;         Assume program data at 00400h
order  DATA BSS BSSEND STACK; Recreate DGROUP
order  CODE CODEEND ROMDATA;  Place initialized data after program code
rom  CODE ROMDATA;            ROM only program and initialized data
```

2

[redacted]

Table 1

Egg timer segment map.

Start	Stop	Length	Name	Class
00000H	0190DH	0190EH	_TEXT	CODE
01910H	01910H	00001H	_ETEXT	CODEEND
01920H	01BC9H	002AAH	_DATA	DATA
01BD0H	01BD1H	00002H	_CVTSEG	DATA
01BE0H	01BE5H	00006H	_SCNSEG	DATA
01BE6H	01BE7H	00002H	_CRTSEG	DATA
01BF0H	01C37H	00048H	_BSS	BSS
01C38H	01C38H	00000H	_BSSEND	BSSEND
01C40H	01E3FH	00200H	_STACK	STACK
01E40H	01E40H	00000H	ROMDATA	ROMDATA

3

[redacted]

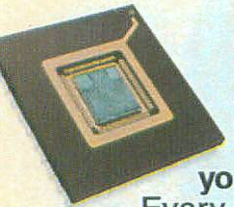
Table 2

Egg timer locate map.

Name	Class	Address	Length
_TEXT	CODE	FC000H	190EH
_ETEXT	CODEEND	FD910H	0001H
_DATA	DATA	00400H	02AAH
_CVTSEG	DATA	006B0H	0002H
_SCNSEG	DATA	006C0H	0006H
_CRTSEG	DATA	006C6H	0002H
_BSS	BSS	006D0H	0048H
_BSSEND	BSSEND	00718H	0000H
_STACK	STACK	00720H	0200H
ROMDATA	ROMDATA	FD920H	0000H
_DATA	ROMDATA	FD920H	02AAH
_CVTSEG	ROMDATA	FDBD0H	0002H
_SCNSEG	ROMDATA	FDBE0H	0006H
_CRTSEG	ROMDATA	FDBE6H	0002H
??BOOT	(ABSOLUTE)	FFFF0H	0005H

Get hardware and software working together...

...and you'll rea
levels you've ne



Harris Introduces Real Time Express™

The first microcontroller optimized for real time, will put your systems on the fast track!

Every designer is looking for a faster system. But the special demands of real-time applications require you to look beyond MIPS—to predictability, repeatability and responsiveness. Because in a real-time world, a late answer is a wrong answer.

Now there's a solution addressing all your performance parameters: our Real Time Express (RTX™) family. Combining the integration of a microcontroller and the speed of a RISC processor.

Real-Time Tradeoffs Before RTX™

Traditional microprocessors sacrifice predictability and external response to achieve high instruction-execution speeds, and they can't switch tasks quickly with minimum overhead.

Traditional microprocessors lack flexible partitioning between hardware and software to meet critical timing requirements. You can't easily extend their architectures to accommodate application-specific needs, either.

Real-Time Software: Hard Without RTX™

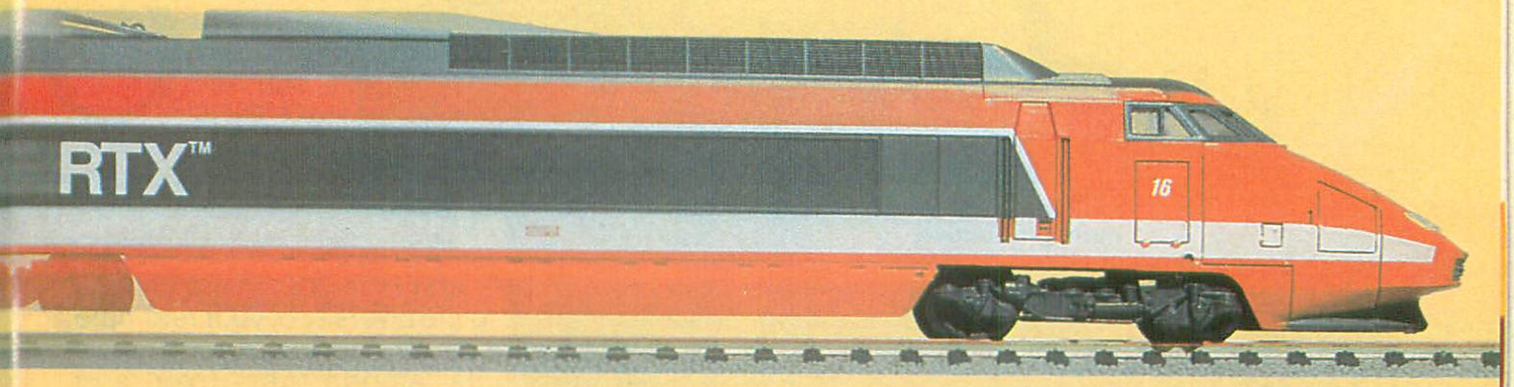
Today's real-time software environment restricts designers' productivity. They have no choice but to mix high-level and assembly language — sometimes microcode, too — during program development. To achieve real-time performance, they must program and debug the most complex tasks at the lowest level. The result: long development cycles, difficult debugging and high maintenance costs.

RTX 2000™ Performance

Processor Clock Speed (MHz)	Typical Instruction Rate (MIPS)*	Power Dissipation (mW)	Interrupt Latency (µs)	Conditional Branch (µs)	16 x 16 Multiply (µs)	ASIC Bus™ Bandwidth (Mbytes/Sec)	Subroutine Call/Return Overhead (µs)
10	15.0	400	0.4	0.10	0.10	20	0.10
8	12.0	320	0.5	0.12	0.12	16	0.12
5	7.5	200	0.8	0.20	0.20	10	0.20
1	1.5	40	4.0	1.00	1.00	2	1.00

*Instruction Rate Measured In Millions Of Instructions Per Second.

ch performance
ver dreamed of.

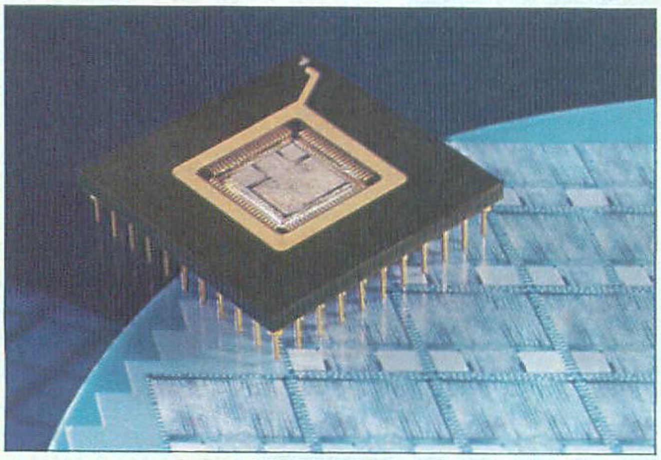


The RTX™ Solution: No More Performance Penalties

The Real Time Express™ family of general purpose, application-specific and semicustom products offers you microcontroller integration and RISC processor speed — for a fraction of the power of conventional designs.

It achieves performance through simplicity and parallelism, using an innovative dual-stack Quad Bus™ architecture with no caches or pipelines.

RTX™ frees you to program your entire application in a structured, high-level language (C, Forth, Prolog) without traditional performance penalties.



The Little Engine That Does: Based on proven macro cells, the Harris RTX is fully integrated into a CAD system, adding flexibility and reducing risk in semicustom designs.

RTX 2000/2001™ Features

Feature	RTX 2000	RTX 2001
Timer/Counters (on-chip)	3 ea. — 16 bit	3 ea. — 16 bit
Interrupt Controller (on-chip)	NMI and 13 Maskable (Expandable off-chip)	NMI and 13 Maskable (Expandable off-chip)
Memory (on-chip)	256 x 16 Parameter Stack 256 x 21 Return Stack	64 x 16 Parameter Stack 64 x 21 Return Stack
Wait State Controller (on-chip)	Yes	Yes
ASIC Bus™	16 bit	16 bit
16 x 16 Multiply	Single Cycle	20 Cycle
32/16 Divide	21 Cycle	21 Cycle
32/16 Square Root	25 Cycle	25 Cycle
Address Range	1 Mbyte	1 Mbyte
Data Bus	16 bit	16 bit
Byte Operations	Yes	Yes
Package	84 pins	84 pins
883C Compliant	1Q CY89	Not Planned
Semicustom Migration Path	Yes	Yes
Rad-Hard Migration Path	Yes	Not Planned
32 bit Migration Path	Yes	Yes
Development System	Yes	Yes
Forth Compiler	Now	Now
C Compiler	4Q CY88	4Q CY88
Prolog Compiler	1Q CY89	1Q CY89

A better hardware-to-software balance can make you 10 times more productive.

You'll boost productivity by debugging interactively — at full speed — with full symbolic debug support. Powerful debugging tools you can use on a low cost PC.

Now you can integrate hardware and software, and debug without investing in costly, complex In-Circuit Emulators (ICE).

It's everything you've wanted in a real-time microcontroller — rapid interrupt response, predictable timing, fast context switch, hardware extensibility (via a unique ASIC Bus™). And our 16-bit RTX 2000™ and RTX 2001™ are just the start. To respond to the diversity of your real-time applications, we'll be announcing a broad family of RTX™ products, among them fixed and floating point versions, and a 32-bit model.

Find out more about how you can move more of your hardware into software — and get them working together like never before.

Contact us for technical briefs or to reserve a spot at an RTX™ seminar near you.

In U.S.: 1-800-4-HARRIS Ext. 1288 (literature)
Ext. 1299 (seminars)
In Canada: 1-800-344-2444 Ext. 1288 (literature)
Ext. 1299 (seminars)

Sales Offices

U.S. HEADQUARTERS
Harris Semiconductor
2401 Palm Bay Road
Palm Bay, Florida 32905

EUROPEAN HEADQUARTERS
Harris Systems Ltd.
Semiconductor Sector
Eskdale Road
Winnersh Triangle
Wokingham RG11 5TR
Berkshire
United Kingdom
TEL: 0734-698787

FAR EAST HEADQUARTERS
Harris K.K.
Shinjuku NS Bldg. Box 6153
2-4-1 Nishi-Shinjuku
Shinjuku-Ku, Tokyo 16 Japan
TEL: 81-3-345-8911

DISTRIBUTORS IN U.S.A.
Anthem Electronics
Falcon Electronics
Hall-Mark Electronics
Hamilton/Avnet Corporation
Schweber Electronics

DISTRIBUTORS IN CANADA
Hamilton/Avnet Corporation
Semad Electronics

IN REAL-TIME CONTROL,
THE NAME IS
HARRIS
Harris Semiconductor: Analog - CMOS Digital
Gallium Arsenide - Semicustom - Custom
HARRIS

Reorder Number: 6AD-5030
©Harris Corporation, July 1988
Printed in U.S.A.

Real Time Express, RTX, RTX 2000, RTX 2001, ASIC Bus, and Quad Bus are trademarks of Harris Corporation.

CIRCLE #225 ON READER SERVICE CARD

BY IAN ASHDOWN

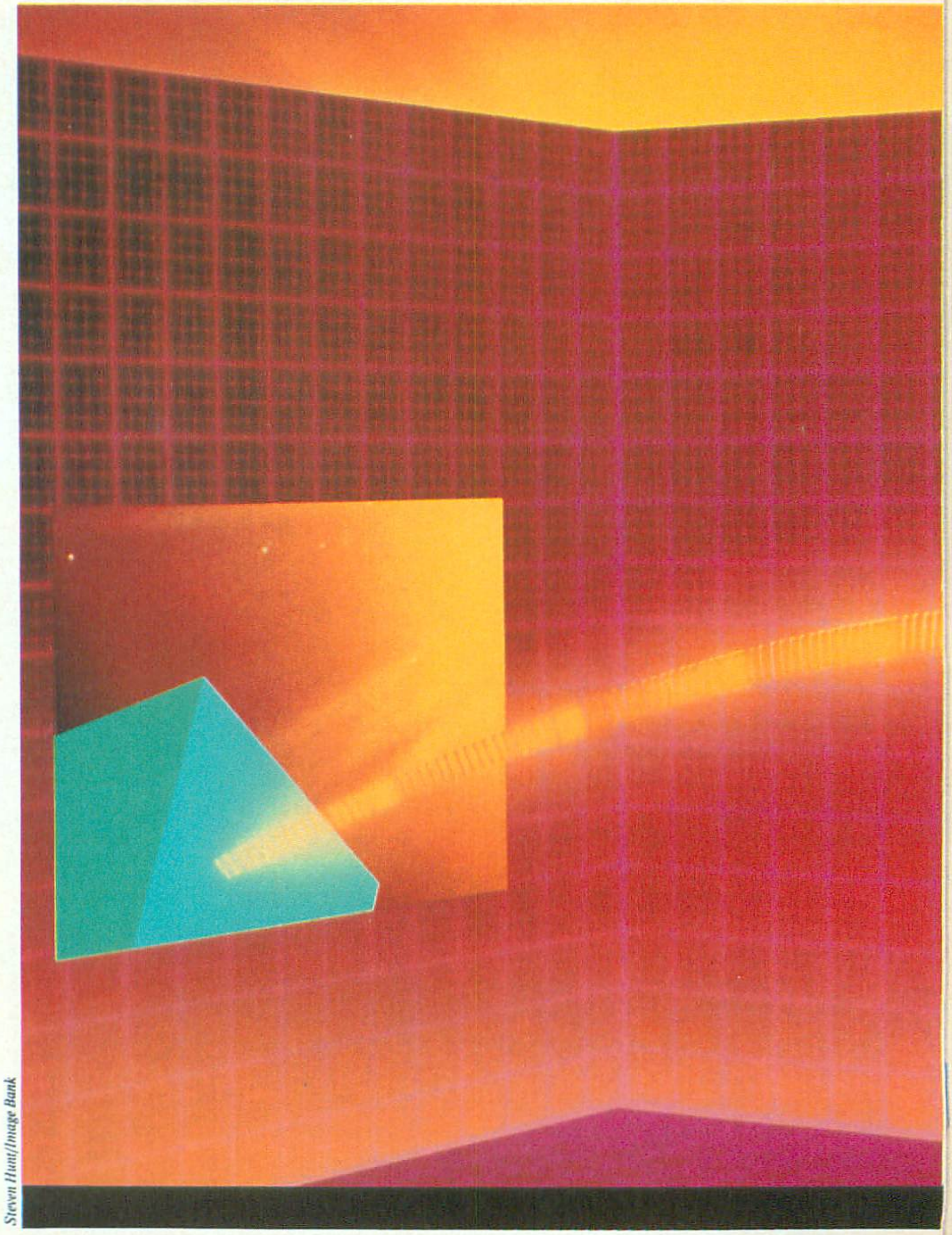
Minimizing Finite State Machines

One of the simplest concepts in computer science is that of the deterministic finite state machine. A DFSM consists of nothing more than a set of states, a set of events, and a state transition table (Figure 1A). If the machine is in state x and event a occurs, it makes a transition to the state specified in the {x,a} entry of the table. Some of the states may be final, where the machine produces some output when the state is entered, while the rest—the nonfinal states—produce none.

The simplicity of this concept belies its usefulness. Finite state machines are commonly used in text editors, document search programs, and compilers, performing such functions as regular expression searching and lexical analysis. They may be found in operating systems, communications and networking software, industrial process control systems, and robotics. DFSMs are also found in some artificial intelligence applications, such as expert systems and pattern matching, and are built into the firmware of advanced logic circuits. For all this, they're easy to simulate in software and easy to debug and maintain.

As simple as a DFSM is, there's something even simpler: its minimal equivalent (Figure 1B). For any finite state machine there's an equivalent DFSM that has a minimum number of states. Given any possible sequence of events as input, the output from the machines will be identical.

The existence of a minimal equivalent DFSM can be important, especially for applications involving machines with hundreds of states and events that must operate in real time or be embedded in a system with limited memory. However, knowing that there's a minimal equivalent DFSM for any given machine isn't the same as knowing how to find it. Trying to minimize even a simple DFSM by trial and error will quickly convince you



Steven Hunt/Image Bank

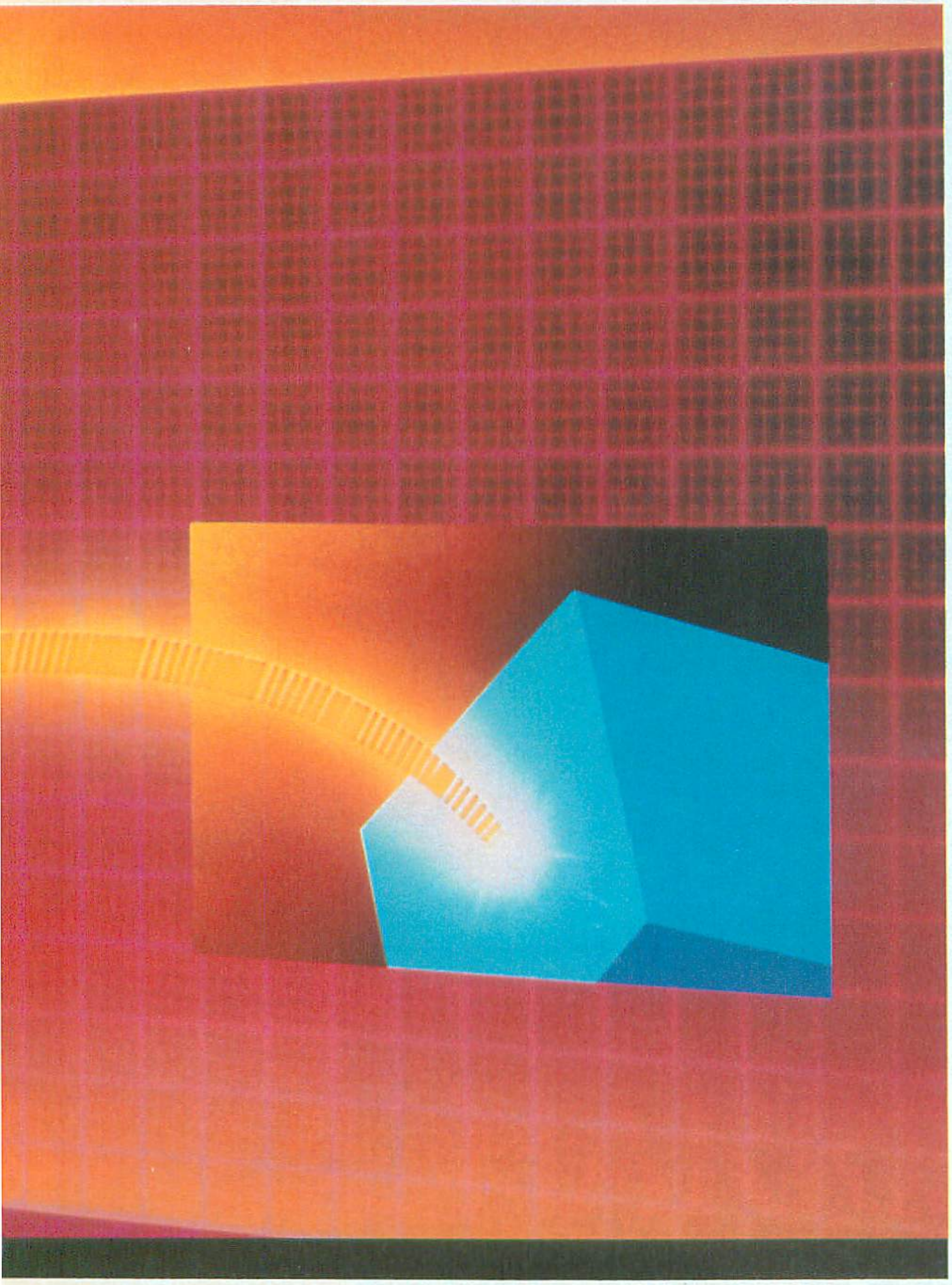
Minimizing Finite State Machines

that an automated method is required.

Fortunately, there are several algorithms available that do just this. They vary in complexity and efficiency, with the most complex—Hopcroft's Partitioning Algorithm—also being the most efficient for large DFSMs.

DEVELOPING THE ALGORITHM

You might ask why it's necessary to present the theory behind Hopcroft's algorithm. The formal mathematical proofs are available, so why not simply present the pseudocode along with suggestions for implementing it?



The answer is twofold. First, while the algorithm may be easy to implement, the way it works isn't at all obvious. If you're to have any confidence in your implementation, you must understand the underlying theory. Second, the mathematical proofs that have been published aren't easy to follow. As noted by D. Gries in his article "Describing an Algorithm by Hopcroft" (*Acta Informatica* 2: 1973), "the algorithm, its proof of correctness and the proof of running time are all very difficult to understand...[the reader] is challenged to first read Hopcroft's original paper and see whether he can understand it easily."

Surprisingly, this is no cause for concern. Shorn of its mathematics, the theory is really quite simple and intuitively logical. The few concepts to be grasped can be illustrated nicely with simple sketches of circles and arrows.

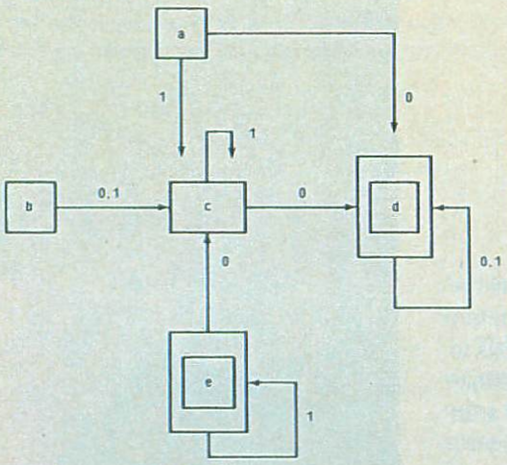
Let's begin with the basics of DFSMs. Each DFSM has a finite number of states, with each state having one transition to a new state (possibly itself) for each of a finite number of events. Also, no state can have a transition to another state without an event. Those states producing some output when they're entered are final states; all others are nonfinal. These conditions define a DFSM.

There are two possible relations between states: equivalent and distinguishable. Any two states of a DFSM are equivalent only if for every possible sequence of events the machine produces exactly the same sequence of outputs, regardless of which of the two states is used as the starting state. If there's any sequence of events for which this isn't true, then the two states are distinguishable.

Again, these are relations between states, not attributes of the states themselves. A state can be distinguishable from a number of states and, at the same time, equivalent to one or more others.

Figure 1A

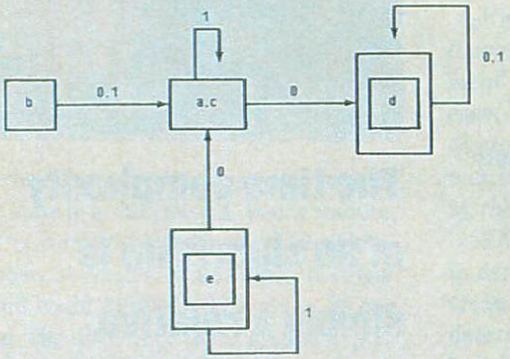
State transition graph and table for original DFSM to be reduced.



State	Type	Transition 0	Transition 1
a	NONFL	d	c
b	NONFL	c	c
c	NONFL	d	c
d	FINAL	d	d
e	FINAL	c	e

Figure 1B

State transition graph and table for minimal equivalent DFSM.



State	Type	Transition 0	Transition 1
e	FINAL	a.c	e
b	NONFL	a.c	a.c
a.c	NONFL	d	a.c
d	FINAL	d	d

Figure 1C

Partitioning of set of states *S*.

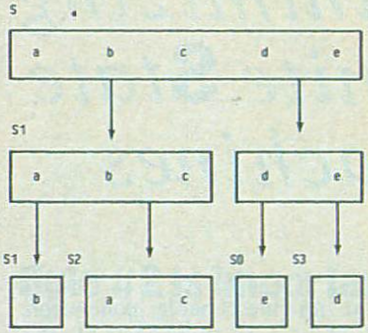
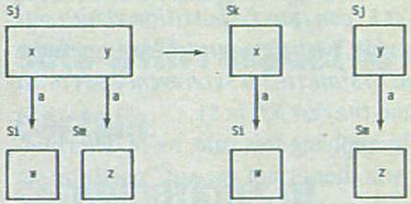


Figure 2

Rule #1: Partitioning of set *Sj* with respect to set *Si* on event *a*.



If two or more states are equivalent, they can be merged into one state in a new DFSM that will produce the same outputs as the original DFSM for any sequence of events. Distinguishable states, on the other hand, can't be merged.

To illustrate the point, consider the DFSM shown in Figure 1A. In general, it's difficult to determine whether or not two states of a DFSM are equivalent simply by inspecting the state transition graph. In this case, however, it's easy to see that states a and c are equivalent and can be merged into one state, a.c (as shown in Figure 1B). The two machines produce exactly the same output for any possible sequence of events, regardless of whether you start in state a or c of Figure 1A or state a.c of Figure 1B.

If we could find all the equivalent states of a DFSM and merge them as appropriate, we would end up with its minimal equivalent. Our task is to develop an algorithm that can systematically identify these equivalent states.

Turning this problem around, let's consider all the states of a DFSM as being initially in one set. We want to partition this set into successively smaller sets so that each set ultimately consists of only equivalent states (possibly with only one state in a set) and each set represents a state of the minimal equivalent DFSM. Distinguishable states will, of course, always be in separate sets.

Looking again at Figure 1A, our initial set would be {a,b,c,d,e}. After fully partitioning this set (see Figure 1C), we would have four sets, {a,c}, {b}, {d}, and {e}, each of which represents a state of the DFSM shown in Figure 1B.

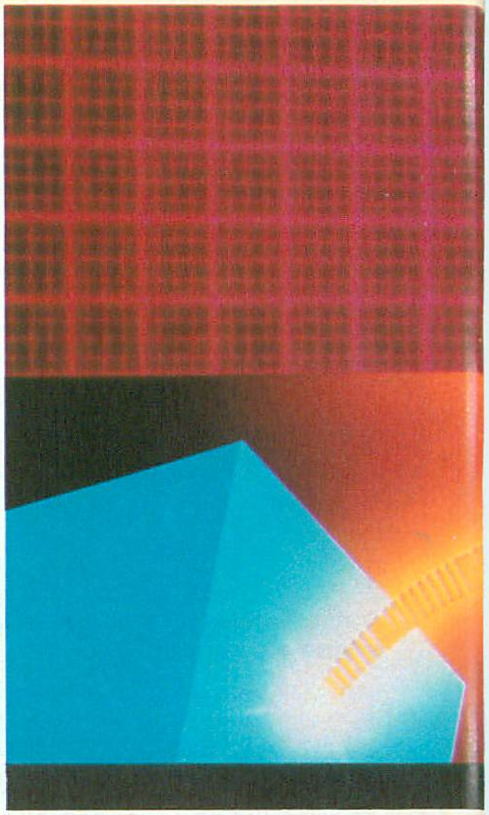
How do we decide how to partition a set? Well, our definition of state equivalence clearly says that final and nonfinal states can't be equivalent. Starting in a final state immediately produces some output, while starting in a nonfinal state doesn't. We can therefore start by partitioning the DFSM's initial set of states, S, into two sets, one (S0) containing the final states and the other (S1) the nonfinal states. As shown in Figure 1C, our DFSM's initial set would be partitioned into {d,e} for set S0 and {a,b,c} for set S1.

THE FIRST RULE

If we stop and think about it, we partitioned the initial set by determining which states were clearly not equivalent and separating them into two sets. What we now need is a rule whereby we can continue to determine which states in a set are distinguishable, then partition that set accordingly. By applying this rule to each new set we generate, we'll eventually have only equivalent states in each set and will have solved our problem.

Consider for a moment state e in set S0. It has a transition on event 0 to state c in set S1. However, state d, which is also in set S0, has a transition on event 0 to itself in set S0. Since we know that a state in a DFSM can't have a transition to two states on the same event and that states in separate sets must be distinguishable, we

Minimizing Finite State Machines



arrive at an inescapable conclusion: states d and e must be distinguishable. Set S0 can therefore be partitioned into sets S0 and S2 (Figure 1C). We can generalize this observation to form our first rule (illustrated in Figure 2):

Rule #1: Given any two sets Si *and* Sj, *examine each state within* Sj. *If, for any given event* a, *state* x *in* Sj *has a transition to a state in* Si *while state* y *in* Sj *does not, then state* x *is distinguishable from state* y. *Set* Sj *can thus be partitioned into sets* Sj *and* Sk, *with those states having transitions to states in set* Si *on event* a *put in set* Sk *and the rest left in* Sj.

In applying this rule, we're determining whether or not we can "partition set Sj with respect to set Si on event a," or, more succinctly, partition Sj wrt {Si,a}. (Note that Si and Sj can refer to the same set, so that a set can be partitioned with respect to itself on a given event.)

Rule #1 doesn't identify all distinguishable states in set Sj each time it's applied; rather, it merely determines which states in Sj are distinguishable with respect to set Si on event a. If there are none, then we can't partition Sj, at least with respect to {Si,a}.

Nevertheless, once we've determined whether or not a set can be partitioned and have done so if possible, we simply say that the set has been partitioned. While grammatically suspect, this convention simplifies the discussion that follows without introducing any ambiguities.

We can now develop a simple algorithm. Given a DFSM, we consider all of its states to be in one set, which we call S. We partition S into two sets, S0 and S1, where S0 consists of all the final states and S1, the nonfinal states. We then continue to partition sets by applying Rule #1 until we no longer have any sets with distinguishable states in them.

Expressing this in a more symbolic form, we have the algorithm in Listing 1, where move(x,a) denotes the state to which state x has a transition on event a. This algorithm is extremely simple, but it's also quite inefficient because it has to repeatedly check every possible combination of sets Si and Sj for every event a. In general, this repetition isn't necessary.

What we want is some means of identifying those set-event pairs, {Si,a}, that we *know in advance* will not result in any other sets' being partitioned. As it turns out, this not only is possible but has significant benefits. Our current algorithm has an $O(m*n^2)$ time complexity, where m is the number of events and n is the number of states. Hopcroft's Partitioning Algorithm, on the other hand, has a time complexity of O(m*n*log(n)) (where the logarithm is to base 2).

Listing 1

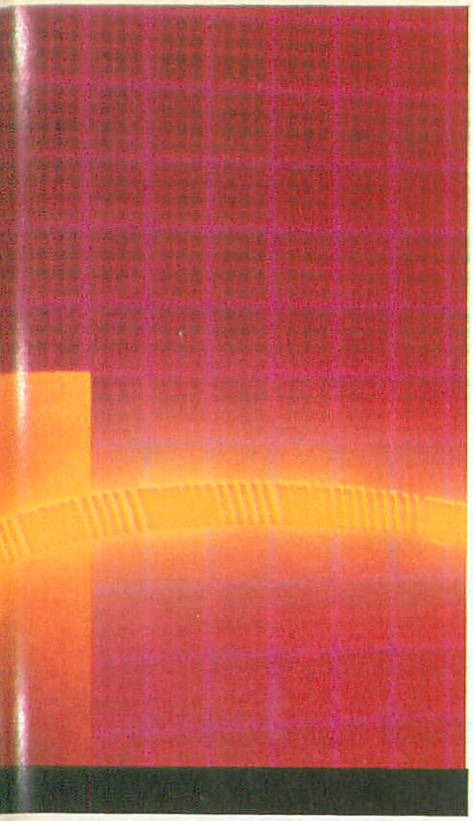
Applying Rule #1 to partitioning of sets.

