CMP august 2000 www.embedded.com volume 13, number 8

Embedded Systems® PROGRAMMING

Nested State Machines Made Simple

Mathematical **Models** of Dynamic Systems

Keeping Track of **Time**

Motor Rotation Control

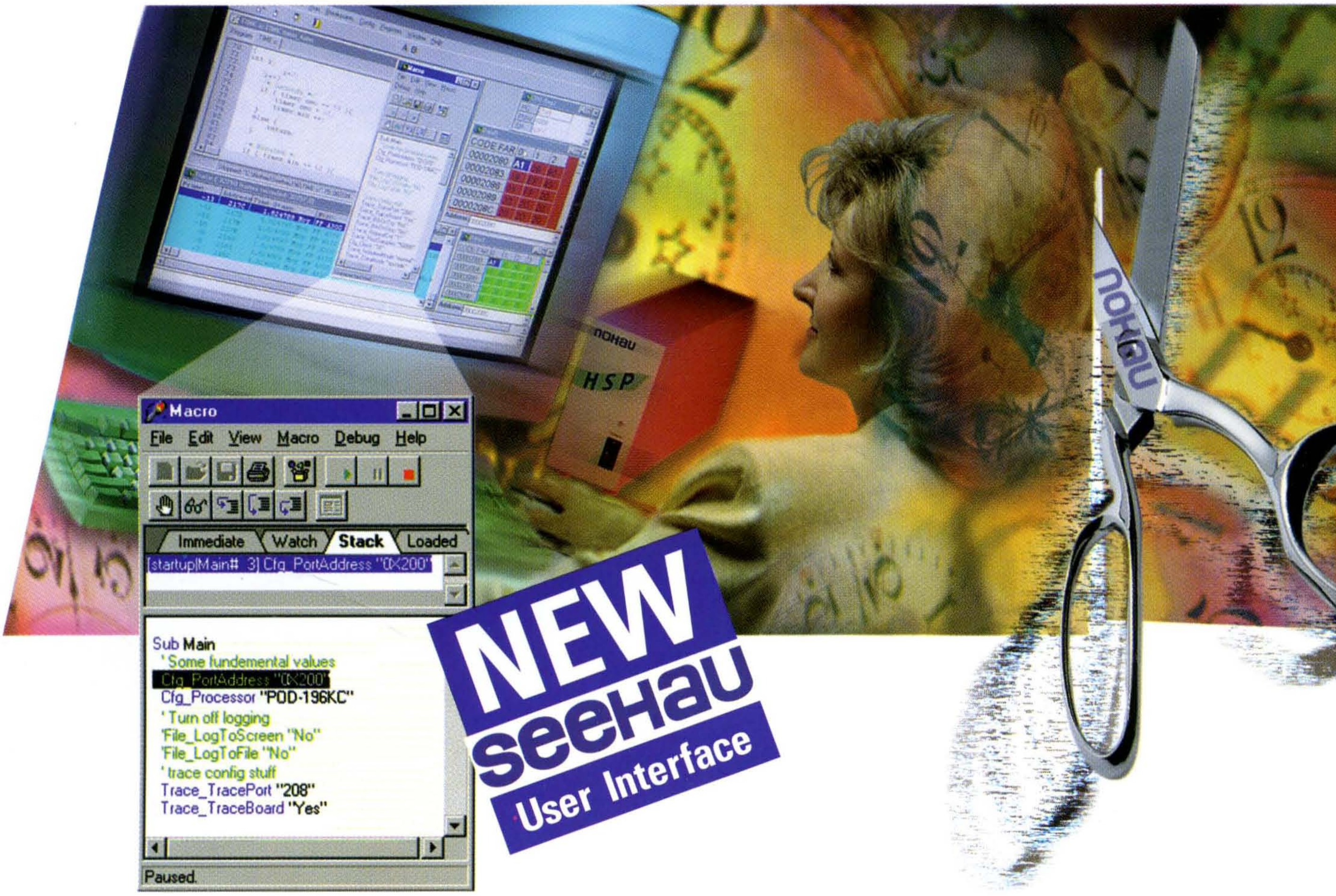
Internet Appliance Design:

Interrupt Handling Under Linux

Voice Over IP

Cut Development Time

Using In-Circuit Emulators and BDM's



```
[startup|Main#  3] Cfg_PortAddress "0X200"
```

```
Sub Main
  ' Some fundemental values
  Cfg_PortAddress "0X200"
  Cfg_Processor "POD-196KC"
  ' Turn off logging
  'File_LogToScreen "No"
  'File_LogToFile "No"
  ' trace config stuff
  Trace_TracePort "208"
  Trace_TraceBoard "Yes"
```

Macros

- Macro debug capability
- Project support
- Command line
- Shadow RAM support
- Single step in your macro
- GUI capable macros

Key Features

- Real-time emulation at maximum chip speeds
- High level support for popular C compilers
- Advanced tracing capabilities
- Configurable user interface with remote hookup capability
- High speed connection to PC through parallel port (LPTx) or plug-in ISA card

Nohau Supports These Microcontroller Families
8051 80C196 683xx 68HC11 P51XA ST10
MCS296 MCS251 68HC16 C166 68HC12 M16C

www.nohau.com

nohau

51 East Campbell Avenue, Campbell, CA 95008
Phone: 1-888-88NOHAU (1-888-886-6428)
Fax: 408-378-7869 E-mail: sales@nohau.com

Lift and Soar.
Embedded Development Tools from the New Wind River.

"Now, I have the tools to keep my project on time."

I remember when I budgeted for my project. I refused to spend money on debug tools. I didn't think I would need them. The money I saved would make me a hero.

Now, my project is one month, two months ... six months overdue. Nobody thinks I'm a hero. What would I give to get some of that time back?