```
partition S into S0 (final) and S1 (nonfinal)
WHILE there are sets Si and Sj and an event a
      such that states x and y are in Sj and
      state move(x,a) is in Si but state
      move(y,a) is not in Si
{
  determine partitionings of all sets wrt {Si,a}
  partition all indicated sets
}
```

The time complexity of an algorithm is simply a relative measure of its worst-case running time with respect to the quantity of input data. For example, if an implementation of an algorithm takes c*n seconds to process n items of data, where c is a constant, we say the algorithm has an O(n) time complexity. If it takes $c*n^2$ seconds to process n items of data, we say the algorithm has an $O(n^2)$ time complexity. (The constant factor c is dependent on the implementation of the algorithm, not the algorithm itself, and is thus ignored.)

To illustrate the significance of a time complexity of $O(m*n^2)$ versus O(m*n*log(n)), let's assume that, for some number

The time complexity of an algorithm is simply a relative measure of its worst-case running time with respect to the quantity of input data. For example, if an algorithm takes c*n seconds to process n items of data, we say the algorithm has an O(n) time complexity.



of events m, both algorithms can process one state in a millisecond. In one minute, our current algorithm would be able to process 244 states, while Hopcroft's algorithm could handle 4,893 states. In one hour, the difference is far greater: 1,897 states versus 204,094. Need I say more?

IMPROVING THE ALGORITHM

Now that we have an incentive to continue, let's consider partitioning once again by looking at Figure 2. We've just partitioned set Sj with respect to set Si on event a into sets Sj and Sk. This means that none of the states in Sj have transitions to any state in Si on event a, while all the states in Sk do.

If we now partition set Sk with respect to some other set and event into sets Sk and Sl, there's no point in reapplying Rule #1 to these new sets with respect to Si on event a. As noted above, all their states have transitions to states in set Si on event a (see Figure 3). Applying Rule #1 would reveal no distinguishable states. We can generalize this observation to form our second rule.

Rule #2: If all sets have been partitioned with respect to set Si *on event* a *(as determined by Rule #1), then there's no need to determine the partitioning of any future set with respect to set* Si *on event* a *(where* future set *refers to all existing sets and any new sets that may be created by partitioning these sets later on).*

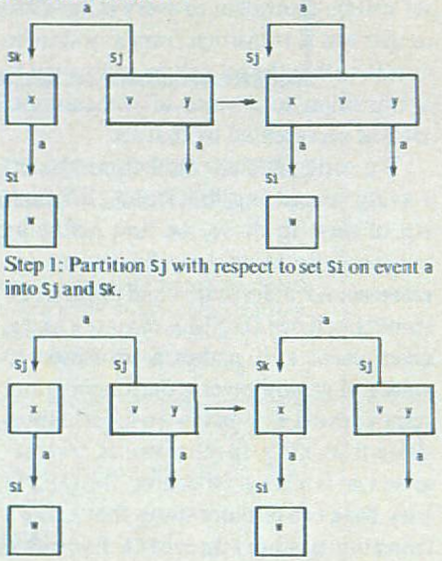
This rule is important to us. With it, we can state that we don't have to determine the partitioning of all sets with respect to certain combinations of sets and events. Before we incorporate the rule into our algorithm, however, two more rules await discussion. The first can be simply put:

Rule #3: Suppose set Sj *has been partitioned with respect to some set on some event into sets* Sj *and* Sk*. Then, for any event* a*, we need to determine the partitioning of all future sets with respect to only two of the three sets (the original set* Sj*, the new set* Sj*, and set* Sk*).*

The basic idea behind Rule #3 is this: in determining the partitioning of some set Sm with respect to Sj on event a, we're determining whether or not each state in Sm has a transition on event a to a state in Sj. If we now partition Sj into Sj and Sk, then determine the partitioning of Sm with respect to, say, the new Sj on event a, we are in fact determining whether each state in Sm has a transition on event a to a state in the new Sj. However, once we've done this for each state in Sm, we also know whether or not it has a transition on event a to a state in Sk and need not determine the partitioning of any set with respect to Sk on event a. The same argument applies for any two of the three sets, as Figure 4 illustrates.

Figure 3

Rule #2: Sets partitioned with respect to set *Si* on event *a*.



Step 1: Partition Sj with respect to set Si on event a into Sj and Sk.

Step 2: No need to partition new Sj or Sk with respect to set Si on event a; no distinguishable states will be revealed.

Rule #2 is important to us. With it, we can state that we don't have to determine the partitioning of all sets with respect to certain combinations of sets and events.

There are two cases to consider here. Suppose we've just partitioned some set Sj into Sj and Sk but have yet to determine the partitioning of all sets with respect to the original set Sj for some event a. Rule #3 says we can now determine the partitioning of all sets with respect to (the new) {Sj,a} and {Sk,a} instead.

On the other hand, say we had partitioned all sets with respect to Sj on event a before we partitioned Sj into Sj and Sk. Rule #3 says we now need to determine the partitioning of all sets with respect to the new {Sj,a} or {Sk,a}, but not both.

For our fourth and final rule, we re-

Minimizing Finite State Machines

turn to our original set of states, S. We partitioned S into two sets, the final states (S0) and nonfinal states (S1). We already know that for any state x and any event a, the state move(x,a) must be in the parent set, S, so there's no need to determine the partitioning of any sets with respect to S. Rule #3 then says we need to partition all sets with respect to either S0 or S1 on event a, but not both.

Rule #4: For any event a*, we must determine the partitioning of all sets with respect to only one of the two sets (*S0 *and* S1*).*

Another interesting thing about Rules #3 and #4 is that they say nothing about the size of the sets involved. Since determining the partitioning of a set with respect to a set Si will require us to examine each state in Si, we can save ourselves some work by always choosing the smaller of the sets by which to partition.

Rule #3 and Rule #4 don't specify the size of the sets. We can save ourselves some work by choosing the smaller set.

ALMOST THERE

These four rules are sufficient to create an O(m*n*log(n)) algorithm. We now need to pull them together into a workable implementation. We'll use a list, L, to keep track of all the set-event pairs, {Si,a}, with respect to which all sets must be partitioned. Our rules enable us to determine which pairs won't be put on this list. We can then continue partitioning until the list is empty.

How does the list become empty? By Rule #2, when we partition all sets with respect to pair {Si,a}, we no longer have to consider that pair and can remove it from the list. Eventually all pairs will be removed and the algorithm will terminate.

We must also add to the list whenever a set is partitioned. As noted for Rule #3, there are two cases to consider. Suppose Sj has been partitioned into Sj and Sk. Now if, for some event a, {Sj,a} is still on the list, then by Rule #3 we need to determine the partitioning of all sets with respect to the two pairs, {Sj,a} and {Sk,a}. This means that we must add {Sk,a} to list L. On the other hand, if {Sj,a} isn't on the list (i.e., it was considered and removed), then we need to add the smaller of {Sj,a} and {Sk,a} to the list.

We can now present a major refinement of our algorithm (Listing 2). When this algorithm finishes, each set will represent one state of the equivalent minimal finite state machine. Every state within a set will be equivalent to every other state in that set. A transition from any state to a state within the set will be equivalent to a transition to a state of the minimal DFSM represented by that set.

We still have to resolve our handwaving concerning the actual partitioning of sets. To do so, we first define an inverse state transition, denoted as inverse-move(x,a) for state x and event a. Assume that if our DFSM is in state y and is given event a, it makes a transition to state x. If we now reverse the machine and remove event a, it makes an inverse transition from x to y. In other words, inverse-move(x,a) is state y. Of course, the DFSM may have two or more states that make a transition to state x on event a. By reversing the machine, we can see that each of these states is an inverse transition from state x on event a.

When the algorithm in Listing 2 finishes, each set will represent one state of the equivalent minimal finite state machine, with every state equivalent to all other states in that set.

Listing 2

Refinement of algorithm in Listing 1; each set now represents a state of the finite state machine.

```
partition S into S0 (final) and S1 (nonfinal)
Si <- smaller of S0 and S1  /* by Rule #4 */
L <- NULL  /* list L is initially empty */
FOR each event a
  add pair {Si,a} to list L
WHILE pairs in list L
{
  remove one pair {Si,a} from L  /* by Rule #2 */
  determine partitionings of all sets wrt {Si,a}
  partition all indicated sets  /* by Rule #1 */
  FOR each Sj just partitioned into Sj and Sk
    FOR each event a
      IF {Sj,a} is in L  /* by Rule #3 */
        add {Sk,a} to L
      ELSE
        add smaller of {Sj,a} and {Sk,a} to L
}
```

We can use this idea of an inverse transition to determine which sets can be partitioned with respect to pair {Si,a}. This is easily done by maintaining a list, D, of the states to be moved to new sets. List D should hold those states (y) whose state

Unbelievable!

SOURCER™

- SEE HOW PROGRAMS WORK
- EASILY MODIFY PROGRAMS

SOURCER™ creates detailed commented source code and listings from memory and executable files. Built in data analyzer and simulator resolves data across multiple segments and provides detailed comments on interrupts and subfunctions, I/O ports and much more. Determines necessary assembler directives for reassembly. Includes a definition file facility to include your own remarks and descriptive labels, force data types, and more. Complete support for 8088/87 through 80286/287 and V20/V30 instruction sets. We welcome comparisons with any other product, because no product comes close to the ease of use and output clarity of SOURCER.

On my list of programs that I simply won't do without!
—Robert Hummel, Senior Technical Editor, PC Magazine

SAMPLE OUTPUT

Fully automatic

Program header

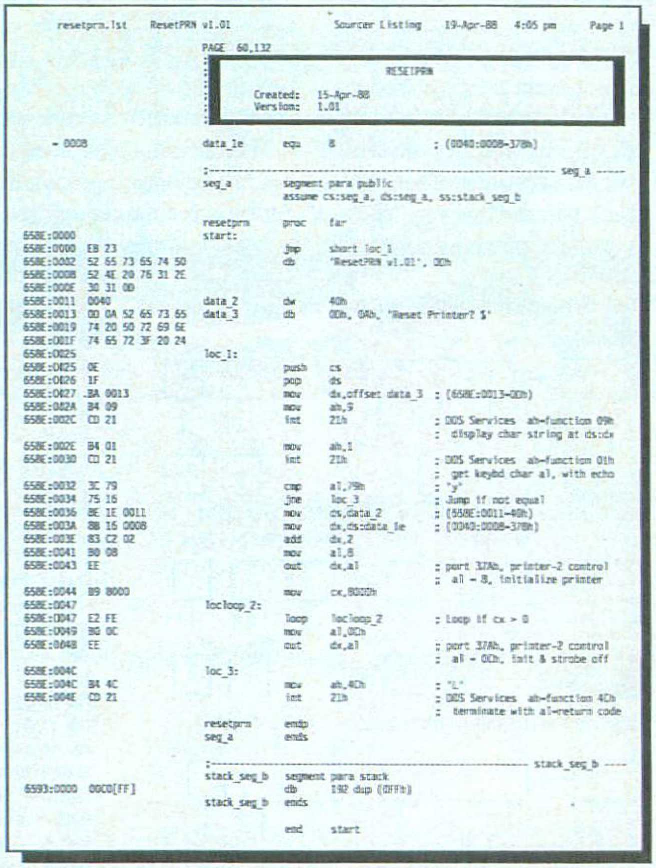
Assembler directives

Determines data areas and type

Detailed comments

Simulator follows segment changes

Easy to read format



```
resetprn.lst   ResetPRN v1.01                 Sourcer Listing   19-Apr-88   4:05 pm    Page 1

                             PAGE  60,132
                             :
                             :                        RESETPRN
                             :
                             :            Created:    15-Apr-88
                             :            Version:    1.01
                             :

  = 0008                     data_le       equ    8                      ; (0040:0008=378h)

                             ;------------------------------------------------ seg_a -----
                             seg_a         segment para public
                                           assume cs:seg_a, ds:seg_a, ss:stack_seg_b

                             resetprn      proc   far
658E:0000                    start:
658E:0000  EB 23                           jmp    short loc_1
658E:0002  52 65 73 65 74 50               db     'ResetPRN v1.01', 00h
658E:0008  52 4E 20 76 31 2E
658E:000E  30 31 00
658E:0011  0040              data_2        dw     40h
658E:0013  0D 0A 52 65 73 65 data_3        db     0Dh, 0Ah, 'Reset Printer? $'
658E:0019  74 20 50 72 69 6E
658E:001F  74 65 72 3F 20 24
658E:0025                    loc_1:
658E:0025  0E                              push   cs
658E:0026  1F                              pop    ds
658E:0027  BA 0013                         mov    dx,offset data_3  ; (658E:0013=0Dh)
658E:002A  B4 09                           mov    ah,9
658E:002C  CD 21                           int    21h               ; DOS Services  ah=function 09h
                                                                    ;  display char string at ds:dx
658E:002E  B4 01                           mov    ah,1
658E:0030  CD 21                           int    21h               ; DOS Services  ah=function 01h
                                                                    ;  get keybd char al, with echo
658E:0032  3C 79                           cmp    al,79h            ; 'y'
658E:0034  75 16                           jne    loc_3             ; Jump if not equal
658E:0036  8E 1E 0011                      mov    ds,data_2         ; (658E:0011=40h)
658E:003A  8B 16 0008                      mov    dx,ds:data_le     ; (0040:0008=378h)
658E:003E  83 C2 02                        add    dx,2
658E:0041  B0 08                           mov    al,8
658E:0043  EE                              out    dx,al             ; port 37Ah, printer-2 control
                                                                    ;  al = 8, initialize printer
658E:0044  B9 8000                         mov    cx,8000h
658E:0047                    locloop_2:
658E:0047  E2 FE                           loop   locloop_2         ; Loop if cx > 0
658E:0049  B0 0C                           mov    al,0Ch
658E:004B  EE                              out    dx,al             ; port 37Ah, printer-2 control
                                                                    ;  al = 0Ch, init & strobe off
658E:004C                    loc_3:
658E:004C  B4 4C                           mov    ah,4Ch            ; 'L'
658E:004E  CD 21                           int    21h               ; DOS Services  ah=function 4Ch
                                                                    ;  terminate with al=return code
                             resetprn      endp
                             seg_a         ends

                             ;------------------------------------------------ stack_seg_b ----
                             stack_seg_b   segment para stack
6593:0000  00C0[FF]                        db     192 dup (0FFh)
                             stack_seg_b   ends

                                           end    start
```

(Source code output and inline cross reference can also be selected)

BIOS SOURCE

- CHANGE AND ADD FEATURES
- CLARIFY INTERFACES

for PS/2, AT, XT, PC, and Clones

The BIOS Pre-Processor™ with SOURCER provides the first means to obtain accurate legal source listings for any BIOS! Identifies entry points with full explanations. Resolves PS/2's multiple jumps for improved clarity. Provides highly descriptive labels such as "video_mode" and much more. Fully automatic.

SOURCER $99.95   BIOS Pre-Processor* $49.95   SOURCER w/BIOS Pre-Processor $139.95

USA Shipping & Handling $3; Outside USA $15; CA Residents add local sales tax 6, 6.5 or 7%; *requires SOURCER

All our products come with a 30 day money back satisfaction guarantee. Not copy protected. To order or receive additional information just call!

VISA **(800) 662-8266** MasterCard

V COMMUNICATIONS
3031 Tisch Way, Suite 905, Dept. ES, San Jose, CA 95128 (408) 296-4224

PS/2, AT, XT, and PC are trademarks of IBM Corp.

CIRCLE #226 ON READER SERVICE CARD

What do IBM, HP, Raytheon and others know about 80X86 embedded system development you should know?

PHAR LAP TOOLS.

Keeping pace in today's competitive environment starts by keeping up-to-date on the latest software innovations. Phar Lap offers you the latest solutions for ROM-based software development. Start with 386|ASM/LinkLoc. This package includes an 8086/186/286/386 assembler, a linker/locator for embedded targets, a librarian, a debugger and 386|DOS-Extender. And we also offer high level languages like C, Pascal, Fortran and others.

These tools are available for a variety of hosts including the IBM PC*, Sun and Apollo UNIX workstations, VAX* and MicroVAX* systems.

So, whether your code is written in assembler or a high level language, our tools will build programs which can be downloaded to ICEs, ROM-based debuggers or PROM programmers. Now you know what Phar Lap has done for others. Think what it can do for you.

Find out more about our set of 80X86 tools. Call or write for more information:

(617) 661-1510
PHAR LAP SOFTWARE, INC.
60 Aberdeen Ave., Cambridge, MA 02138

PHAR LAP

"THE 80386 SOFTWARE EXPERTS"

Phar Lap and 386|DOS-Extender are trademarks of Phar Lap Software, Inc.
VAX and MicroVAX are registered trademarks of Digital Equipment Corporation.
IBM PC is a trade mark of IBM Corp.

CIRCLE #227 ON READER SERVICE CARD

Minimizing Finite State Machines

transitions (move(y,a)) are in Si or, equivalently, all the inverse-move(x,a) states for each state x in Si.

By Rule #1, all these states should be moved from their current sets to the twins of those sets created by partitioning them. This means that for each set x in list D, the state Sj to which it belongs should have a new empty set (Sk) created for it (if one doesn't already exist) and state x moved to Sk. As we work our way through the states in list D, each set Sj has a twin set, Sk, created for partitioning.

Finally, there's a case where partitioning a set is unnecessary. If, for each and every state y in some set Sj, move(y,a) is in set Si, then we need not partition set Sj with respect to set Si on event a. If we did, every state would be moved to the new set, leaving the original set empty.

Incorporating these ideas into our algorithm, we get Hopcroft's Partitioning Algorithm in its final form (Listing 3). If you have trouble following the pseudocode, try reading it in conjunction with the step-through of the algorithm (Figure 5) for the DFSM shown in Figure 1A.

FINISHING TOUCHES

Half the fun (and most of the agony) of program design is in choosing the appropriate data structures for algorithms. A good example of this is list L. If we implemented it as a linked list, we could determine whether or not a given set-event pair is in the list in O(n) time by simply scanning the list. However, we can do much better than this if we use a threaded list.

Consider that our algorithm doesn't specify the order in which set-event pairs need to be added to or removed from list L. We can therefore use a stack as the simplest form of list data structure.

Also consider that no set-event pair can ever appear on list L more than once. This means that we can implement our stack as a two-dimensional array of sets and events, where each possible set-event pair is represented by an element of the array (see Figure 6).

Initially, an empty stack is represented by having all elements of the array set to EMPTY. A separate variable holds the indices of the top of the stack, which is also initially set to EMPTY.

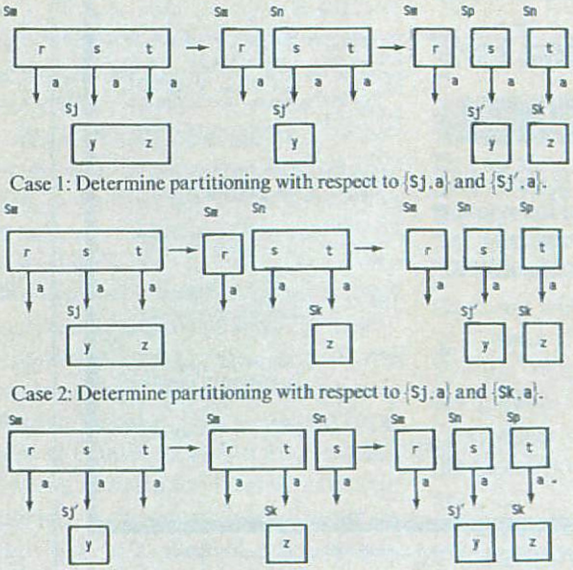
When we push the first set-event pair onto the stack, we use its set and event numbers as array indices. We place these values into the top-of-stack variable and into the array element itself. If we think of the array element as a link to another array element, this means that the link is pointing to itself—a convenient means of indicating the bottom of the stack.

As each new set-event pair is added to the stack, its array indices are placed in the top-of-stack variable and its array element is set to the array indices of the previous top-of-stack pair. In this way, each new stack element has a link to the previous element on the stack. These links can be viewed as a thread tying the elements of the stack together, hence the name *threaded list.*

Removing a set-event pair from the stack is equally simple. The top-of-stack variable holds the set number and event as array indices. The values in the indicated array element, which point to the previous set-event pair on the stack, are copied to the top-of-stack variable and the array element is set to EMPTY. This effectively pops the current pair off the stack. If the indicated array element points to itself, we know that we've emptied the stack, so we set the top-of-stack variable to EMPTY instead.

The advantage of using threaded lists becomes most apparent when we have to determine whether or not a particular

Figure 4
Rule #3: Set *Sj* partitioned into *Sj'* and *Sk*.



Case 1: Determine partitioning with respect to {Sj,a} and {Sj',a}.

Case 2: Determine partitioning with respect to {Sj,a} and {Sk,a}.

Case 3: Determine partitioning with respect to {Sj',a} and {Sk,a}.

Figure 5
Step-through.

```
S = {a,b,c,d,e}
S0 = {d,e} S1 = {a,b,c}
LIST L = {NULL}
ADD to L: {S0,0}
ADD to L: {S0,1}
REMOVE from L: {S0,1}
List D = {d,e}
REMOVE from L: {S0,0}
List D = {a,c,d}
CREATE S1's twin S2
MOVE a from S1 to twin S2
MOVE c from S1 to twin S2
CREATE S0's twin S3
MOVE d from S0 to twin S3
ADD to L: {S3,0}
ADD to L: {S3,1}
ADD to L: {S1,0}
ADD to L: {S1,1}
REMOVE from L: {S1,1}
List D = {NULL}
REMOVE from L: {S1,0}
List D = {NULL}
REMOVE from L: {S3,1}
List D = {d}
REMOVE from L: {S3,0}
List D = {a,c,d}

Result:
S0 = {e}     (FINAL)
S1 = {b}     (NONFINAL)
S2 = {a,c}   (NONFINAL)
S3 = {d}     (FINAL)
```

Figure 6
Array *SEPLST* with sets {*S1,1*}, {*S1,0*}, {*S3,1*}, and {*S3,0*} in List *L*.

EVENT	0		1	
SET	PSET	PEVT	PSET	PEVT
0	-1	-1	-1	-1
1	3	1	1	0
2	-1	-1	-1	-1
3	-1	-1	3	0

STACK_top: set = 1, event = 1

Figure 7
Array *ITARY* showing inverse transition lists.

EVENT	0		1	
STATE	HEAD	PREV	HEAD	PREV
a	-1	c	-1	b
b	-1	e	-1	c
c	b	d	a	-1
d	a	-1	d	-1
e	-1	-1	e	-1

All the Tools You Need For Motorola 680X0 From Whitesmiths

Whitesmiths, Ltd. now offers a complete set of 68K Cross Development Tools — specifically designed to work together — for the Motorola 68000 family of microprocessors. You get:

A C CROSS COMPILER
Whitesmiths' C Compilers offer the closest conformance currently available to the draft ANSI C Standard. We've added 68020 and 68881 support, and dramatically optimized code generation, so you can get the code quality you need today with the language you'll need tomorrow.

SUPPORT TOOLS
We have all the extras you need to develop embedded programs. Our powerful object utilities help you link multi-segment programs, build direct and sequential libraries, create load maps and interspersed listings, and talk to dozens of downloaders, emulators, and PROM programmers.

C SOURCE LEVEL DEBUGGING
We have the support you need to debug in terms of C functions, data types, and source lines. You debug what you write, not a lower level language.

A MICSIM SIMULATOR
You can debug your embedded programs right on your development host — our MICSIM Simulator needs no extra hardware. It's like debugging on your favorite emulator, but with no contention for dedicated resources, no download time, and with the symbolic breakpoint and trace control you've always dreamed of having.

AN XA8 CROSS ASSEMBLER
Our macro assembler is both fast and powerful, with support for 68020, 68881, and 68551.

A PASCAL COMPILER
You can program as much as you want in ISO Standard Pascal, or use the powerful extensions we've added to this production quality compiler. And you get complete integration with C and assembly language as well.

Working together, the 68K Cross Development Tools deliver both optimized performance and improved programmer productivity. Best of all, Whitesmiths offers everything you need at a very competitive price. We've been delivering and supporting high quality software development tools since 1978, and we're committed to continually enhancing our product line.

If you develop 68000 programs on a DEC VAX, an IBM PC, or a UNIX workstation, chances are we can save you time and money. For more technical details, call our toll-free number today. We also offer attractive packages for OEMs.

Whitesmiths, Ltd.
59 Power Road
Westford, MA 01886
617/692-7800

TOLL-FREE
NUMBER
1-800-225-1030

CIRCLE #228 ON READER SERVICE CARD

set-event pair is in list L. To do this, we simply check the appropriate array element. If it isn't EMPTY, then we know the pair is in the list. Nothing could be faster or simpler.

Figure 7 illustrates another use for threaded lists, this time as an array representing the inverse state transitions of the original DFSM. Each element of the array has two members, HEAD and PREV. For any state x and event a, HEAD holds the state number (y) of the head of a threaded list. The PREV member of element {y,a} is the link to the previous pair in the list. If the entry is EMPTY, it indicates that the current pair is the tail of the list.

We need to keep track of which states of the original DFSM are in which sets as the algorithm progresses. Once again, threaded lists are used as the most appropriate data structures (Figures 8A and 8B). This time, however, they're doubly linked, as we need to link and unlink states from anywhere in the lists as we move states from one set to another. For any set, array PSARY (Figure 8A) indicates the number of states in the set, the number of the set that was created as its twin, and the head of the threaded list of states in the set. Array SETMAP (Figure 8B) holds the threaded lists.

Figure 8A
Array *PSARY* showing set attributes.

SET	SIZE	HEAD	TWIN
0	1	e	3
1	1	b	2
2	2	a	-1
3	1	d	-1

Figure 8B
Array *SETMAP* showing set state lists.

STATE	SET	NEXT	PREV
a	2	c	-1
b	1	-1	-1
c	2	-1	a
d	3	-1	-1
e	0	-1	-1

Finally, we need a quick method of determining whether or not all states y in Sj have move(y,a) in Si when we're partitioning sets. This can be done by maintaining an array of counters, one for each state. (We can never have more sets than there are states in the original DFSM, of course.) We clear these counters each time we initialize list D. Then, each time we append inverse state transition y to D, we increment the counter representing the set to which y belongs. When we partition sets, we need only compare the counter for set Sj against the number of states currently in Sj. If they're equal, then there's no need to partition the set.

[redacted]

Listing 3

Hopcroft's Partitioning Algorithm in its final form.

```
/* Initialize */
partition S into S0 (final) and S1 (nonfinal)
Si <- smaller of S0 and S1  /* by Rule #4 */
L <- NULL  /* list L is initially empty */
FOR each event a
  add pair {Si,a} to list L
WHILE pairs in list L
{
  remove one pair {Si,a} from L  /* by Rule #2 */
  /* Determine partitioning of all sets
     wrt {Si,a} */
  D <- NULL  /* list D is empty */
  FOR each state x in Si
    add all inverse-move(x,a) states to list D

  /* Partition each set as just determined */
  FOR each state x on list D  /* by Rule #1 */
  {
    Sj = set number in which state x appears
    IF all states y in Sj have move(y,a) in Si
      CONTINUE  /* no need to partition set */
    ELSE
    {
      IF Sj has not been partitioned yet
        create empty set Sk  /* twin of Sj */
        move x from Sj to Sk
    }
  }
  /* Fix list L according to partitions that
     just occurred */
  FOR each Sj just partitioned into Sj and Sk
    FOR each event a
      IF {Sj,a} is in L  /* by Rule #3 */
        add {Sk,a} to L
      ELSE
        add smaller of {Sj,a} and {Sk,a} to L
}
```

AN ALTERNATIVE

While it's nice to know about Hopcroft's Partitioning Algorithm and its O(m*n*(log n)) time complexity, a simpler algorithm might be preferable. There is such an algorithm; works listed in the bibliography below explain that and Hopcroft's algorithm in detail.

[redacted]

Ian Ashdown is a senior software engineer with Glenayre Electronics, Vancouver, B.C., Canada, specializing in real-time programming for electronic voice mail systems.

Further Reading

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Reading, Mass.: Addison-Wesley, 1986.

Aho, A. V., and J. D. Ullman. *Principles of Compiler Design*. Reading, Mass.: Addison-Wesley, 1977.

Barrett, W. A., and J. D. Couch. *Compiler Construction: Theory and Practice*. Chicago: Science Research Associates, 1979.

Gries, D. "Describing an Algorithm by Hopcroft." *Acta Informatica* 2 (1973): 97-109.

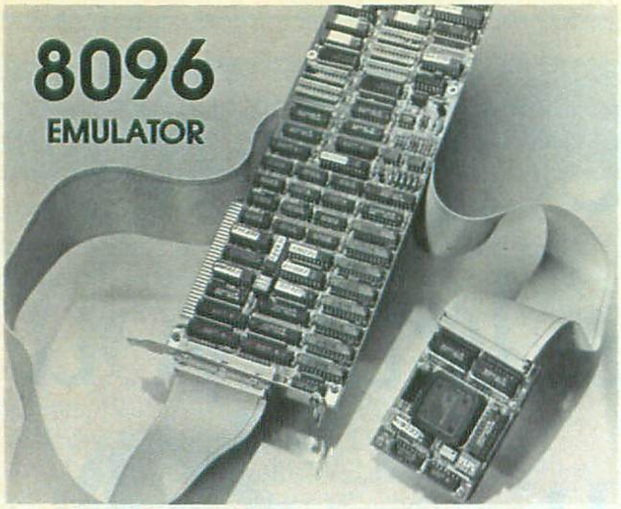
Hopcroft, J. E. "An n log n Algorithm for Minimizing States in a Finite State Automaton." *Theory of Machines and Computations*. Academic Press (1971): 189-96.

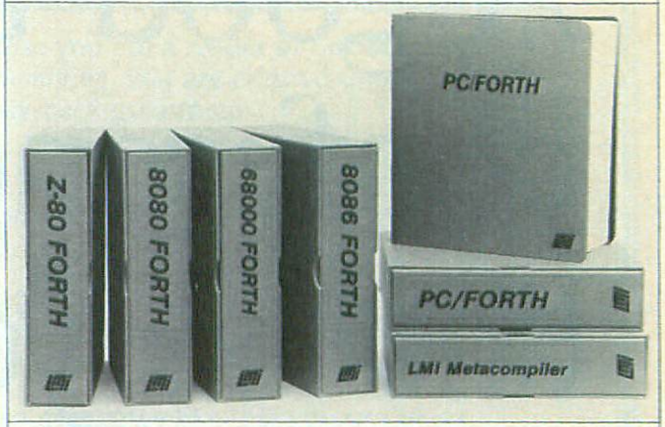
Hopcroft, J. E., and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Reading, Mass.: Addison-Wesley, 1979.

Hopkin, D., and B. Moss. *Automata*. London: MacMillan Press Ltd., 1976.

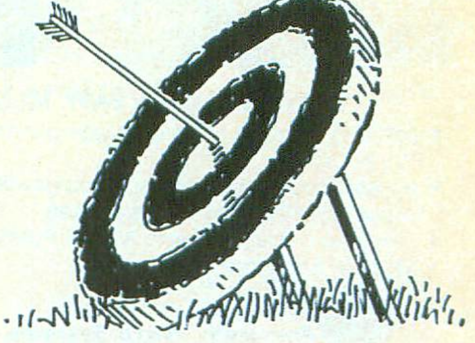
Huffman, D. A. "The Synthesis of Sequential Switching Circuits." *Journal of the Franklin Institute* 257: 161-90, 275-303.

Moore, E. F. "Gedanken Experiments on Sequential Machines." In *Automata Studies*, edited by C. Shannon and J. McCarthy, 129-53. Princeton University Press, 1956.


8096
EMULATOR


Z-80 FORTH
8080 FORTH
68000 FORTH
8086 FORTH
PC/FORTH
PC/FORTH
LMI Metacompiler

A good technical conference lets you choose your own focus.



Choose yours from over 150 classes in 3 1/2 days at SOFTWARE DEVELOPMENT '89.

Now there's a solution to an old dilemma: How do you learn tried-and-true solutions to your everyday programming problems while keeping an eye on the industry as a whole?

Find the answer at the largest technical conference designed by and for programming professionals. SOFTWARE DEVELOPMENT '89 gives you the opportunity to sample the newest ideas in every facet of programming. Presented by 90 of the brightest minds in the profession.

You'll find a base of classes wide enough to satisfy your interest in general industry trends. But you'll also find that by following one of nine different program "tracks," you can tailor a program to your particular programming specialty.

"SOFTWARE DEVELOPMENT '88 was very useful--a person can't be an expert at everything, so it was good to be able to take some introductory sessions in other areas."
--Dennis Jump THE GREAT SOFTWESTERN CO. Inc. Denton, Texas

One whole track is devoted exclusively to Embedded Systems Development. If your interest lies here, you'll greatly increase your level of practical, marketable skills in this field. Choose from courses like:

* Assembly Optimization/Memory Emulators
* Controlling Concurrency in Tasking
* Debugging Embedded Systems
* Designing for Embedded Systems
* How to Design a Real-Time System
* Marrying Software and Hardware
* Memory and Interrupt Execution in Embedded Ada
* Microcontrollers and Forth
* Next Generation CPUs
* System Developer's Workbench

If you want to explore the industry outside of your specialty, check into any of the dozens of classes in other tracks, covering subjects in Artificial Intelligence, C Programming Issues, Design Methods, Graphics, Languages, Object-Oriented Programming, and Software Tools and Issues.

Software Development '89, Miller Freeman Publications, 500 Howard St. San Francisco, CA 94105 (415) 995-2471 FAX#:(415) 543-0256

Vital Information, to Save You Vital Time

Optimize the time you need to invest in education. Compress a year's worth of learning into three full days by benefiting from the knowledge of leaders in every key facet of software development. People like:

Bill Gates, Philippe Kahn, P.J. Plauger, Bjarne Stroustrup, Ed Yourdon, Ray Duncan, Chuck Moore, Lou Mazzucchelli, Walter Bright, Robert Ward, Jim Brodie, Ken Orr, Charles Duff, Ken Iverson, Terry Winograd, Richard Finkelstein and Earl Sacerdoti

And you'll have time to:

* Meet with your peers in informal receptions and evening round table discussions.

* Visit the largest trade show for the industry: the SOFTWARE DEVELOPMENT '89 Development Products Exhibition.

* Meet with the people behind the products at evening vendor panel discussions.

The sponsors of SD '89 are dedicated to bringing you up-to-the-minute information in software development. They're the same people who bring you *AI Expert, Computer Language, Database Programming and Design, Embedded Systems Programming*, and *UNIX Review*, so you can be sure course topics are fresh and practical.

Save $100 And Make Sure You Get The Classes You Want.

While the regular fee for the full three and a half days of SOFTWARE DEVELOPMENT '89 is $695, by registering now (before December 31st) you'll receive a $100 discount. Register two or more people from the same organization and you'll get an additional discount of $25 each.

At last year's conference many sessions were filled up months ahead of time. So in addition to the important savings in the registration fee, you'll guarantee getting the classes you want by registering today.

Special No-Risk Guarantee

Because you are a reader of *Embedded Systems Programming*, you are entitled to our first-ever money-back guarantee:

If, after attending, you feel SOFTWARE DEVELOPMENT '89 did not give you the practical information you came for, we'll refund your fee. No questions asked.

Register now — take the discount — then if you need to cancel for any reason before Jan. 30, 1989 you will receive a full 100% refund.

With a guarantee like this you can't lose.

SOFTWARE
DEVELOPMENT '89

February 14-17, 1989
Hyatt Regency Burlingame
San Francisco Peninsula, CA

Clip and mail the coupon today.
Or to register instantly, call Bill Rutledge or Anastasia Mills at (415) 995-2471.

PLEASE CUT HERE

□ **Yes. Please register me to attend Software Devlopment '89**

□ Not yet, but I'm interested — Please send me a complete course description catalog

□ Please send me information on exhibiting

Name: __________________

Title: __________________

Company: __________________

Address: __________________

City, State, Zip: __________________

Phone: __________________

Registrant fee: $695. Pay only $595 if payment is sent before 12/31/88. Deduct $25 each for two or more registrants.

□ Bill me □ Payment enclosed __________________

□ Visa □ MC □ Amex # __________________

expires: ____ Signature: __________________

Software Development '89, Miller Freeman Publications, 500 Howard St.
San Francisco, CA 94105, (415) 995-2471 FAX#: (415) 543-0256

Priority Code:(ES12)

CIRCLE #232 ON READER SERVICE CARD

by John M. Dlugosz

Debugging Without ICE

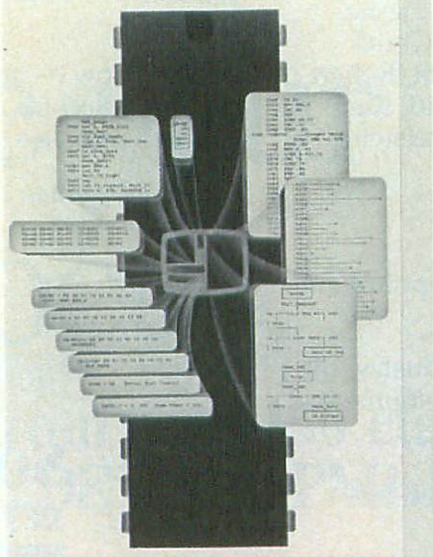
Programming an embedded system is by its very nature an audacious enterprise. It's not a task to be undertaken lightly, nor frivolously embarked upon by coders accustomed to the warm security of DOS or UNIX. Sans prototype, sans newfangled development "environment," sans (as often as not) operating system armor, programmers step into the ring armed only with their wits and a stack of impenetrable manuals.

This system may have been sufficient for hacking out the odd kilobyte of assembly language for low-level controller applications, but primitive development tools fall short in the creation of the ambitious systems in demand today. Embedded systems now require that the computer's power be applied to the creation and refinement of computer software. Programs that aid in the development of other programs have become the norm. In some cases, software can even substitute for expensive, inflexible dedicated hardware devices.

Microprocessor simulators are a case in point. Simulators are software-based models of the target system's hardware. The models reside in RAM within the development system—typically a DOS- or UNIX-based microcomputer—and precisely mimic the operations of the target processor as it executes the programmer's code. Simulators are most useful for debugging, permitting programmers to set breakpoints, perform single-step code execution, and monitor the values of variables as the program proceeds.

In fact, simulators are theoretically capable of replacing in-circuit emulators entirely, at least during the application-debugging phase of development. And since they cost much less than emulators, everyone can have one. No one will have to wait around until the ICE is free.

Simulators aren't without problems, though, the most important being execution speed—execution is slower by as much as an order of magnitude.



EMBEDDED SYSTEMS PROGRAMMING RATES SIM8051 v. 3.1

Available From: Cybernetic Micro Systems Inc., P.O. Box 3000, San Gregorio, Calif. 94074, (415) 726-3000
Price: $495
Support: 90-day warranty on diskette, free telephone support
System Requirements: IBM PC or clone, 256 kbytes RAM, DOS 2.0 or later

Subtler problems exist as well. Simulators vary in the depth to which they match the target system's hardware peculiarities. Some are based on processors' published specs instead of the (slightly different) chips that actually leave the production facility. And simulators vary in their ability to mimic or communicate with special-purpose support chips that may be incorporated into the design.

This review compares the strengths and weaknesses of a pair of PC-based simulators for the Intel 8051 controller. Although the vendors take different approaches to implementing the simulators, a number of fair comparisons and supportable conclusions may be drawn.

SIM8051

Sim8051 is a simulator-cum-debugger from Cybernetic Micro Systems. It runs on PCs under MS-DOS and simulates the Intel 8051 and 8052 families of microprocessors. It provides source-level debugging facilities when used with CMS's Cys-8051 assembler or Archimedes C.

Commands are communicated via special keys (such as Ctrl-T to toggle trace mode) or a command line. Macros are defined by pressing a function key or by special stimulus comments in the program source code. The keyboard remains live while the simulator is running, so you can examine and change data while the program executes.

In the upper-left quadrant of the display is a source listing. Because this is a source-level debugger, more than just a simple disassembly is performed. Rather, it shows the program as written, complete with comments and macros.

The PgUp and PgDn keys scroll the code window. Normally the next instruction to be executed is marked with an asterisk that stays in the center line of the window and doesn't move as the window scrolls. Pressing the Asterisk key resets the program counter to the asterisk's current position in the source. Pressing Ctrl-B sets a breakpoint. Sim8051 prompts for an address, but the position of the asterisk is the default.

As many as 25 breakpoints can be set. In addition to breakpoints on the program counter, Sim8051 offers traps to monitor registers. You can specify that a break should occur if some register is less than, equal to, or greater than some value. Three traps may be set per register.

Defining a trap couldn't be simpler. Simply press the = key, then type a trap expression such as `R5 > 17`. You'll recall that the 8051 has four distinct `R5` registers. Sim8051 only traps the register in the bank that's active at the time the command is issued. Different registers can be

Microprocessor simulators reside in the development system's RAM and precisely mimic the operations of the target processor as it executes the programmer's code.

trapped in different banks, but unfortunately you can't set traps for the same register in different banks.

Pressing Ctrl-V sets a memory trap that breaks when the specified byte is assigned the specified value. You can't specify inequalities, and only one memory trap may be active at a time.

The data window shows eight bytes, in hex on the upper line and in ASCII below. The window can be set to track different address spaces. For example, if set to `R1` it will show what `R1` points to as the register changes. Using the Up and Down cursor keys, you can cycle through four indirect modes (`R0`, `R1`, `PC`, and `PD`)and four direct modes (`Code`, `Data`, `Bit`, and `Special`).

Below the data window, the display shows the contents of all the registers in hex. The first few words on the stack are listed vertically down the middle. A bullet next to a register denotes what the data window is tracking. You can change the value of a register simply by typing an assignment statement such as `R7=0F`.

The entire right side of the display is consumed by a flow window that shows a flowchart of the program as it executes. This feature appears to be a novelty at first but is quite a useful tool. The appearance of the flowchart is enhanced if special comments are included in the source to describe and document the structure of the program.

The flow window can be replaced by a histogram of program execution by memory region. When the simulator runs at full speed (about 1 to 3% as fast as a real 8051), the histogram automatically replaces the flow window. The instruction pointer is sampled at regular intervals (the default is every 50 instructions), and the graph is updated accordingly.

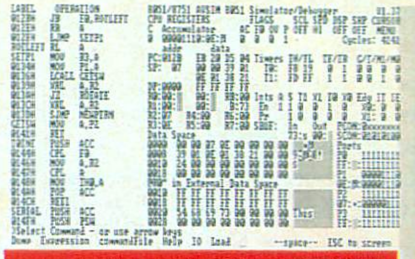
By default, the histogram divides the monitored range into 16 regular intervals. It would be far more useful if the zones in the histogram could be set to represent function boundaries. As it stands, the intervals must be regularly spaced and don't reflect the structure of the program at all.

Special comments in the source code mark which portion of the program is to be traced. Everything outside this range is run at full speed, so you can put debugged functions outside the trace range and trace only the functions you're studying. The problem is that you can only trace one contiguous range of code. To move functions in and out of the debug range, you must restructure your entire program.

Scripts are available to stimulate input ports. File `STIMULUS.HEX` is loaded if found and supplies a byte at one of the four ports at regular intervals. This mechanism is rather weak, and the reserved file name can be inconvenient if you ever want more than one file.

AVSIM51

Avocet makes simulators for a wide variety of small CPUs and microcontrollers, including AVSIM51 for the Intel 8051 and 8052. These remarkable simulators actually let you watch the workings of the processor on screen as it executes your code, providing probably the closest thing to a perfect simulation. You can run programs on this simulator that would crash an ICE.



```
LABEL    OPERATION           8051/8751 AVSIM 8051 Simulator/Debugger        V1.37
0128H    JB   F0,ROTLEFT     CPU REGISTERS          FLAGS    SCL SPD DSP SKP CURSOR
012BH    RR   A              C Accumulator     AC F0 OV P OFF HI  OFF OFF  MENU
012CH    LJMP SETP1          0 00001110:0E:    0  0  0 1             Cycles: 4241
ROTLEFT  RL   A                addr      data
SETP1    MOV  R3,A           PC:0128    E8 20 35 04 Timers TH/TL  TF/TR  C/T/M1/M0
0134H    MOV  P1,A           SP:  07    00 00 37 01   T0:
0136H    LCALL GETSW                    0E 01 38 21   T1:
0139H    XRL  A,R2           DP:0000    FF FF FF FF
013AH    JZ   ROTATE
013CH    XRL  A,R2
013DH    SJMP MEMPTRN
GETSW    MOV  A,P2
0141H    RET                 Data Space
TRINT    PUSH ACC
0144H    CPL  F0
0146H    MOV  A,R2
0147H    CPL  A
0148H    MOV  INB,A
014AH    POP  ACC
014CH    RETI
SERIAL   PUSH ACC
014FH    PUSH PSW
>Select Command - or use arrow keys
Dump Expression commandFile Help IO Load          --space--  ESC to screen
```

EMBEDDED SYSTEMS PROGRAMMING RATES AVSIM51 v. 1.37

Available From: Avocet Systems Inc., 120 Union St., Rockport, Maine 04856, (800) 448-8500, or (207) 236-9055 in Maine

Price: $379

Support: Free telephone support for registered users

System Requirements: IBM PC or compatible, 256 kbytes RAM

Everything about the CPU's workings is displayed along with simulator status, disassembled code, and two memory dump windows. It takes a while to get used to, but you can find out anything you want to know at a glance. It would be an improvement, though, if the different screen regions were separated from each other by visible borders or displayed in different colors.

Driving the simulator is easy. It operates in two modes: display and command. In display mode, you position the cursor over the value you want to change and change it. Use the arrow keys to position the cursor or use shortcuts such as Ctrl-A, which jumps to the accumulator.

A value is changed by editing the display or by using + and - to increment or decrement. Many values are displayed in more than one format; for example, the accumulator is shown in binary, hex, and ASCII. A change in any of the three changes contents of the accumulator.

Pressing Esc toggles between display mode and a command mode that's really a menu tree. Highlight the choice and press Enter or simply press the initial of the choice.

Pressing F1 runs the simulator at full speed, making the display look like a pinball machine gone wild. Portions of the display can be selectively disabled from updates during full-speed execution, thus making the simulator run even faster.

F10 single-steps one op code while F9 undoes one op code. The undo operation works on registers and memory changes, so F9 really runs the program in reverse. This isn't simply a trace backward for reviewing the recent history, but true backward execution. You can hit a breakpoint and run in reverse to see how the program got there. You can even back up, make a change, and go on.

A wide variety of breakpoints is available, and there's no limit to the number of breakpoints. A breakpoint can be set to trap when a register or address contains a specified value, falls within a range of values, or matches a bit mask. You can also trap the occurrence of some op code when a register or address is accessed (read or written) or just when it's written to. Breakpoints can be permanent (stay set until explicitly cleared) or one-shot (automatically cleared after use). All breakpoints except the op code type can have an associated passcount.

The op code breakpoint is a convenient way to jump to the end of a subroutine or interrupt. To set an op code breakpoint, you enter the mnemonic and sample arguments. The arguments are ignored but are supplied to specify the addressing mode of the instruction.

The input to the simulated chip can be driven by a script. The form of inputs is far more than adequate. The script file is merely a sequence of bytes. The bits in a byte may be tied to any port or address, and each bit may be assigned its own destination. The timing is at a preset I/O rate measured in cycles. Alternatively, input can be set to deliver the next byte whenever a specified register or port is accessed. This ability is useful for programs that use polling. The problem is that input can't always reflect the simulated environment. What's needed is a script that supplies the next value and the time it's to be delivered.

The simulator has a limited command-file ability. You can save all your keystrokes to a file. Playing back the file has the same effect as typing the contents of the file at the keyboard. Command files can be nested.

I give the manual mixed reviews. It's clear, concise, and well cross-referenced. The tutorial would be great if it matched the program, but it seems the software out-evolved the manual and rendered it inaccurate. The manual is specific to the MS-DOS version and refers UNIX and VMS users to Appendix B for instructions. (Appendix B contains nothing of the sort, by the way.) I don't know if other versions of the simulator come with different manuals, but I suspect parts of it are changed and other parts are common.

THE BOTTOM LINE

Sim8051 and AVSIM51 both promise to help programmers debug code for embedded applications. Both provide fairly full emulation of the 8051 and 8052. Sim8051's source-level compatibility with Archimedes C makes it the natural choice for users of that compiler.

Still, if I had to limit my purchase to one of these products, I'd choose AVSIM51. It was thoughtfully designed around just those features that are important during debugging. With AVSIM51, your days of waiting in line to use the ICE are over.

[redacted]

John M. Dlugosz is a programmer and analyst with Conductor Software in Irving, Texas.

by John M. Dlugosz

Debugging with ICE

An in-circuit emulator is a powerful device. It has an on-board computer that interprets a command language and drives the emulator based on your commands. The simplest configuration is to simply attach a dumb terminal to the ICE. This lets you communicate directly with the emulator and issue low-level commands from the console. Sometimes a PC is pressed into service as a terminal emulator. With the power of today's PCs, this is drastic overkill—sort of like using a nuclear reactor to power a toaster.

It's natural to imagine that you could program the PC to issue commands to the ICE on your behalf, parsing high-level macro statements into the low-level directives understood by the ICE system. A number of products on the market do precisely this, serving as front-end debuggers for ICEs. Such front ends vary; some are simple filters, while others are complete and robust development environments. This review presents my impressions of three ICE-based debuggers: a simple one, an average one, and an advanced one.

MICROTEK USD

Microtek's Universal Symbolic Debugger (USD) is a front end for Microtek ICEs. It runs on a PC, Sun, VAX, or MicroVAX.

Installing the program is simple: just copy the disk into a hard-disk subdirectory or onto a floppy, then edit a text file containing system setup information (including the display mode and COM port number). The protocol setting can also be changed. Use the speediest transfer rate the PC will handle or specify that the optional parallel interface is installed.

Simple as it is, the requirement to edit a text file is a sore spot. It's just one more thing that can go wrong during setup. Almost every program I've ever seen can figure out for itself what video mode to use. The port number could be specified with a simple /1 or /2 on the command line with a built-in default. The text-file approach is OK, but the program should use reasonable defaults or figure out if an item is missing.

EMBEDDED SYSTEMS PROGRAMMING RATES MICROTEK USD

Available From: MicroCASE Inc., 19545 N.W. Von Neumann Dr., Beaverton, Ore. 97006, (800) 547-4445
Price: $500
Support: One-year warranty, free telephone support
System Requirements: IBM PC or compatible, DEC VAX, or Sun workstation (PC version was tested); Microtek in-circuit emulator

As for the ICE, no setup is needed; the emulator detects and automatically configures itself to conform to the protocol being used.

USD acts as a filter between the programmer and the emulator. It's "ICE-command transparent," meaning anything you can type at the prompt on a dumb terminal can be typed at the USD prompt. But USD includes significant enhancements beyond these.

First, it simplifies uploading and downloading to the ICE. A simple command transfers a file between your disk and the target. Second, it adds symbolic support. When you enter an emulator command, a name preceded by a percent sign is recognized as a symbol. USD looks up the value and replaces it in the command line sent to the ICE. It also recognizes addresses coming back from the ICE and replaces them with symbolic names before displaying them to you.

The last major feature is the command file, which is nothing more than a text file with ICE commands in it. It's loaded into memory and executed like a macro or batch file. It features replaceable parameters, IF, GOTO, and a LOOP construct, allowing you to extend the ICE's command language by building your own high-level procedures. This capability greatly increases the power of the system and eases the debugging process.

The manual has different installation pages for each host. The rest of the documentation is common to all hosts—which is OK—but is also common to all targets, which is not. Why should an 8051 programmer have to read examples in 68000 assembly language? It makes it tough to figure out what's a USDism (percent sign before names) and what's part of the assembly language, unrelated to the example (pound sign before numbers).

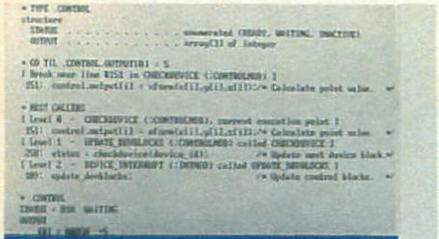
The manual is brief. Since the language is a superset of the regular ICE command language, only the enhancements are covered. There are three pages on symbols and seven on file processing, with most of the manual (40 pages) devoted to the command-file facility.

SOFTSCOPE

SoftScope is available in simulator and emulator versions for the 8086 family of CPUs. It's available from Concurrent Sciences or from Applied Microsystems (which markets the product as Validate XEI) with the ES-1800.

The program runs on IBM PCs and compatibles under DOS 3.0 or later. The target may be any member of the 80x86 family. The emulator version requires an Applied Microsystems ES-1800. Source-level debugging is supported in a number of high-level languages, including Intel assembler, PLM, Pascal, and FORTRAN; Intel, Lattice, and Metaware C; and PSS Jovial.

To install the program, just place the



```
* TYPE .CONTROL
structure
  STATE . . . . . . . . . . . . . enumerated (READY, WAITING, INACTIVE)
  OUTPUT . . . . . . . . . . . . array[3] of integer

* GO TIL .CONTROL.OUTPUT[0] = 5
[ Break near line #151 in CHECKDEVICE (:CONTROLMOD) ]
 151: control.output[i] = xform(x[i],y[i],z[i]);/* Calculate point value.  */

* LIST CALLERS
[ Level 0 - CHECKDEVICE (:CONTROLMOD), current execution point ]
 151: control.output[i] = xform(x[i],y[i],z[i]);/* Calculate point value  */
[ Level 1 - UPDATE_DEVBLOCKS (:CONTROLMOD) called CHECKDEVICE ]
 250: status = checkdevice(device_id);          /* Update next device block.*/
[ Level 2 - DEVICE_INTERRUPT (:INTMOD) called UPDATE_DEVBLOCKS ]
 189: update_devblocks;                         /* Update control blocks.  */

* CONTROL
STATE = RUN WAITING
OUTPUT
   [0] : [illegible] -5
```

EMBEDDED SYSTEMS PROGRAMMING RATES SOFTSCOPE

Available From: Concurrent Sciences Inc., 105 S. Washington St., Moscow, Idaho 83843, (208) 882-0445
Price: $1,000-$2,000
Support: One-year warranty
System Requirements: IBM PC or compatible, 256 kbytes memory, DOS 3.0 or later, Applied Microsystems ES-1800 ICE

supplied disk in drive A and run the installation program. The problem in my case was that there was no disk! I decided to look over the simulator version and found its disk contained the emulator software. I never did find the simulator program.

After copying the debugger onto the host, you must set up the ES-1800. If you use COM2, you must also edit the configuration file. You then attach the cable between your computer and the terminal port on the ICE. I borrowed the 25-pin/9-pin adapter that came with my mouse to accomplish this. You then pop the front panel off the ICE and twiddle the thumb switch, turn everything on and run the program, go into setup mode (which turns the program into a dumb terminal emulator), and issue commands to the ES-1800 to save the proper settings in the ICE's EEPROM. After waiting for EEPROM-burning, you quit the program, shut off the emulator, twiddle the thumb switch back to "use EEPROM settings," button it up, move the serial cable to the other port, turn the emulator back on, and manually start SoftScope.

The process isn't as bad as it sounds. If you know the ES-1800, the procedure is apparent without reading SoftScope's instructions.

This product appears at first glance to be a simple monitor-style debugger. All the output is in the tradition of a glass teletype. Most commands can be abbreviated to a single letter: L to list source code, S to single-step, etc.

SoftScope reads your source file and symbol table when it downloads the program. You can use symbol names as addresses in commands, and disassembly can be matched to the source code.

The real-time trace buffer can be accessed and displayed in a variety of useful formats. Showing source lines or individual source statements with disassembled instructions in a trace-backward form is one of the most valuable debugging methods. However, SoftScope insists on displaying the trace upside down with the most recent lines on top.

One nice display lets the programmer examine the stack; it shows all the pending function calls and, optionally, the source lines that contain the calls.

Any number of code breakpoints can be set. Hardware-cycle breakpoints are part of the GO command. You can GO until a specific hardware condition is met using the power of the ES-1800's logic analyzer.

SoftScope's best feature is its ability to handle typed data. It can handle simple types, such as integers and strings, as well as structures and arrays. Just type the variable name preceded by a period, and its value is displayed in a meaningful form consistent with the declared type. Changing memory is just as easy. A simple assignment statement changes the value of any variable, array element, or structure member.

The on-line help is good. The HELP (keyword) command brings forth volumes on any subject. The entire command reference manual, with both appendices, is

It's common to press a PC into service as a terminal and plug it into the in-circuit emulator, but with the power of today's PCs this is drastic overkill—sort of like using a nuclear reactor to power a toaster. Debugging software takes the next logical step, making the PC issue instructions to the ICE on your behalf.

THE FUTURE OF THE BUS/BOARD INDUSTRY IS HERE NOW...AND IT'S AT BUSCON/89-WEST

There's something exciting going on in the bus/board industry these days...the future.

New chips. New products. New technology. And ever-increasing choices. There's a world of difference between the bus/board industry of today and the one of tomorrow...a world that's waiting for you at BUSCON/89-West.

Whether you're interested in NuBus, VME, Multibus or other bus/board architectures, you'll find more products, services and solutions at BUSCON than ever before. Because the BUSCON Show and Conference has grown right along with bus/board technology to become the industry event. In fact, BUSCON has doubled in size in the last two years alone.

GET A FIRST-HAND VIEW OF THE FUTURE... PLAN NOW TO ATTEND BUSCON/89-WEST!

Sponsoring Publications: Computer Design, Computer Technology Review, Control Engineering and the Microcomputer Interface Group, EDN, ECN, EETimes, Electronic Buyers' News, Electronic Design, Electronic Products, ESD, IAN, I&CS, InfoBus Report, SuperMicro, UNIX World, VMEBus Systems Magazine, VME News, Embedded Systems Programming.

Association Sponsors: VITA, MMG, STDMG, STEMUG, PDOS.

□ **YES!** I am interested in attending BUSCON/89-West, the bus/board industry event.

□ I am interested in exhibiting. Please send me more information.

BUSCON/89-WEST
FEBRUARY 7-9, 1989
SANTA CLARA CONVENTION CENTER
SANTA CLARA, CA

Name ______________________________

Title/Position ______________________________

Company ______________________________

Address ______________________________

City ______________________ State ________ Zip __________

Phone (_____) ______________________________ BW2

RETURN TO: CMC, 200 Connecticut Avenue, Norwalk, CT 06856-4990 (203) 852-0500.

CIRCLE #235 ON READER SERVICE CARD

EMBEDDED SYSTEMS PROGRAMMING RATES XRAY/68K

Available From: Microtec Research Inc., 2350 Mission College Blvd., Santa Clara, Calif. 95054, (408) 980-1300
Price: $3,500-$14,000
Support: Depending on host, one-year warranty provides maintenance and update of product and response to documented software performance reports.
System Requirements: 640 kbytes memory; DOS, UNIX, or VMS; Applied Microsystems ES-1800 emulator

available on-line.

In short, SoftScope is a very powerful and effective debugging tool. Its symbolic support and typed data display are great. The user interface can be frustrating, though. I wish you could ask for the trace or disassembly display and scroll through it using the cursor keys. As it is, you get a single screenload and must press Q to quit or any digit (one through nine) to request more lines. There's no way to back up.

The system ought to be made crash-proof as well. When I uploaded a garbage file with a valid source file, it locked up.

XRAY/68K

XRAY/68K is available for use with or without ES-1800 for the 68000 family. It runs under MS-DOS, UNIX, and VMS.

Setting up XRAY and the ES-1800 requires the same ritual as for SoftScope, but the settings are all different. Fortunately, the ES-1800 can hold two sets of user settings in its EEPROM. The manual is good, with clear setup instructions and a tutorial.

This program will dazzle you from screen one. It is a source-level, window-oriented debugger with separate windows for source and assembled code, registers, stack, data, watchpoints, etc. You can configure the display to your needs and even define your own windows.

The high-level screen shows the source code with a highlight over the current line, a data "watch" window, a trace window, and the command window. Pressing F9 executes one line. The single-step command is slow in this mode; there's a noticeable pause before execution advances to the next line.

F3 switches to the assembly screen. The source window now shows mixed source and assembly, the trace window is hidden, and stack and register windows become visible. If these screens don't suit your needs, you can rearrange and reconfigure to your heart's content.

Programmers control XRAY/68K by pressing function keys or by typing commands in the command window. The command capability is fantastic. You can define macros that are essentially user-defined procedures, in effect extending the debugging language. It will interpret any legal C expression. Control flow is like C's, with `if...else`, `while`, `do`, and `for` statements.

Commands can be run in the command window, of course. A command can also be attached to a breakpoint and automatically run when the breakpoint is tripped or attached to a window and run every time that window is displayed.

The command capability is a real treasure. Better than a simple breakpoint, it almost encourages you to write a little procedure to check for anomalous conditions and attach it to a particular point in the code. When the breakpoint is tripped, the macro is executed. The macro may pass control back to your application or generate a break condition.

XRAY/68K is a true source-level debugger. You can step a source line at a time, dump all local variables (formatted for their declared type) for a function, and examine the calling sequence of pending functions. You can debug on a high level in the context of the language you used when writing your code.

One especially useful command is `printf`, which lets you access program data and print it in any format. The `expand` command shows all local variables for any or all pending functions.

For C programming, XRAY/68K offers a library of simulated I/O routines. These let you use standard input and output in C programs run on the emulator, even if you haven't written I/O primitives. So if you were writing a program for a toaster, which doesn't have any output (the toast doesn't count), you could still use `debug printf ()` statements in your development work.

XRAY/68K's library of simulated I/O routines lets you use standard input and output in C programs even if you haven't written I/O primitives. So if you're writing a program for a toaster that doesn't have any output (the toast doesn't count) you could still use printf statements for debugging purposes.

[redacted]

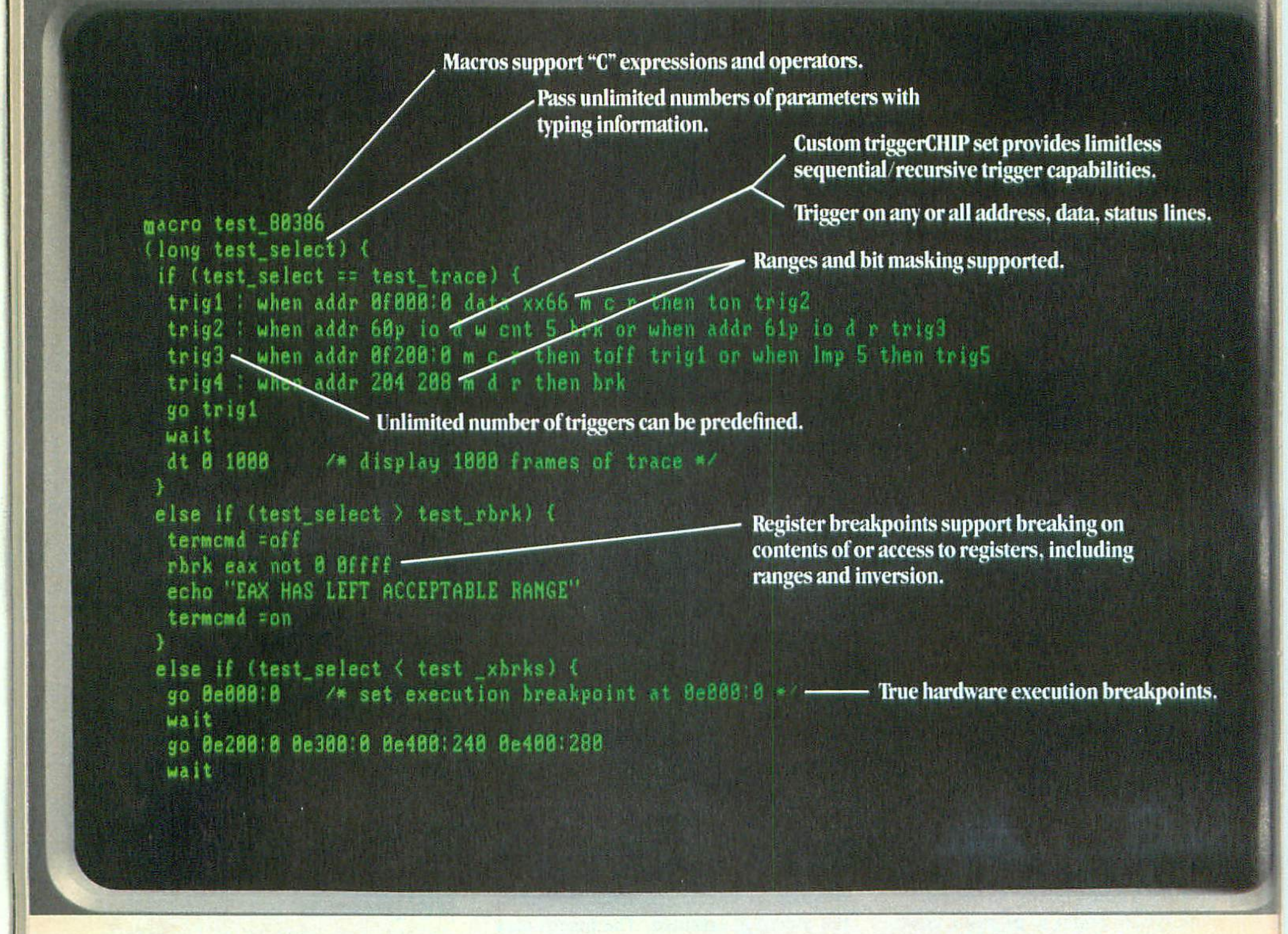
XRAY/68K also provides an on-line help facility. An index window pops up and you pick a topic by moving a highlighted bar with the cursor keys. The help is good, but the index is off by one. You have to pick the topic after the one you want to see.

XRAY/68K, the ES-1800, and Applied Microsystems' C compiler form a powerful, well-integrated system. This is my favorite of the debugging tools reviewed this month.

[redacted]

John M. Dlugosz is a programmer and analyst with Conductor Software in Irving, Texas.

hyperICE™ Feature Set Update Series No. 1 CUSTOM TRIGGERCHIP™ SET



```
macro test_80386
(long test_select) {
 if (test_select == test_trace) {
  trig1 : when addr 0f000:0 data xx66 m c r then ton trig2
  trig2 : when addr 60p io d w cnt 5 brk or when addr 61p io d r trig3
  trig3 : when addr 0f200:0 m c r then toff trig1 or when lmp 5 then trig5
  trig4 : when addr 204 208 m d r then brk
  go trig1
  wait
  dt 0 1000     /* display 1000 frames of trace */
 }
 else if (test_select > test_rbrk) {
  termcmd =off
  rbrk eax not 0 0ffff
  echo "EAX HAS LEFT ACCEPTABLE RANGE"
  termcmd =on
 }
 else if (test_select < test _xbrks) {
  go 0e000:0    /* set execution breakpoint at 0e000:0 */
  wait
  go 0e200:0 0e300:0 0e400:240 0e400:280
  wait
```

Macros support "C" expressions and operators.

Pass unlimited numbers of parameters with typing information.

Custom triggerCHIP set provides limitless sequential/recursive trigger capabilities.

Trigger on any or all address, data, status lines.

Ranges and bit masking supported.

Unlimited number of triggers can be predefined.

Register breakpoints support breaking on contents of or access to registers, including ranges and inversion.

True hardware execution breakpoints.

FIND BUGS FASTER WITH MICROCOSM'S™ TRIGGERCHIP SET

Microcosm's hyperICE product line gives you the information you need for fast, accurate debugging of 16 and 32 bit microprocessor-based systems. The custom triggerCHIP set allows four levels of sequential and recursive triggers for designing your system and rooting out bugs buried under layers of code.

Trigger on any target condition—address, data and status lines, even up to 8 lines of an optional logic probe. What's more, new trace or break definitions can be loaded during realtime emulation. The triggerCHIP set monitors your system continuously. With Microcosm's hyperICE system, you've got the draw on all your bugs, no matter how complex your system design.

With the hyperICE system, you don't need to walk through code, line by line, visually monitoring register activity. Register Breakpoints automatically break on contents or access of a register for faster debugging. HyperICE is the only in-circuit emulator to give you full-featured trigger capabilities, register breakpoints, fully-qualified trace *and* high-level language debugging, coupled with realtime emulation. Using hyperICE, your bugs are an easy target.

If you're ready for smooth system integration, you're ready for Microcosm's advanced triggering. Call now at (503) 626-6100. We'll show you the slickest triggering features around.

Microcosm's ICE supports the 80386, 80286, 80186/188, 8086/88, 68000/10/08, MCS-51, 68HC11, and 64180 microprocessors.

MICROCOSM™
15275-E S.W. Koll Pkwy., Beaverton, OR 97006

hyperICE, triggerCHIP and Microcosm are trademarks of Microcosm, Inc.

CIRCLE #236 ON READER SERVICE CARD

by Ernest L. Meyer

Walk, Trot, or Gallop?

Built-in self-test algorithms for catching errors in solid-state memory automate error diagnosis—on the manufacturing floor or in the field.

As memory systems grow beyond 16 kbytes, simple memory tests become increasingly inadequate. When you think that a 1-Mbit chip is the same size as a 16-kbit chip but contains 64 times as many memory cells, it's hardly surprising that the chip must be tested much more thoroughly. In fact, even a 64-kbyte memory is prone to highly complex errors that simple memory tests will miss.

Memory error diagnosis is therefore a growing concern to embedded systems developers as the average memory size in embedded systems increases. Because memory tests are highly repetitive, however, a more complex test doesn't need a great deal more code space. A few added bytes of code can vastly improve the test.

On the other hand, a more thorough test does take longer to run. Nested loops keep the code size down, but the test time can be as long as minutes, or even hours. A shorter test may be preferable in some situations. As we shall see, there's a surprising variety of possible test routines.

Manufacturing tests should be as thorough as possible. Running even minute-long tests on the production floor with multimillion-dollar automatic test equipment would prove exorbitantly expensive. Therefore, it's even more sensible to embed manufacturing tests in the code. Then the units need only be plugged in and turned on to run a thorough memory test.

This technique, called BIST, also simplifies error diagnosis in the field after the units are shipped. The user can identify problems by running the test remotely. All the manufacturer may need to do is send a new chip to plug in, thus cutting field repair costs as well.

RELIABILITY ISSUES

The usefulness of a memory test is partly defined by a bathtub-shaped reliability life curve. The curve drops sharply in the early stages of system life due to "infant mortality." About half of bad components fail in the first day or so of use, so manufacturing tests should be thorough. The level middle region is our real concern. It indicates a relatively low, random failure at an average frequency indicated by the manufacturer's component failure rate. The memory component failure rate may be between 0.03 and 0.3% per 1,000 hours in a typical system.

The failure rate for an individual component is multiplied by the number of components on the board to get the system failure rate. Failure rates quickly add up. For instance, if the board contains thirty-two 64-kbit chips with an error rate of 0.18% per kilohour, one quarter of the shipped products will fail within the first six months of use.

It's advisable to perform run-time memory tests at regular intervals. Typically, these tests are executed every time the system is turned on. Using a hard-booted test helps ensure memory diagnostics can be run after a system crash.

TYPES OF FAILURE

There are two basic types of memory failure: soft errors and hard faults. Soft errors are usually caused by stray radiation at run time and aren't permanent. Soft-error frequency occurs 10 times as often in NMOS as in CMOS RAMs.

Hard faults are more serious since they are caused by wires fusing out and necessitate replacement of at least one chip. They can be opens (burned-out wires) or shorts (cross-connected wires).

Hard faults have varying effects, depending on where they occur. Both shorts and opens can have static or dynamic effects. Static faults don't change over time and can be found relatively quickly. Dynamic faults are more difficult to spot, since they are often manifest only after particular sequences of events.

Figure 1 shows a typical dynamic memory system structure. Faults can occur in the decoder, refresh, driver, or sense amp circuitry or in the matrix of memory cells themselves. Finding the faults requires writing data to specific locations in the memory matrix; it's often necessary to know the length (in number of rows) and width (in number of columns) of the memory cell matrix. This information can be obtained from the chip manufacturer.

Most memory chips store one bit of each memory word. All the following tests assume a bit-wide memory chip structure; nibble- and byte-wide memory chips may require different test step intervals to achieve the desired pattern of ones and zeros.

The tests are presented here in a generic high-level notation. It's worth mentioning that the variables used in the actual test code will be destroyed by the test if they are stored in the area of memory that's being diagnosed!

Ideally, the variables should be stored in a CPU register. If no registers are available, the variables should be written to an absolute location and moved halfway through the test. Alternatively, variables can be replaced by absolute values wherever possible.

Also, most of the tests run twice, the second time with complemented data. Rather than looping through a procedure and passing ones instead of zeros through the procedure call (and vice versa), the loops can be expanded.

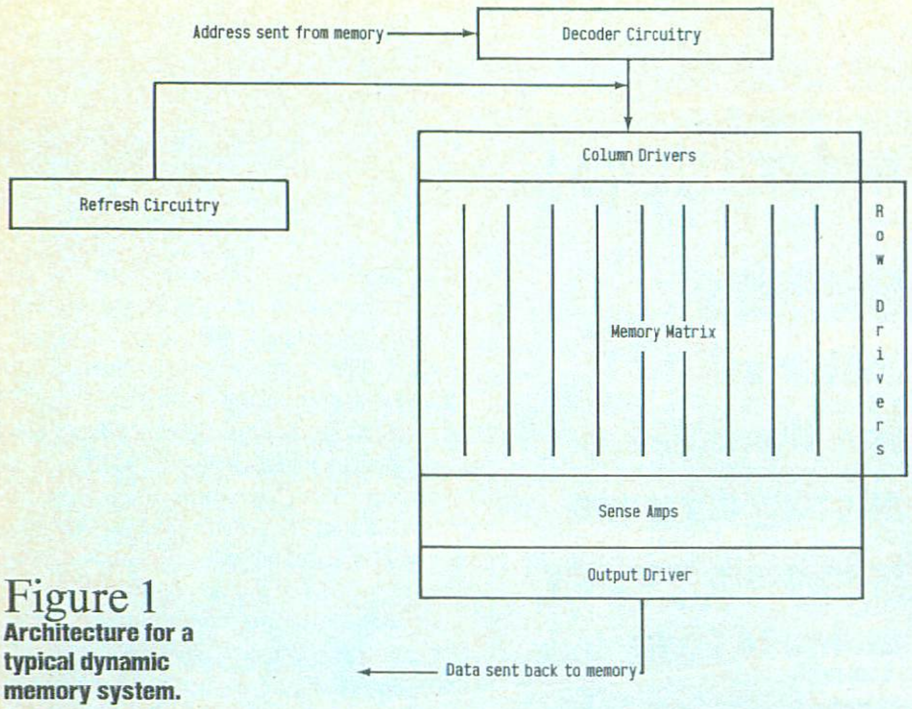


Figure 1
Architecture for a typical dynamic memory system.

ERRORS IN THE MEMORY MATRIX

The most obvious type of fault occurs when a cell in the memory matrix is "stuck" by a short or open. Most soft errors manifest themselves as stuck-at-zero and stuck-at-one faults.

The Sequential Write/Read test (see Listing 1) is the most obvious matrix test. Zeros are written sequentially to each address in memory. After each write, the addressed cell is read to see if the bit was successfully written. The test is then repeated with ones instead of zeros.

This test is of order N complexity and constitutes the simplest possible memory cell test. The test checks for the existence of any cell opens or shorts that have tied the storage cell to a high or low value. Unfortunately, single-cell faults are the least common type of fault and the test finds very few other types of errors.

The Sequential Write/Read test can, however, identify a soft error if the system hasn't been powered down since the error occurred. If the system fails the sequential test before power-down, but then passes the test after the power is switched off and back on, a soft error has occurred. Since about a third of all NMOS errors are soft, they're worth checking for.

Not all matrix faults are of the "stuck-at" variety. If a short occurs between two storage cells, writing a value to one memory location will change the value in its neighbor. The GALPAT test (explained later) will find these faults.

DRIVERS AND DECODERS

The row and column drivers below the decoder circuitry turn on an individual memory cell and force it to a high or low value during memory writes. Faults in the row or column drivers affect the xth bit of every y memory word.

Driver faults are the most common because the row and column drivers use more power and, as a result, get hotter and burn up faster. If test time or test code length is a concern, a simple driver test (Listing 2) can be performed in which only one bit is tested in each row and column.

The decoder circuitry in a RAM system interprets the row and column in which an addressed memory cell is located. Some decoder is inside the memory chip; some is in separate chips to pick particular memory segments and subsets. Hard faults in the decoder circuitry can result in the same data being written to more than one location. If a decoder fault occurs inside the chip, it only affects one bit. Faults in the decoder chips around the memory chips, on the other hand, can have a similar effect on bytes, words, or entire blocks of memory.

The Walk test (Listing 3) provides a complete decoder check. Because it has an N^2 complexity, though, other tests are preferable for systems larger than 64 kbytes. On a zero background, the system writes a one to the first cell and then reads all cells sequentially. After writing a one back to the first cell, Walk repeats the process for each cell in memory. The pattern is then repeated for complemented data.

Walk also tests for memory matrix opens and shorts. When Walk is used, sequential write/read isn't necessary. By inserting a delay into the write/read/write cycle, Walk can also be used to test for sleeping sickness, the gradual loss of data after it has been stored for a while.

SLEEPING SICKNESS AND VOLATILITY

The remaining tests deal with dynamic faults. Sleeping sickness and volatility are the easiest dynamic faults to find because they aren't pattern sensitive: the fault is manifest regardless of the data written to RAM.

In dynamic RAM systems, refresh circuitry keeps data active in the dynamic memory cells. Hard faults in the refresh circuitry usually cause sleeping sickness.

Static RAMs aren't susceptible to sleeping sickness because they have no refresh circuitry. However, static RAMs do have data retention problems. Shorts between adjacent memory cells in static RAMs can cause the inversion of stored data in one direction or the other. This problem is known as volatility.

The Checkerboard test (Listing 4) is the most common check for retention problems. The program writes alternating zeros and ones (in a checkerboard pattern) into memory. The pattern is then checked and the whole process repeated for complemented data.

The Column Bar test is similar to

Listing 1
The Sequential Write/Read test.

```
PROC Sequential():
{
  FOR i = 0 TO TopOfMemory
    WRITE: Ci = 0
    READ:  Ci (=0)
    WRITE: Ci = ∧0
    READ:  Ci (=∧0)
  NEXT i
}
```

Checkerboard except that strips of ones and zeros are used instead of a staggered pattern. Listing 4 actually produces a column bar pattern for most chip architectures. (Most chips have an even number of rows and columns; if there's an odd number of rows, a checkerboard pattern is produced.)

Listing 2
A faster Row/Column Driver test.

```
PROC TestDriver():
{
  TestDriverLoop(RowWidth)
  TestDriverLoop(ColumnWidth)
}

SUB TestDriverLoop(j):
{
  FOR i = 0 TO TopOfMemory STEP j
    WRITE: Ci = 0
    READ:  Ci (=0)
    WRITE: Ci = ∧0
    READ:  Ci (=∧0)
  NEXT i
}
```

SENSE AMP FAULTS

Retention problems are relatively easy to find; sense amp faults are another kettle of fish altogether. The sense amp circuitry detects the value stored at the addressed cell during dynamic RAM reads. Sense amp errors become increasingly common as the memory matrix size increases. This is especially true in "noisy" systems, such as battery-run units, where there's a great deal of power and temperature fluctuation.

Larger memory chips are more likely to manifest sense amp faults. Megabit chips are particularly likely to have faults because they only use one transistor in each memory cell and store the memory value as a charge in a capacitor. Since the storage mechanism is charge-based, the memory cells are sensitive to changes in the electric field. The charges in neighboring capacitors themselves change the electric field. Sense amp tests should *always* be run with megabit chips.

Listing 3
The Walk test.

```
PROC Walk():
{
  FOR i = 0 TO TopOfMemory
    WRITE: Ci = 0
  NEXT i
  WalkLoop(0,∧0)
  WalkLoop(∧0,0)
}

SUB WalkLoop(j,k):
{
  FOR i = 0 TO TopOfMemory
    WRITE: Ci = j
    FOR x = 0 TO TopOfMemory
      IF x = i THEN NEXT x
      WRITE: Cx = k
    NEXT x
    READ: Ci (=j)
  NEXT i
}
```

Listing 4
The Checkerboard test.

```
PROC Checkerboard(j,k):
{
  i = 0
  DO:
    WRITE: Ci = j
    i = i + 1
    WRITE: Ci = k
    i = i + 1
  LOOP UNTIL i = TopOfMemory
  i = 0
  /* optional pause here */
  DO
    READ: Ci (=j)
    i = i + 1
    READ: Ci (=k)
    i = i + 1
  LOOP UNTIL i = TopOfMemory
}
```

Listing 5
The Shifted Diagonal test.

```
PROC DiagonalShift():
{
  DiagonalLoop(1,0)
  DiagonalLoop(1,1)
  DiagonalLoop(∧1,0)
  DiagonalLoop(∧1,1)
NEXT i
}
SUB DiagonalLoop(j,Flag):
{
  FOR i = 0 TO TopOfMemory
    IF Flag = 0 THEN
      WRITE: Ci = j
      ROTATE j
    ELSEIF Flag = 1 THEN
      READ: Ci (=j)
      ROTATE j
      WRITE: Ci = j
    ENDIF
  NEXT i
}
```

Listing 6
A neighborhood GALPAT.

```
PROC CheckSensitivity():
{
  FOR i = 0 TO TopOfMemory
    WRITE: Ci = 0
  NEXT i
  SensitiveLoop(0,∧0)
  SensitiveLoop(∧0,0)
}
SUB SensitiveLoop(j,k):
{
  FOR i = 0 TO TopOfMemory-1
    WRITE: Ci = j
    WRITE: Ci + RowWidth = k
    WRITE: Ci + ColumnWidth = k
    WRITE: Ci - RowWidth = k
    WRITE: Ci - ColumnWidth = k
    /* pause for a while */
    READ: Ci (=j)
  NEXT i
}
```

Unfortunately, faults in the sense amp are a devil to find. Moreover, each type of sense amp error—slow write recovery, pattern sensitivity, or disturb—requires a different test, as we shall see.

The simplest sense amp error to find is slow write recovery. In many sense amplifier designs, switching to a new output value after reading a long sequence of the same bit value can result in a small delay. The Shifted Diagonal test (Listing 5) is designed to find this error.

On a background of zeros, a diagonal of ones is written across the memory space. Each column is then read individually. The diagonal is shifted one step at a time so that it sweeps through memory, and the column read operation is repeated. The entire process is then repeated using complemented data.

The effect of a shifted diagonal is to intersperse a long sequence of identical writes with one write of a complemented data bit. Note that in a bit-wide memory chip the code in Listing 5 will move the complemented bit across the chip. The error recovery routine should therefore reference the count variable, c, and divide it by the word width to find which chip is at fault.

Shifted Diagonal is a relatively long test to run, with a $4N^{3/2}$ complexity. It's nevertheless shorter than tests for pattern sensitivity or disturbance.

Galloping patterns, or GALPATs, look for memory cell opens and shorts, address uniqueness and decoder errors, sense-amp interaction, capacitive coupling between neighboring cells, noise effects, and access-time problems.

GALPATS provide an exceptionally high degree of fault coverage and are particularly good at finding pattern sensitivities, which occur when the value in one memory cell is changed and different values are stored in the cells around it. Pattern sensitivity may be introduced when every bit except one in a column or row is the same value. More often, individual cells are sensitive to the values stored within a two-cell radius.

Listing 6 shows a neighborhood GALPAT. On a background of zeros, the test cell is complemented and then read alternately with a set of cells in the chip. While a full GALPAT reads the test cell alternately with every other memory cell in the chip, a neighborhood GALPAT reads only a set of cells in close proximity to the test cell. This sequence is then repeated, with each memory cell acting as the test cell once, and the entire process is repeated using complemented data.

For a full GALPAT, execution time is proportional to the cube of the cell count; for a neighborhood GALPAT, it's proportional to the square of the cell count.

Ping Pong is a variant of GALPAT. The address sequencing technique in Ping Pong is the same; however, instead of using a flat background of ones or zeros, Ping Pong uses alternating blocks of ones and zeros in a megacheckerboard pattern. (Additional code is required to write the background data matrix using Ping Pong.) While Ping Pong has execution time and fault-finding capabilities similar to GALPAT's, it can also locate other data-sensitivity problems.

Disturbance is caused by several consecutive writes of a complementary value to the same cell. The continual shifting from one to zero and back confuses the memory cell and causes it to contain a less-than-ideal high or low value at the end. If the sense amps aren't sensitive enough, they can't determine the final value in the disturbed cell. In fact, that's the main problem with designing a memory chip: if the sense amps are more sensitive, they'll have pattern sensitivity problems; if they're less sensitive, they'll have disturbance problems. Individual chip manufacturers will report the type of error that's more common for each chip they produce.

Of the three disturb tests—Column Disturb, Row Disturb, and Surround Disturb—the latter is best, but takes much longer to run. Listing 7 is the test for disturbance in the main routine; Listing 8 is the test for row or column disturb.

Column Disturb works on a column of zeros. The first and last bits of data in the column are then disturbed by comple-

Listing 7

Test for disturbance problems in main routine.

```
PROC Disturb():
{
 FOR i = 0 TO TopOfMemory
  WRITE: Ci = 0
 NEXT i
 j = 0
DisturbLabel:
 /* STEP ColumnWidth or RowWidth */
 FOR i = 0 TO TopOfMemory
  SurroundDisturb(j)  /* or QuickDisturb(J) */
 NEXT i
 IF j = 0 THEN j=∧j: GO ColumnDisturbLabel
}
```

Listing 8

Test for row or column disturb.

```
QuickDisturb(J):
{
 FOR x = 1 TO 254 /* must be an even number */
  j = ∧j
  WRITE: Ci = j
  WRITE: Ci + 1 = ∧j
  WRITE: Ci + ColumnWidth -
1 = j /* OR RowWidth */
  WRITE: Ci + ColumnWidth -
2 = ∧j /* OR RowWidth */
 NEXT x
 READ: Ci (=j)
 READ: Ci + ColumnWidth -
1 (=j) /* OR RowWidth */
}
```

Listing 9

Test for surround disturb.

```
SUB SurroundDisturb(j,k):
{
 WRITE: Ci = j
 FOR x = 1 TO 254
  j = ∧j
  WRITE: Ci + 1 = j
  WRITE: Ci - 1 = j
  WRITE: Ci + RowWidth - 1 = j
  WRITE: Ci + RowWidth = j
  WRITE: Ci + RowWidth + 1 = j
  WRITE: Ci - RowWidth - 1 = j
  WRITE: Ci - RowWidth = j
  WRITE: Ci - RowWidth + 1 = j
 NEXT x
 READ: Ci (=j)
}
```

menting them continuously up to 255 times. The last write to the first and last bits in the column should leave a zero in those locations. This process is repeated for the second and penultimate memory cells in the location, after which the data in the first and last cells is read. The sequence is repeated for every column in memory, then for complemented data.

Row Disturb is the same as Column Disturb except the process is performed on rows instead of columns. The pattern's execution time basically depends on the number of disturbs executed.

With Surround Disturb (Listing 9), every single cell is disturbed. On a background of zeros, the system complements the first cell and repetitively reads the eight physically adjacent cells up to 255 times. The first cell is then read and restored to zero. This procedure is repeated for each memory cell.

Memory cells on the edges of the arrays, obviously, have only five neighbors; the cells in the corners have only three. However, to achieve higher code compaction, the quirks at the edges and corners can be ignored.

Surround Disturb finds single-cell faults and shorts between adjacent memory cells as well as disturbance problems. Execution time varies with the number of disturb cycles. Complexity is Nds, where d is the number of disturb cycles and s is the size of the disturbance ring.

HARDWARE ERROR CHECKING

Hardware can be used for error detection and correction and can reduce the risk associated with memory failure. The two most common hardware techniques are parity and ECC. Parity detects many memory errors but doesn't identify which memory chip is at fault and why. Software diagnostic and recovery routines are still necessary. ECC, usually achieved by modified Hamming codes, can correct single-bit errors but introduces a 25%-35% memory hardware overhead. ECC is therefore impractical for most commercial memory systems. It introduces additional diagnostic problems because the ECC code storage space isn't directly readable.

Ideally, the ECC subsystem should be switched off during main memory test. It's worth consulting with the hardware designer to find out if the ECC subsystem can be made switchable from software.

These steps are only part of the solution, however. Testing the memory space used to store ECC codes can be a real problem. The simple solution is to make the bit fields used for DATA and ECC CODE jumper-selectable. After ECC is turned off and main memory is tested, the storage space used for ECC codes is then swapped into the main memory space for a second suite of tests. Naturally, this makes field testing much more difficult. The alternatives are more complex than can justifiably be covered here.

Some chips have built-in self-tests to facilitate manufacturing tests. The most common memory BIST technique places four quadrants of the memory matrix in parallel. The data stored in these regions is then logically ORed and output as a single value, cutting test time by 75%.

Placing the chip in a BIST state usually involves raising one of the chip's pins to a high voltage or negative voltage level. If the required voltages are available in the system, it may be possible to place chip BIST under software control.

In very large memory systems, this course may yield significant overall cost advantages. On the other hand, it may require additional chips and board real estate, which would be greeted with reluctance by the hardware designer.

Nevertheless, during production test full utilization of all available resources to minimize test time is obviously a desirable course. It makes sense to work in concert with the hardware designer in these areas to ensure that the final product is both functional and testable.

Ernest Meyer is an independent technical writer and former editor of VLSI Systems Design. *He holds a B.Sc. in philosophy and psychology from Oxford University.*

by Ernest L. Meyer

ISDN Made Simple: National Semiconductor's HPC16400

After years of fretful indecision, Integrated Services Digital Network (ISDN) is finally settling down into a reasonably coherent standard. National Semiconductor Corp., Santa Clara, Calif., has thrown its not inconsiderable weight behind CCITT standards Q.921 and Q.931 (as approved by the T1/D1 for North America) in its high-performance data communications controller, the HPC16400.

National's contribution to the ISDN bandwagon is based on the 16-bit HPC microcontroller core. HPC is used inside a number of sister products, so a panoply of C compilers, assemblers, linkers, debuggers, and libraries on DOS, VMS, and VAX UNIX platforms is immediately available for the HPC16400. The company is also introducing an ISDN basic rate interface (BRI) software package for the HPC16400 that may simplify ISDN code development substantially.

The HPC core is a relatively simple execution unit. It contains four general-purpose 16-bit registers (a, b, x, and k), a 16-bit stack pointer, a 16-bit program counter, three timers, a seven-level interrupt, and a synchronous data communications channel.

About 60 instructions are available in the HPC core, with 10 addressing modes. Most instructions are only one byte long for efficient code compaction. Jumps and loops result in particularly efficient code compaction.

The Microwire/Plus unit inside the chip core can transmit or receive bytes of data through a synchronous serial data link. The Microwire serial link is suitable for attaching keyboards and LCD displays to the ISDN system.

Since the HPC16400 is based on the HPC microcontroller core, a panoply of C compilers, assemblers, linkers, debuggers, and libraries is immediately available.

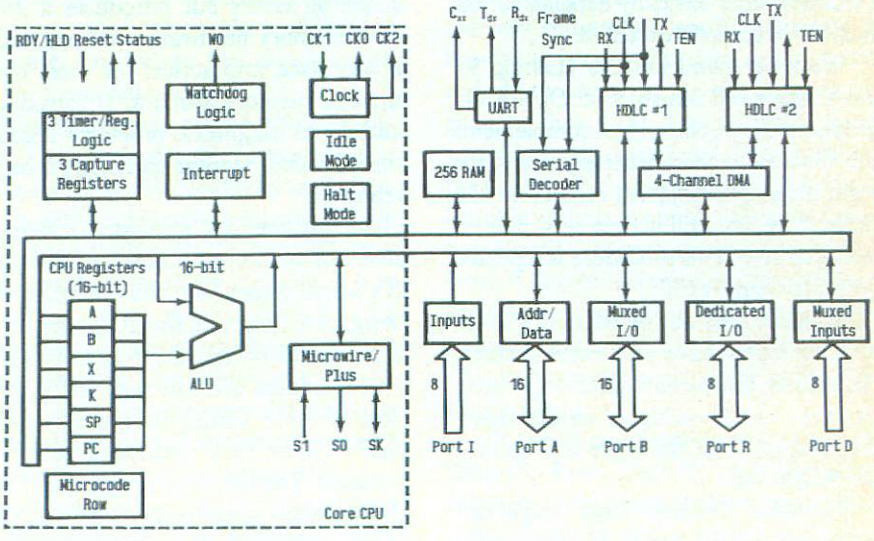
THE CHIP

The HPC16400 includes a number of functional blocks in addition to the basic HPC core, as shown in Figure 1. There are 256 bytes of user RAM, a UART, two HDLC controllers with a decoder and four separate DMA channels, and four parallel ports in different shapes and sizes.

Besides the 256 bytes of on-chip RAM, the chip can support up to 544 kbytes of off-chip extended addressing space. Facilities are provided for shared-memory architectures and direct memory access. These features permit the chip to reside on the same memory bus as another processor and allow fast memory

Figure 1

The HPC16400 includes a number of functional blocks in addition to the basic HPC core, including 256 bytes of user RAM, a UART, two HDLC controllers with a decoder and four DMA channels, and four parallel ports.



National Semiconductor Corp.

reads and writes. To reduce cost, the chip doesn't contain user ROM.

The on-chip UART communicates with terminal equipment. Speed is user-settable from 8 bps to 208.3 kbps. Standard UART protocols are supported.

The two HDLC controllers can support an ISDN BRI D channel at the standard rate of 16 kbps. The first HDLC can instead be used to support the ISDN BRI B channel at the standard 64-kbps speed. Each HDLC can in fact support data rates up to 4.64 Mbps, making the chip usable in Ethernet applications.

The serial decoder, which ties the two HDLC units to a common external link, allows the HDLC channels to be used with single-line X.25 packet-switching protocols for point-to-point and multipoint data exchanges.

The parallel ports offer a great deal of system flexibility. Port A is a 16-bit multiplexed address/data bus for program and data memory access. Port B has 12 bits of multiplexed address/data lines, for direct-mapped access to external I/O devices, and four bus control lines. The I and D ports contain the interrupt, Microwire, UART, and HDLC control lines. The R port is a general-purpose eight-bit I/O port that can be used for virtually anything.

THE SUPPORT SOFTWARE

What makes this chip really unique is the support software. National Semiconductor has done everything within reason to make this chip easy to use.

The software package features six main modules for executive, I/O driver, data link, network, call control, and trace functions.

The HPC executive provides the operating environment and general support services. Intertask communication is by mail messages deposited in mailboxes and by semaphores. The executive module examines each mailbox in turn and exercises the requisite activity if the mailbox contains any mail from another process. The modular structure of the code separates code development into discrete units, each of which can be developed and debugged separately.

The I/O drivers are on the lowest tier and control the Microwire, UART, data memory, and HDLC hardware. They also take care of system initialization, frame reception and transmission, and error handling. Since X.25 support is advocated on the data link layer, IEEE 802.3 standards should be attainable through additional hardware support.

The data link software implements the full LAPD data channel and LAPB control signal protocol as described in the CCITT Q.921 standard and X.25 link access procedures. The software provides error-free in-sequence message processing, transmission, and multiplexing.

The network layer module implements the protocol control procedures of Q.931—specifically version T1/D1.2/87-171, ratified in May 1988—to set up, answer, suspend, resume, and disconnect a call.

The remaining two software modules are development tools. The call control module allows the software engineer to emulate ISDN phone call placement and reception during code development and debugging. Similarly, the tracer module allows the engineer to monitor the operation of the software via a terminal attached to the on-chip UART.

ISDN PROTOCOL SUPPORT

Understandably, there's still a certain amount of flux in the ISDN standard, and it's natural to ask how National Semiconductor fits into the picture. The ISDN standard is a moving target, and any company that embodies it in silicon at this early date runs the risk of obsolescence as the standard evolves.

Layers one and two of ISDN (physical and data link) are pretty well established at this point, and the chip is reported to conform fully to the established standards. Layer three of the ISDN protocol, however, is still undergoing some change. National Semiconductor therefore supports circuit-switched call establishment and clearing on both BRI B channels. Packet-switched call establishment and clearing, because they are still subject to revision, aren't supported by the HPC ISDN software package.

Since this layer of system activity is totally under software control, however, it should be possible to adapt the HPC-16400 to any protocol changes. The processor speed seems adequate to cope with most requirements, and the company is making the source code available for all the ISDN software interface modules. The user can therefore develop packet-switched call control from the circuit-switched call control procedures offered in the network layer software module.

By using a separate socketed EPROM for program storage, subsequent standard revisions can be accommodated by changing only one chip. Alternatively, because the shared-memory architecture of the HPC16400 permits the processor to co-reside in a larger computer system, programs could be bootstrapped from some secondary storage medium such as a floppy disk.

It's only a matter of time before all the ISDN functions are fully standardized. Versions of chips like the HPC16400 will then inevitably appear with full ISDN support embedded in on-chip ROM. Until that time (which could still be years away), the modular approach adopted by National Semiconductor for this product may be the most efficient way to use the manifold advantages of ISDN.

DEVELOPMENT SUPPORT

National Semiconductor provides a complete development system for the HPC-16400 including on-line support and full documentation. The chip itself is $24 each in 10,000-piece quantities. A demonstration disk of the system is available for $50.

The IBM PC development board for the chip contains a multitasking executive in on-board ROM that can set up a call between two boards and monitor the software while it's placing and handling a call. The board costs $500; in-circuit emulation for it is an additional $1,500.

The multitasking executive is $5,000 for the object code and $10,000 for the source code. A generic BRI module is also available, as are certified BRIs for the Northern Telecom VM100 switch and the AT&T 5ESS protocols. The first BRI purchased is $10,000 for the object code and $20,000 for the source code; additional BRIs are half price.

Ernest Meyer is an independent technical writer and former editor of VLSI Systems Design. *He holds a B.Sc. in philosophy and psychology from Oxford University. Meyer also writes* Embedded Systems Programming's *At the Bench column.*

Embedded Gallery

DEBUGGING TOOLS

[redacted]

Simulator-debuggers

Mecklenburg Engineering
P.O. Box 744
Chagrin Falls, Ohio 44022
(216) 338-1900

CIRCLE #184 ON READER SERVICE CARD

Mecklenburg Engineering has introduced a line of simulator-debuggers for a variety of eight-bit microprocessors. The $95 development tools are available for 63xx, 65xx, 68xx (including the 65C02 and 68HC11), 8085, 8048, 8051, and Z80 processors. The debuggers allow continuous execution, single-stepping, and breakpoints. Programmers can access machine registers and absolute memory locations.

REM86

Development Associates
1520 S. Lyon St.
Santa Ana, Calif. 92705
(714) 835-9512

CIRCLE #180 ON READER SERVICE CARD

REM86 is a $479 remote target debugger for Development Associates' FUTURE86 systems development language for 80x86 CPUs. REM86 consists of two parts: a target monitor that requires less than 2 kbytes of memory space and a host-based control program that is functionally similar to FDT86, DA's host debugger.

REM86 comes with an immediate-execution patching assembler and supports file loading, disassembly, single-stepping, and a variety of breakpoints. Full symbolic support is provided for applications written in FUTURE86.

[redacted]

EMX96

Annapolis Micro Systems Inc.
612 Third St., No. 301
Annapolis, Md. 21403
(301) 269-8096

CIRCLE #185 ON READER SERVICE CARD

The EMX96 is a $2,995 in-circuit emulator for 8096-based applications. The system allows multiple hardware breaks on instruction fetch, data read, write, address, or pattern-match. The emulator requires an IBM PC, XT, AT, or compatible. It supports host processor speeds from 3 MHz to 12 MHz and target processor speeds from 3 MHz to 18 MHz. Symbolic debugging support is provided for ASM96, PL/M96, and C96.

[redacted]

SoftScope III

Concurrent Sciences Inc.
P.O. Box 9666
Moscow, Idaho 83843
(208) 882-0445

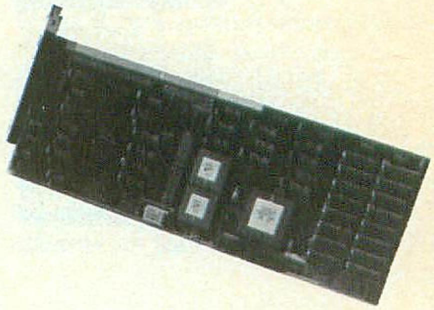
CIRCLE #187 ON READER SERVICE CARD

SoftScope III is a PC AT-hosted source-level debugger for 80376 and 80386 target applications. The system communicates with a firmware monitor via serial connection. Target programs can be transferred from host to target at 100 kbytes per minute.

SoftScope III's source-level interface lets developers simply type symbol names to display all variable types—including C and PL/M structures, multidimensional arrays, and bit fields—in a formatted display. The debugger's command language lets programmers create C-like macros and use C syntax with commands. The suggested retail price is $1,500 in quantity one.

BOARDS

[redacted]



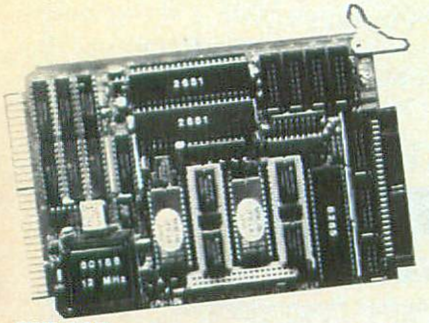
ASPI Banshee System

Atlanta Signal Processors Inc.
770 Spring St.
Atlanta, Ga. 30308
(404) 892-7265

CIRCLE #181 ON READER SERVICE CARD

The Banshee System is a PC AT-compatible plug-in board for development and testing of Texas Instruments TMS320C30 DSP applications. The $4,995 package includes a mother board and development software; add-on daughter boards for memory and I/O functions are available separately.

Two memory options are available. Dual-access memory is 32 kbytes to 512 kbytes of 35 ns static RAM that can be accessed by the AT host or by the TMS320C30 processor with the AT stealing cycles from the DSP. Dual-port memory is 90 ns static RAM that can be accessed by the AT and the DSP simultaneously with no wait states for either processor. The mother board holds 8 kbytes of dual-port memory.



CPU-186

Computer Dynamics Inc.
107 S. Main St.
Greer, S.C. 29651
(803) 877-8700

CIRCLE #186 ON READER SERVICE CARD

CPU-186 is an 80C186-based STD bus single-board computer for embedded control applications. The $476 (in OEM quantities) board features as much as 512 kbytes of battery-backed static RAM, 256 kbytes of EPROM, operating speeds up to 12 MHz, and an SBX interface. An 8087 coprocessor daughter board is available, as is a complete set of DOS-hosted software development tools.

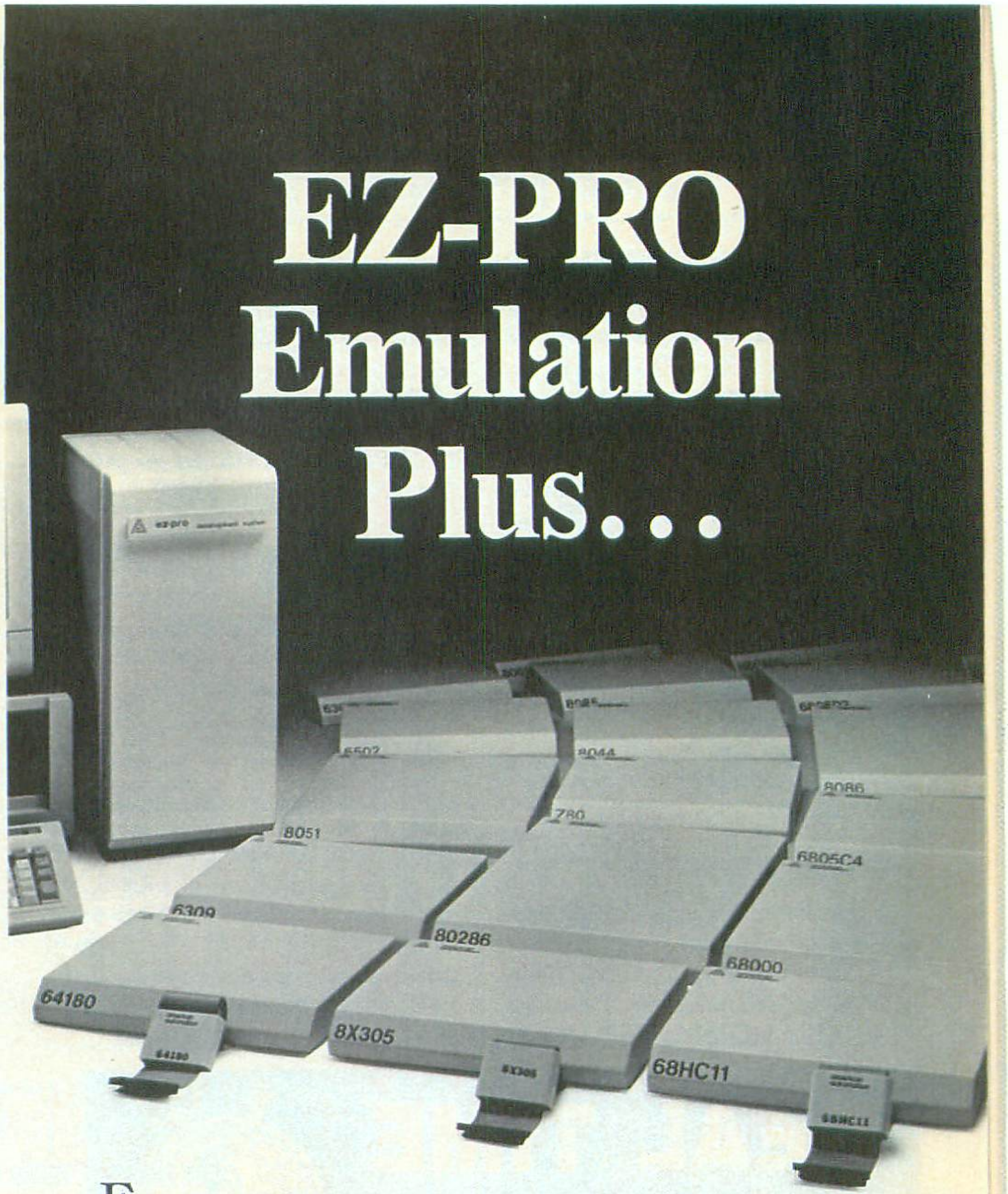
LANGUAGES

chipFORTH 8051

Forth Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, Calif. 90266
(213) 372-8493

CIRCLE #183 ON READER SERVICE CARD

Forth Inc.'s chipFORTH 8051 is a comprehensive development system for embedded Intel 8051 applications. The interactive system, which includes a multitasking operating system executive, allows mixed Forth and assembly language programming. It includes a target Forth compiler and macro assembler, plus a Cavendish Automation prototyping board with 16 kbytes of configurable RAM/EPROM and 8 kbytes of static RAM.



chipFORTH 8051 uses an IBM PC or compatible as host to write 8051 code and download it to the chip via a serial link. Using the PC as a virtual terminal, programmers can execute code on the target and see the results on the host in real-time. Forth Inc. claims this eliminates much of the need for an expensive ICE for development and debugging purposes.

[redacted]

29000 C, assembler

Microtec Research Inc.
3930 Freedom Circle, No. 101
Santa Clara, Calif. 95054
(408) 733-2919

CIRCLE #188 ON READER SERVICE CARD

Microtec has introduced an integrated software development toolset for the AMD 29000 family of 32-bit RISC processors. The toolset consists of the MCC29K globally optimizing C compiler, the ASM29K relocatable macro assembler, and the XRAY29K source-level debugger.

The toolset is available for a variety of hosts: Sun workstations, VAXes running ULTRIX, and IBM PCs and compatibles. The PC toolkit retails for $3,500.

[redacted]

Pascal-2/VMEPROM Link 10

Oregon Software
6915 S.W. Macadam Ave.
Portland, Ore. 97219
(503) 245-2202

CIRCLE #191 ON READER SERVICE CARD

Oregon Software and Regensdorf, Switzerland-based W. MOOR AG have introduced the VMS-based Pascal-2/VMEPROM Link 10, a library that lets programmers connect either VMEPROM or PDOS kernel commands in application programs written in Pascal.

The package includes the library, a Pascal-2 cross-compiler, and (optionally) Oregon's Pascal-2 native compiler. All run on DEC systems, including the MicroVAX and MicroVAX 3500.

KERNELS

[redacted]

MTOS-UX/Ada

Industrial Programming Inc.
100 Jericho Quadrangle
Jericho, N.Y. 11753
(516) 938-6600

CIRCLE #182 ON READER SERVICE CARD

Industrial Programming Inc. has released MTOS-UX/Ada, a complete run-time support executive for TeleSoft Ada. MTOS-UX/Ada replaces TeleSoft's run-time executive and supports Ada primitives plus access to the full range of MTOS-UX operating system services via Ada packages. The operating system supports parallel processing; as many as 16 tightly coupled CPUs may share the system bus—including mixed 68000, 68010, 68020, and 68030 processors.

Other MTOS-UX/Ada features include dynamic object creation, a UNIX interface, mixed-language programming support, and a range of interprocess communication functions.

[redacted]

C EXECUTIVE 29K

JMI Software Consultants Inc.
P.O. Box 481
Spring House, Pa. 19477
(215) 628-0846

CIRCLE #189 ON READER SERVICE CARD

JMI has introduced a version of its C EXECUTIVE operating system for the Advanced Micro Devices 29000 RISC microprocessor. The new version of the multitasking operating system is compatible with other C EXECUTIVE releases, minimizing portability problems for designers who seek to migrate from CISC to RISC.

want to know more about the products advertised in this issue?

Circle the corresponding number on the postage-paid reader service card, fill in the necessary information on the card and drop in the mail. Literature will be mailed to you free of charge by the manufacturer.

FREE PRODUCT INFORMATION

Embedded Systems PROGRAMMING

READER INQUIRY CARD

PLEASE PRINT

□ MS.
□ MR.

Name ______________________ Title

Company ______________________

Address ______________________

Dept./Mail Stop ______________________ Phone (   )

City ______________________ State ______ Zip

□ Please send me one year (12 issues) of EMBEDDED SYSTEMS PROGRAMMING. Please bill me: U.S. $37; Canada, Mexico $43 surface mail; Other foreign $52 surface mail, $77 air mail.

PREMIER ISSUE — EXPIRES MARCH 15, 1989

101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260
261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320
321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340
341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360

A. WHAT INDUSTRY DO YOU WORK IN? (CHECK ONLY ONE)

1. □ Consumer electronics/appliances
2. □ Computers/peripherals
3. □ Avionics/marine/space/military electronics
4. □ Communications systems/equipment
5. □ Automotive
6. □ Industrial control/systems/robotics
7. □ Medical electronic equipment
8. □ Test & measurement equipment
9. □ Independent software contractor/developer

B. NUMBER OF EMPLOYEES AT YOUR WORK LOCATION:

1. □ under 20
2. □ 20–99
3. □ 100–249
4. □ 250–999
5. □ 1000 or more

C. DO YOU WORK FOR OR DOES YOUR COMPANY RESELL PRODUCTS OR SERVICES TO THE GOVERNMENT?

1. □ YES
2. □ NO

D. REQUESTED PRODUCTS PLANNED FOR PURCHASE WITHIN

1. □ 0–3 months
2. □ 3–6 months
3. □ 6–12 months

E. WHAT IS YOUR PURCHASE AUTHORITY? (CHECK ALL THAT APPLY)

1. □ evaluate products
2. □ recommend purchase
3. □ specify vendor
4. □ approve purchase

Comments ______________________

Embedded Systems PROGRAMMING

READER INQUIRY CARD

PLEASE PRINT

□ MS.
□ MR.

Name ______________________ Title

Company ______________________

Address ______________________

Dept./Mail Stop ______________________ Phone (   )

City ______________________ State ______ Zip

□ Please send me one year (12 issues) of EMBEDDED SYSTEMS PROGRAMMING. Please bill me: U.S. $37; Canada, Mexico $43 surface mail; Other foreign $52 surface mail, $77 air mail.

PREMIER ISSUE — EXPIRES MARCH 15, 1989

101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260
261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320
321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340
341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360

A. WHAT INDUSTRY DO YOU WORK IN? (CHECK ONLY ONE)

1. □ Consumer electronics/appliances
2. □ Computers/peripherals
3. □ Avionics/marine/space/military electronics
4. □ Communications systems/equipment
5. □ Automotive
6. □ Industrial control/systems/robotics
7. □ Medical electronic equipment
8. □ Test & measurement equipment
9. □ Independent software contractor/developer

B. NUMBER OF EMPLOYEES AT YOUR WORK LOCATION:

1. □ under 20
2. □ 20–99
3. □ 100–249
4. □ 250–999
5. □ 1000 or more

C. DO YOU WORK FOR OR DOES YOUR COMPANY RESELL PRODUCTS OR SERVICES TO THE GOVERNMENT?

1. □ YES
2. □ NO

D. REQUESTED PRODUCTS PLANNED FOR PURCHASE WITHIN

1. □ 0–3 months
2. □ 3–6 months
3. □ 6–12 months

E. WHAT IS YOUR PURCHASE AUTHORITY? (CHECK ALL THAT APPLY)

1. □ evaluate products
2. □ recommend purchase
3. □ specify vendor
4. □ approve purchase

Comments ______________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 760 PITTSFIELD, MA

POSTAGE WILL BE PAID BY ADDRESSEE

Embedded Systems
PROGRAMMING
Reader Service Department
P.O. Box 5322
Pittsfield, MA 01203

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 760 PITTSFIELD, MA

POSTAGE WILL BE PAID BY ADDRESSEE

Embedded Systems
PROGRAMMING
Reader Service Department
P.O. Box 5322
Pittsfield, MA 01203

Advertiser Index

PG	ADVERTISER	RS#
91	Adams MacDonald Enterprises Inc.	253
90	Advin Systems Inc.	244
90	Aldia Systems Inc.	241
15	Alsys	209
87	American Automation	238
67	Annapolis Micro Systems Inc.	229
16-17	Applied Microsystems Corp.	210
49	Archimedes Software Inc.	223
91	BP Microsystems	249
92	Bryte Computers Inc.	256
76	BUSCON/89-West	235
Cov2	Cadre Technologies Inc.	201
18	Datalight	211
92	Elan Digital Systems	255
51	Eyring	224
33	First Systems Corp.	218
45	Forth Inc.	*
91	Genesis Microsystems Corp.	248
91	Grammar Engine Inc.	251
53-56	Harris Corp.	225
35	HILEVEL Technology Inc.	219,220
21	Huntsville Microsystems Inc.	213
25	Industrial Programming Inc.	216
12	Intel	208
73	Intermetrics	234
89	Introl Corp.	240
20	JMI Software Consultants Inc.	212
95	KADAK Products Ltd.	260
67	Laboratory Microsystems Inc.	231
91	Logical Systems Corp.	247
Cov4	Manx Software Systems Inc.	261

PG	ADVERTISER	RS#
90	Micro Interfaces Corp.	245
91	MicroComputer Tools Co.	252
78	Microcosm Inc.	236
92	Microprocessor Report	270
4	Microtec Research Inc.	204
90	NCI	243
91	Needham's Electronics	250
81	New Micros Inc.	237
95	Nohau Corp.	258
11	Oasys	207
88	Oregon Software	239
63	Phar Lap Software Inc.	227
2	Polytron Corp.	203
1	Quantum Software Systems Ltd.	202
26-27	Ready Systems Corp.	217
67	Sierra Systems	230
72	Signum Systems Inc.	233
37	Soft Advances	221
91	Softaid Inc.	254
68-69	Software Development '89	232
Cov3	Software Development Systems Inc.	*
6-7	Sophia Systems	205
24	Systems & Software Inc.	215
23	2500 A.D. Software Inc.	214
90.95	US Software	246,257
63	V Communications	226
95	Vesta Technology Inc.	259
65	Whitesmiths Ltd.	228
37.90	Z World	222.242
8	ZAX Corp.	206

The index on this page is provided as a service to our readers. The publisher does not assume any liability for errors or omissions.

* These advertisers prefer to be contacted directly.

JOIN THE ADVERTISERS HALL OF FAME

You too can join the list of illustrious companies that are selling their products to over 25,000 software engineers, project leaders, and embedded developers in *Embedded Systems Programming*. As the only publication dedicated to embedded systems, it is the most targeted and cost-efficient way to reach your best prospects. Call today for more information and a media kit!

East/Midwest
(212) 683-9294
Jo Ben-Atar
Nancy Schnarr

West
(415) 995-2431
Ted Bahr
Robert Murphy

February issue
Reservations deadline: 12/2/88
Materials deadline: 12/9/88

March 1989 deadlines
Reservations: 1/2/89
Materials: 1/9/89

Embedded Marketplace

Want to know more about products and services displayed here? It's simple: just circle the Reader Service number on the response card bound into this issue. Each advertiser will send you more information free of charge!

Embedded Marketplace is a special showcase section reserved for advertisers with standard 1/9th-page display ads (2¼ x 3 inches). Cost are $395 (1x), $375 (6x), $355 (12x), and $340 (24x). For more information on how to sell your products to over 25,000 software engineers, project leaders, and embedded developers, call Jo Ben-Atar or Nancy Schnarr at (212) 683-9294 (East), or Rob Murphy at (415) 995-2475 (West).

IT'S EASY TO LOCATE ROM SOFTWARE SUPPORT

PC-LOCATE: Produce ROM-able code from your "exe" files. PC-LOCATE assigns physical addresses to the re-locatable image based on user inputs. PC-LOCATE supports the entire Intel processor family including the 8086, 8088, 80186, 80188 and 80286.

PC-PROMPAK: A PROM/ROM expansion board for IBM and IBM-compatible computers. PC-PROMPAK provides up to 384 Kbytes of expansion memory and supports most 28-pin JEDEC compatible devices.

PC-ROMDRIVE: Create a "Diskless PC" that can include MS-DOS and your application program. Program execution can be automatically invoked through the use of an "autoexec.bat" file.

ALDIA SYSTEMS INC.

P.O. BOX 37634 PHOENIX, AZ. 85069
(602) 866-1786

CIRCLE #241 ON READER SERVICE CARD

Z80 and HD64180

We have the development aids and tools you need:

- PC-based C cross compiler.
- Cross assembler and linker.
- Remote Debugger.
- In-circuit emulator.
- Single board computer.
- PC Coprocessor with CP/M-80 and ISIS emulators.

Z-World
1772 Picasso Ave, Davis, CA 95616
(916) 753-3722
Fax: 753-5141

CIRCLE #242 ON READER SERVICE CARD

25MHz 48 CHANNEL
PC-BASED LOGIC ANALYZER
$1595.⁰⁰ + POD PRICE

- ■ 48 Channels @25MHz X 4K word deep
- ■ 16 Trigger words/16 trigger sequence
- ■ Automatic set-up and loading of symbol tables
- ■ Comparison of Reference and Trace Memories
- ■ Storage and recall of trace data to disk
- ■ 12 Channels @ 100MHz w/Glitch Detection Pod
- ■ Disassemblers available for: 8088, 8086, 68000, Z80, 8085, 6502, 6801, 6303, 8031, 6809, 6811

NCI □ 6438 UNIVERSITY DRIVE,
HUNTSVILLE, AL 35806
(205) 837-6667

CIRCLE #243 ON READER SERVICE CARD

You need a Sailor-8 even if you already have another EPROM programmer.

★ ★ ★ ***DISCOVER WHY:*** ★ ★ ★

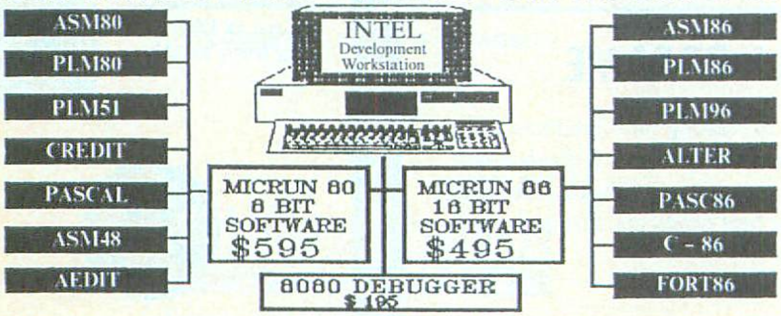
Sailor-8 is beloved and recommended by software engineers. *"Once you used a Sailor-8, you won't be able to stand anything else."*

Please call for free demo disk.

ADVIN SYSTEMS INC. Sunnyvale, CA 94086
(408) 984-8600 • FAX (408) 736-2503

CIRCLE #244 ON READER SERVICE CARD

RUN INTEL SOFTWARE ON YOUR PC!



NOW YOU CAN DEVELOP & DEBUG INTEL CODE ON YOUR PC

- **Work up to 5 Times faster on your PC**
- **No more waiting in line for the Intel "Blue Box"**
- **No more costly Intel service calls**

For Information call: **1-800-888-8086**

Micro Interfaces Corp. 16359 NW 57th Ave. Miami, FL 33014

CIRCLE #245 ON READER SERVICE CARD

8086 8051 8096

ROYALTIES

Multi-Tasking EXECS

U S Software offers hi-performance software tools for embedded applications.

Get the full details by calling:

800-356-7097

U S SOFTWARE.

United States Software Corporation
14215 NW Science Park Drive, Portland, OR 97229
(refer to our other ad in this issue for int'l orders/other product information)

6800 68HC11 Z80

CIRCLE #246 ON READER SERVICE CARD

SC87C451 PROGRAMMER $125.

Logical Systems now brings you support for the Signetics SC87C451. The UPA451N programs this expanded I/O microcontroller on programmers that support the 87C51 or 8751H. To use your general purpose programmer combine the UPA451N and the UPA87C51. Logical Systems, helping you get the most out of your programming equipment with our growing line of adapters. OEM inquiries welcome.

ADAPTER	PROGRAMS	PRICE
UPA8751	C8751, 8751H, AMD8753H, 8744	$ 95.00
UPA87C51	C8751, 8751H, AMD8753H, 8744 87C51, 87C51FA	125.00
UPA63701V	Hitachi HD63701V0	65.00
UPA63701X	Hitachi HD63701X0 (64 pin shrink dip) -L Low insertion force socket -Z Textool ZIF socket	 95.00 149.00
UPA73701Y	Hitachi HD63701Y0 (64 pin shrink dip) -L Low insertion force socket -Z Textool ZIF socket	 95.00 149.00
UPA63705V	Hitachi HD63705V0	65.00

CALL (315) 478-0722 or FAX (315) 475-8460

LOGICAL SYSTEMS CORPORATION
P.O. Box 6184, Syracuse NY 13217-6184 USA, TLX 6715617 LOGS

CIRCLE #247 ON READER SERVICE CARD

ROM TOOLS

Develop ROM code from Microsoft-C and MASM with

GeneLink™

- **Independently locate segments, classes, and groups.**
- **Complete debugging records for emulators and GeneScope/Target™ ROM debugger.**
- **Automatic initialization of registers and initialized variables.**

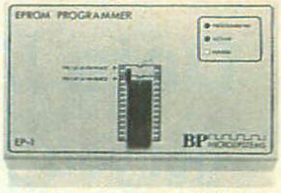
Genesis™
Microsystems

Genesis Microsystems Corporation
13300 Sunset Dr., Los Altos, CA 94022
(415) 964-9001

CIRCLE #248 ON READER SERVICE CARD

NEW FASTER

EPROM PROGRAMMER



- Programs 2764 in 8 sec.
- Reads, programs, copies over 475 devices from 35 mfg's: 2716-27513, 27011, 68764, 68766, 2804-28256, etc
- Optional microcontroller heads support 874x and 87C51 series
- Improved design now programs over twice as fast
- Connects via RS-232 to any computer: PC, XT, AT, PS/2, Mac, etc.
- Supports XMODEM/XMODEM CRC protocols & ASCII file xfers
- Supports Intel, Motorola, hex-space, straight hex and binary files
- 30-day money-back guarantee
- Engr support team for fast updates
- One-year warranty (parts and labor)
- Toll-free technical support
- Collates 16 & 32-bit data
- Gold Textool ZIF IC socket
- 8 baud rates to 38,400
- Same day shipment
- Thousands of satisfied customers attest to the EP-1's great value
- Low price of **$349** includes IBM compatible communications program, user's manual and two free firmware update coupons

CALL TODAY FOR MORE INFO 1-800-225-2102

BP MICROSYSTEMS
10681 Haddington #190, Houston, TX 77043
(713) 461-9430 FAX (713) 461-7413

CIRCLE #249 ON READER SERVICE CARD

DEVICE PROGRAMMER
$550 $750

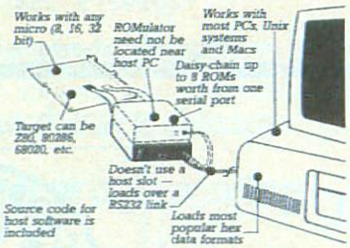
1 Megabit of DRAM, User upgradable to 32 Megabit **4x20 LCD Display**, .3, .6 Zif socket, RS232, PARALLEL in and out 20 Key tactile keypad (not membrane). 32K internal EEPROM **(easy firmware upgrades) QUICK PULSE ALGORITHM** (27256 in 5 sec. **1 Megabit** in 17 sec.) Completely **stand alone**, 10 day money back guarantee **2 year warranty**, made in **U.S.A. Technical support** by phone, Complete **manual** and **schematic.**

NEEDHAM'S ELECTRONICS
4535 Orange Grove Ave., Sacramento, CA 95841
Call for more information
Phone (916) 924-8037 FAX (916) 972-9960
VISA/MC

CIRCLE #250 ON READER SERVICE CARD

Still Blasting ROMs?

Develop and test ROM code in minutes without leaving your keyboard — with The ROMulator™.


Works with any micro (8, 16, 32 bit)
ROMulator need not be located near host PC
Works with most PC's, Unix systems and Macs
Daisy-chain up to 8 ROMs worth from one serial port
Target can be Z80, 80286, 68020, etc.
Doesn't use a host slot — loads over a RS232 link
Source code for host software is included
Loads most popular hex data formats

Ask us about faster (than 150ns) ROMulators and custom cables for most non-27xxx ROM sizes.

FOR MORE INFORMATION, PLEASE CALL OR WRITE

Grammar Engine Inc

Grammar Engine, Inc
3314 Morse Road
Columbus, Ohio 43231
614/471-1113

VISA and MasterCard Accepted
Dealer Inquiries Welcome

In California, call
415/595-2252

CIRCLE #251 ON READER SERVICE CARD

CROSS ASSEMBLERS

Macros
PC Compatible
Relocatable
Conditionals
Fast
Reliable ..... **from $150**

also: Disassemblers
EPROM Programmer Board

MICROCOMPUTER TOOLS CO.
Phone (880) 443-0779
In CA (415) 0825-4200
912 Hastings Dr., Concord, CA 94518

CIRCLE #252 ON READER SERVICE CARD

THE UNIVERSAL PROGRAMMER FOR YOUR PERSONAL COMPUTER

SPRINT PLUS

Assembler/Unassembler

A complete software driven E/EE/PROM, PLD Programmer and development tool in a single PC expansion board!

- Handles all popular NMOS, CMOS, and Bipolar technologies in 20, 24, and 28 pin packages. (32 + pin packages available.)
- Pin-programmable for future device support. Simple, low-cost disk updates.
- Includes Assembler, 'Unassembler', pin exerciser, screen editor, logic translator, etc.

Call for **FREE** evaluation disk!

adams-macdonald
ENTERPRISES INC
800 Airport Road
Monterey, California 93940
Tel (408) 373-3607
Telex 882140
Fax (408) 373-3622
800-777-1202

CIRCLE #253 ON READER SERVICE CARD

More Good Code - *Fast!*

Softaid's ICEBOX In-Circuit Emulators provide all the resources needed to complete hardware/software projects on time and on budget. The ICEBOX is an inexpensive yet extremely powerful microprocessor development system.

- Emulators for: 64180, Z80, Z280, Z180, 8085, 8088/8086, 80188/80186, V40/V50
- 65,536 Full speed breakpoints
- Source Level windowed debugging
- Optional Performance Analysis quickly isolates software bottlenecks
- Call for our free "Guides to Applications Development"
- Priced from $595 to $2995

SOFTAID, INC.
8930 Route 108
Columbia, MD 21045
(301) 964-8455
(800) 433-8812

CIRCLE #254 ON READER SERVICE CARD

AN EMBEDDED SUPERMARKET!

Make sure your product is on the shelves when 25,000 software engineers, project leaders, and embedded developers do their shopping! The **Embedded Marketplace** is the most targeted and cost-effective way to reach your best prospects. Call today for more information!

East
(212) 683-9294
Jo Ben-Atar
Nancy Schnarr

West
(415) 995-2475
Rob Murphy

by P.J. Plauger

Rating Reference Manuals

If there's a university course on embedded systems programming, I've not heard of it. Sure, there are various offerings on operating systems design and courses dealing with the special issues of real-time programming. But I don't know of any school that will prepare you to go out into the world ready to deal with all aspects of designing, building, and programming embedded systems.

In the absence of a clear academic discipline, we have each of us learned our trade the hard way. We take on a project that requires us to cross that great divide between hardware design and programming. We find ourselves without an adequate operating system, or at least without all the components of one ready-made. Willy-nilly, we become systems programmers for a day. Except that day often turns into half a year.

More often than not, our employers pay our tuition for this on-the-job training. The coin of that tuition is not just money that can be budgeted and counted against a particular project. One form of currency is the extra time it takes us to get up to speed on a project that stretches our skills. A more insidious form is the loss of business due to late shipments, fixing bugs on the customer's premises, or redesigning to catch up with the features of competitors' products.

As we grow into senior engineering, management, or entrepreneurial positions, more and more we begrudge paying that tuition. We ourselves must scurry to stay even with the technological juggernaut. We must also train subordinates in the skills we had to learn the hard way. In either case, we find ourselves yearning for neatly encapsulated training or, at the very least, knowledge.

There are always folks out there sniffing the wind for any form of yearning. If these folks can convince us to part with some of our hard-earned cash in the hopes of satisfying our yearning, you can bet they'll try. They'll write textbooks and trade books. They'll offer seminars in hotel meeting rooms and in your very own conference rooms. They'll provide you with reference cards, help menus, videotapes, and expert consulting.

Some will be skilled technicians and mediocre educators; some will be the reverse. Too many will have neither skill, and a scant few will have both. Some will be saints preaching their version of the one true gospel. Some will be charlatans. Most will be hard-working folk like you and me hoping to make a buck by offering a cost-effective service.

In the absence of academic discipline we have learned our trade the hard way, taking on projects that require us to cross that great divide between hardware design and programming.

THE GOOD AND THE BAD

The good thing about training in embedded systems programming is that more and more people are trying to satisfy the yearning for it. Hardly a day passes that I don't see yet another book or seminar with a tantalizing title. Embedded programming, real-time systems, small operating systems, RISC, CISC, TCP/IP, Ada, POSIX, free-standing C—all are buzzwords that instantly turn my head. As a result, I currently own a stack of some 30 books, each of which offers to tell me what I need to know about this trade that has fascinated me for years.

The bad thing about training in embedded systems programming is that I currently own a stack of some 30 books, each of which offers to tell me concisely what I need to know about this trade that has fascinated me for years.

I have no more free time than any of you. This hurly-burly we call the computer industry catches us up in its maelstrom, shakes the spare seconds out of our pockets, and leaves us stranded on an islet surrounded by priorities. How the hell am I to know which books are worth reading? If I listed all the books without remark, how would you know which are worth reading to meet your needs?

The purpose of this column is to give you a break. I'm willing to chew through my ever-growing stack of books and read them (or at least skim them) for you. It's something I do all the time anyway as part of keeping up with my profession. The added work is to provide an occasional rambling narrative, such as this one, to put a few books into perspective and summarize what they may have to offer you.

I use the term *books*, by the way, in the most general sense. I also consider computer reference manuals, language standards, and the documentation shipped with commercial products to be fair game for review. I even hope to review the odd seminar, although that's a bit chancier. In short, I consider grist for the mill any source of knowledge about embedded programming that you might have to digest or evaluate.

If you've read any of my Programming on Purpose columns in *Computer Language*, you already know that I tend to be eclectic, opinionated, and occasionally wrong. I have no problem admitting to any of those vices/virtues, so long as I succeed in providing a useful service to readers. If you don't know about me, however, here's a thumbnail sketch of my credentials for this particular endeavor.

I learned how to program embedded

systems the hard way starting in the early sixties. My academic career as a nuclear physicist centered largely around computerized data acquisition and control. I've written operating systems both general-purpose and dedicated to specific tasks. I've programmed peripheral controllers and data switches. What I learned was by imitation. What I invented was often a rehash of existing technology, a fact of which I was sadly unaware until much later.

I've also worked the other side of the street. I've written textbooks and taught commercial seminars, though not always specifically aimed at embedded programming. I've written magazine articles and tutorials. I've lectured and consulted in more venues than I can possibly remember. How much good I've done, I'm not terribly sure.

In any event, I feel that I know what would have been helpful to me as I learned this stuff. I know what I'd like to have today, for myself and for the people who work for me. I also feel that I know how hard it is to teach this subject or to sell commercial software that helps develop embedded programs. It ain't easy to satisfy the yearning for knowledge, but it is possible.

READING TO WRITE

With that as a preamble, I'd like to begin at the same place many of you began your careers as programmers of embedded systems. Often the only literature you can consult before you set out is the programmer's reference manual for the CPU you'll be using in the delivered product. Here's where you begin to find out how much trouble you've gotten yourself into.

A microprocessor has four distinct aspects that the vendor must describe well enough that you can design products around it:

1. To use a microprocessor as an analog device, you need to know size and pinout, power drain and heat dissipation, clock speeds and signal tolerances. If it can't stand the heat, you can't use it in the kitchen (or under the hood). Maybe you can clock it at 20 MHz, but only if you can tolerate five errors per day of operation. Otherwise, you'd better know and honor its limitations.

2. To use a microprocessor as a digital device, you need to know data formats, clock phases, and signaling protocols. Even if you can pretend that all your logic chips are pure digital devices (silly you), you can't interface memory or other devices to the microprocessor unless you know how the bits fly.

3. To program the operating system for a microprocessor, you need to know how it handles power-on, traps, interrupts, and memory management. You also need to know how to control timers and various peripheral devices. All that stuff that DOS or UNIX does for you, you have to handle for yourself.

4. To write applications for a microprocessor, you need to know data formats, the set of valid instructions, and the valid addressing modes for each instruction. You also need to know register layouts, sensible function-calling sequences, and how to inhibit or handle exceptions.

If you write only applications hosted under a popular operating system, you can confine your attention to aspect 4. A machine-specific operating system such as PC-DOS probably models its user exception-handling machinery after the hardware trap and interrupt requirements, so you may touch on aspect 3 as well. Most of that and the preceding aspects, though, is largely hidden from you. That's the purpose of off-the-shelf operating systems and computers.

If, on the other hand, you find yourself writing free-standing code, then you're in the business of writing operating systems. You may write only a tiny operating system compared to a UNIX or even a PC-DOS, but it must touch all the bases nonetheless. For such an enterprise, you must know all about aspects 3 and 4. And if you must control a few peripheral devices in the bargain, then likely enough you must touch on aspect 2 as well.

People who build processor boards and complete boxes must be thoroughly at home with aspect 2. They can be largely ignorant of aspect 4 and only dimly aware of aspect 3. How much such people must know about aspect 1 depends upon how hard they're pushing the state of the art and how much design control they have over all the system electronics.

People who live in a world where aspect 1 is most important are often ignorant of any issues beyond aspect 2. Many modern processor architectures are available in a number of different form factors. The circuit designer can often select a cheap plastic part for economy, a CMOS version for low power drain, or a hardened version for military applications. In sooth, the jobs of the applications programmer and systems programmer are often unchanged across different form factors. The separation of concerns between hardware and software designers often benefits productivity.

The potential audience for information about each of these aspects grows dramatically as you proceed from aspect 1 to aspect 4. There are more people who assemble boards into systems than there are people who design boards and boxes. There are more people who program boards and boxes than there are people who build them. There are *far* more people who program under operating systems than there are people who write operating systems.

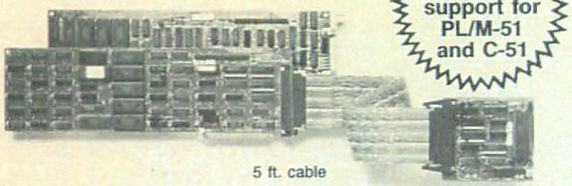
All these considerations color the way I view a reference manual for a microprocessor. I judge the usability of a reference manual by:

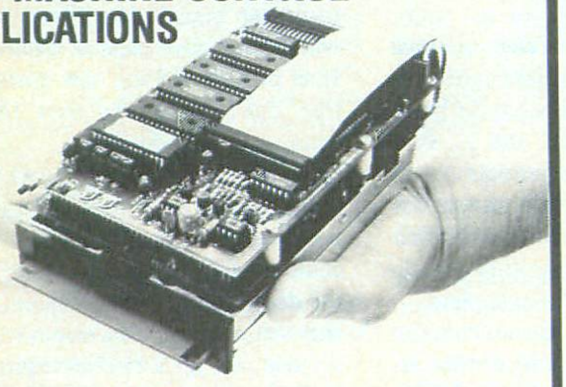
- How well it presents the four aspects.
- How clearly it distinguishes among the four aspects in addressing the differing needs of various constituencies.
- How smoothly the applications programmer is steered to matters from aspect 4.
- How smoothly the systems programmer is directed to the additional matters from aspect 3.

Two of the most important processor families for embedded applications are the Motorola 680x0 and the Intel 80x86. Let's see how they stack up by these criteria.

Motorola, Inc., *M68000 16/32-Bit Microprocessor Programmer's Reference Manual,* Prentice-Hall, Englewood Cliffs, N.J. (1979 and various later editions).

The first Motorola 68000 manual I read contained bus timing diagrams and, I believe, pin-outs and chip characteristics as well. The various aspects of the chip were more or less separated in different chapters, but the presentation was all stirred together. It's an interesting sign of our times that Prentice-Hall saw fit to publish a microcomputer reference manual. (I've even seen it in hardcover.) At least the part of the manual that's of interest to programmers. Along the way, all but the architectural considerations (aspects 3 and 4) disappeared.


5 ft. cable



I think this manual really does try to meet the criteria outlined earlier. Where it fails is where it persists in copying traditional presentations. For example, Figure 1.1 shows the User Programmer's Model, which is really the registers visible to a program running in user mode. That's all well and good, but the next three figures bull right into information that only a systems programmer could love. I also can't assert that any of the figures are adequately supported by descriptions in the text.

The first thing I, as a longtime compiler writer, want to know about a computer architecture is what the address modes are. How do you build addresses and use them to access memory? The second thing I want to know is how general these modes are. What combinations of address modes are valid with which instructions? These considerations are more important to me than the number of bits in a byte, or the number of bytes in various integers, or how the machine encodes integers and floating-point values.

In this regard, the folks at Motorola deserve an E for effort. The 68000 manual presents address modes right off the mark. Machine instructions follow closely on their heels. You get all this information summarized a couple different ways, and you get lots of block diagrams by way of illustration. That's the good news.

The bad news is that the 68000 is a very irregular machine masquerading as a souped-up PDP-11. Most instructions set aside all the bits you need to access one operand with an arbitrary address mode. (A register usually serves as the second operand. Several moves let you write two arbitrary address modes, one for source and one for destination.) The problem is, only some of these modes work for each instruction. Just because the bit pattern exists for using, say, the address register predecrement mode, the hardware won't necessarily do something sensible.

Table B-1 in the manual is an invaluable summary of the various subsets of addressing modes and the names used for the subsets. For each instruction description, pay close attention to the last sentence or two. There, and only there, do you learn just how irregular the 68000 really is. I still carry second-degree burns from my first few years of writing assembly-level support code for C on the 68000. (It seems that Motorola's assemblers have been slow to check for all invalid combinations of instructions and address modes.) Later members of the 68000 family, from 68020 on, have filled in most of the holes in the table of valid modes but have done so by adding more complicated codes, not by giving more sensible definition to the existing codes.

So much for helping the applications programmer. We can only be grateful that excellent C and Pascal compilers are available for this family. As a result, fewer and fewer people need concern themselves with these warts. In fairness to the anonymous authors of the 68000 reference manual, I should emphasize that most of the problems stem from the chip itself. Still, the presentation can and should draw attention to the dangers.

As for systems programmers, they're pretty much on their own here. There are a few pious paragraphs on "Structured Modular Programming" and "Improved Software Testibility *[sic]*." There's a brief essay explaining how Motorola fixed the virtual-memory support in the 68010 and later processors. This is mostly back-patting; you'll find little guidance in laying out the low-memory image of a control ROM.

Intel Corp., *The 8086 Family User's Manual,* Intel, Santa Clara, Calif. (1979 and various later editions).

The Intel 8086 reference manual takes almost the exact opposite approach. It seems to be written primarily for hardware designers (aspects 1 and 2). Programming considerations are almost an afterthought. Intel documentation has repeatedly given me the impression that programming is what those other guys do after the real design work is complete.

On the other hand, the documentation is easy to read and is, I must confess, as complete as I've ever needed. It begins with pin-outs and system block diagrams. It devotes almost as much space to describing support chips as it does the CPU proper, but buries in the middle of this thick document a thorough description of the address modes, instruction set, and interrupt vectors.

You also find real-live essays that actually try to describe what the various design features mean. Sometimes you can even find a bit of justification for why the architecture ended up the way it did. And the material stops well short of being preachy. (I intentionally didn't review the Intel 80286 and 80386 manuals, which can be as pious as early DEC manuals. They leave you with the general impression that computer science was invented by Intel in Aloha, Oregon.)

Of course, the Intel 8086 architecture really is irregular. With its Intel 8080 heritage, it could hardly be otherwise. (See my article "Son of PC Meets the C Monster" in *Computer Language,* Feb. 1987.) That makes it hard to present some topics clearly. Under the circumstances, you could hardly ask for more (at least from a computer manufacturer).

WRITE AND REWRITE

What both manuals sorely need is a complete rewrite by someone skilled in the art. One of the nicest results of the PC revolution is that our standards for adequate documentation are steadily rising. It seems that the future marketplace for both manuals is promising enough to warrant investments by both Motorola, Inc. and Intel Corp. in truly readable prose.

A couple of years ago, it looked like the battle lines were drawn for the last great shootout. There were only a few widely used architectures; Motorola and Intel owned two of them. The only issue was which would win what corners of the computer marketplace. Now there's a whole new set of players. Motorola and Intel own two of the new entrants, but they don't necessarily have the inside track on future adoptions.

I'd dare to suggest to vendors that documentation is important and always has been, though in the past we took what we could get. (Old definition: If they give you the documentation, it's a mainframe. If they sell it to you, it's a minicomputer. If it's sold out, it's a microcomputer.) The battle for the hearts and minds of embedded systems programmers can be won by supplying readable programmer's reference manuals.

P.J. Plauger has coauthored several popular textbooks with Brian W. Kernighan, including The Elements of Programming Style *(New York, N.Y.: McGraw Hill, 1978) and* Software Tools in Pascal *(Reading, Mass.: Addison-Wesley, 1981). He's the secretary of X3J11, the ANSI C standard committee, and president of Whitesmiths, Ltd.*

Embedded systems designers have already used **CrossCode C** *in over 291 different applications.*

How to choose a 68000 C compiler for your ROMable code development

These twelve important **CrossCode** C features could make the difference between success and failure

It's hard to know ahead of time what features you'll be needing in a 68000 C compiler. But if you're using **Crosscode C** you won't need to think ahead, because **CrossCode C** is already equipped with these twelve important features for your ROMable code development:

1. A 100% ROMable Compiler: CrossCode C splits its output into five memory sections for easy placement into ROM or RAM at link time.

2. Integrated C and Assembler: You can write your code in any combination of C and assembly language.

3. Readable Assembly Language Output: The compiler generates assembly language code *with your C language source code embedded as comments*, so you can see each statement's compiled output.

4. Optimized Code: CrossCode C uses minimum required precision when evaluating expressions. It also "folds" constants at compilation time, converts multiplications to shifts when possible, and eliminates superfluous branches.

5. Custom Optimization: You can optimize compiler output for your application because *you* control the sizes of C types, including pointers, *floats*, and all integral types.

6. Register Optimization: Ten registers are reserved for your register variables, and there's an option to automatically declare all stack variables as *register*, so you can instantly optimize programs that were written without registers in mind.

7. C Library Source: An extensive C library containing over 47 C functions is provided in source form.

8. No Limitations: No matter how large your program is, **CrossCode C** will compile it. There are no limits on the number of symbols in your program, the size of your input file, or the size of a C function.

9. 68020 Support: If you're using the 68020, **CrossCode C** will use its extra instructions and addressing modes.

10. Floating Point Support: If you're using the 68881, the compiler performs floating point operations through the coprocessor, and floating point register variables are stored in 68881 registers.

11. Position Independence: Both position independent code and data can be generated if needed.

12. ANSI Standards: CrossCode C tracks the ANSI C standard, so *your* code will always be standard, too.

There's More

CrossCode C comes with an assembler, a linker, and a tool to help you prepare your object code for transmission to PROM programmers and emulators. And there's another special tool that gives you symbolic debugging support by helping you to prepare symbol tables for virtually all types of emulators.

CrossCode C is available under MS-DOS for just $1595, and it runs on all IBM PCs and compatibles (640K memory and hard disk are required). Also available under UNIX & XENIX.

CALL TODAY for more information:

1-800-448-7733
(ask for extension **3001**)

Inside Illinois or outside the United States, please dial

PHONE: **1-312-971-8170**
FAX: **1-312-971-8513**

SOFTWARE DEVELOPMENT SYSTEMS, INC.
DEPARTMENT 31
3110 WOODCREEK DRIVE
DOWNERS GROVE, ILLINOIS 60515 USA

CrossCode™ is a trademark of SOFTWARE DEVELOPMENT SYSTEMS, INC. MS-DOS® is a registered trademark of Microsoft. UNIX® is a registered trademark of AT&T. XENIX® is a registered trademark of Microsoft.

Aztec C ROM Cross Development Systems Produce Fast, Tight C Code with Less Effort

Aztec C ROM Cross Development Systems give you the best results — clean, tight and fast running code. Aztec C systems are available for a variety of targets and for both MS-DOS or Apple Macintosh hosts! And, Aztec C systems come complete with all the tools to edit, compile, assemble, optimize and, now, *source debug* your C code in less time and with less effort.

Quality, tight code that's fast and efficient. An abundance of tools to produce better results in less time. That's why Aztec C systems are the choice of more professional ROM developers.

So when you're looking for the best results, insist on Aztec C ROM Cross Development Systems. Call today and find out more about our complete line of Cross Development Systems.

Supported targets include: the 68xxx family, the full 8086 family, the 8080/Z80 family and the 6502 family of microprocessors.

1-800-221-0440 (outside NJ)

(NJ and Outside U.S.) 1-201-542-2121 Telex: 4995812MANX
Fax: 201/542-8386

Aztec by MANX
SOFTWARE SYSTEMS
One Industrial Way Eatontown, New Jersey 07724

MS-DOS is a registered trademark of Microsoft Corporation
Apple and Macintosh are registered trademarks of Apple Computer Corporation.

CIRCLE #261 ON READER SERVICE CARD