Microtek debug tools can make time. They speed up development and testing, saving you precious time at the end of the project, where you need it most!

Microtek emulators offer the debug features you need to track down and correct software, hardware, and system integration issues. They can find the errors software debuggers cannot see.

First, emulators can debug before the operating system is functioning. If an unexpected issue is affecting boot up, an emulator can find it. They are also operational after a hard crash. This is significant, because software debuggers lose debug information. Microtek emulators keep track of the last 128 KB of bus cycles, allowing you to sift through and find the problem.

Finally, if there is a subtle issue during the integration of hardware, Microtek EA emulators are capable of providing a clock-cycle-by-clock-cycle trace that allows you to view each signal and determine whether it was in the correct state. Microtek's tools are clearly superior to software debug solutions.

URGENT!

If you are developing a Pentium class target using a reference design or off-the-shelf board, please take a moment to speak with our staff. They will insure your design is on track, and tool friendly if debug is needed in the future.

There has never been a project that couldn't use more time when it's critical ... like in final testing when everything has come together ... or in final debug, when everything is going down in flames.

Wouldn't it be great to deliver your next project on time? Next time, be sure to put a Microtek emulator in your project plan from the start.

It will help take the knot out of your stomach. And it will improve your company's bottom line!

Additional interfaces: CAD/UL® and Windriver Tornado II®

MICROTEK
IN-CIRCUIT EMULATORS

1 (800) 886-7333
Phone (503) 533-4463
Fax (503) 533-0956
Email – info@microtekintl.com

www.microtekintl.com

Financing Available

EmbeddedSystems
PROGRAMMING

EDITORIAL DIRECTOR
Lindsey Vereen, lvereen@cmp.com

EDITOR IN CHIEF
Michael Barr, mbarr@cmp.com

MANAGING EDITOR
Felisa Yang, fyang@cmp.com

SENIOR SPECIAL PROJECTS EDITOR
Tarita Whittingham, twhittingham@cmp.com

EDITORIAL ASSISTANT
Timothy Sullivan, tsullivan@cmp.com

CONSULTING TECHNICAL EDITORS
Jack G. Ganssle
Jerome L. Krasner, PhD

CONTRIBUTING EDITORS
Jack W. Crenshaw
Larry Mittag
Don Morgan
Dan Saks

VICE PRESIDENT/ELECTRONICS
Donna J. Esposito

EMBEDDED/DSP GROUP DIRECTOR
Mike Flynn, (415) 278-5251

PUBLISHER
Eric Berg, (415) 278-5220

NATIONAL SALES MANAGER
Stephen Corrick, (323) 512-4100

EASTERN REGIONAL SALES MANAGER
Damon Graff, (781) 839-1285

EASTERN SALES REPRESENTATIVE
Jared Grimm, (781) 839-1286

CALIFORNIA SALES MANAGER
Andres Diaz, (415) 278-5274

CALIFORNIA SALES ASSOCIATE
Molly Bruns, (415) 278-5298

WESTERN ACCOUNT EXECUTIVE
Sam Louis, (415) 278-5223

PRODUCTION COORDINATOR
Matthew Powell

CIRCULATION MANAGER
Jennifer Schuler

CIRCULATION ASSISTANT
Michael Falabella, mfalabel@cmp.com

CIRCULATION DIRECTOR
Susan Harper

SUBSCRIPTION CUSTOMER SERVICE
Toll free: (877) 676-9745
(847) 559-7307
Fax: (847) 291-4816
Omeda Communications
610 Academy Drive
Northbrook, IL 60062
esp@omeda.com

Back issues may be purchased on a prepaid basis through: Miller Freeman, 1601 West 23rd St., Suite 200, Lawrence, KS 66046; (800) 444-4881; (785) 841-1631

REPRINTS
Stella Valdez, (916) 983-6971

ONLINE PRODUCTION COORDINATOR
Billy Biondi, wbiondi@cmp.com

PRESIDENT/CEO, CMP MEDIA INC.
Gary Marshall

EXECUTIVE VICE PRESIDENTS
Regina Starr Ridley John Russell
Steve Weitzner Tony Uphoff

**SENIOR VICE PRESIDENT/
GLOBAL SALES & MARKETING**
Bill Howard

**SENIOR VICE PRESIDENT/
BUSINESS DEVELOPMENT**
Pam Watkins

PRESIDENT/ELECTRONICS
Steve Weitzner

Consumer Audit BPA

CMP

Visit our Web site at
www.embedded.com

Michael Barr

The More Things Change

The work of embedded software developers hasn't changed significantly in the 12 years that *Embedded Systems Programming* has been published. Sure, more of you than ever are using 16- and 32-bit processors in your designs, and the use of high-level languages has made substantial inroads against assembly. But the basic responsibilities and day-to-day hassles of the embedded systems programmer are much the same today as they were way back in 1988.

From the start, *ESP* has committed itself to providing practical, technical information that you can use to get your job done, and done well. Along the way, it has also tracked some of the trends and changes affecting our industry. Some, like programming in C, have caught on; others, like fuzzy logic, haven't fared as well.

This month the magazine is making a change of its own, as I take over the editorial reins from Lindsey Vereen. Lindsey has served as editor in chief for just over five years. During his tenure, *ESP*'s circulation increased by 50% and the size of an average issue approximately doubled. When we increase the circulation again later this year, we'll have 60,000 readers—embedded software developers all. I'd like to thank Lindsey for his hard work during those years and to congratulate him on a job well done.

Fortunately, Lindsey is moving up in our organization rather than out. In his new position as group editorial director, Lindsey will manage CMP's coverage of embedded systems across multiple magazines, conferences, and Web sites. So he'll still have his hands in the mixing bowl.

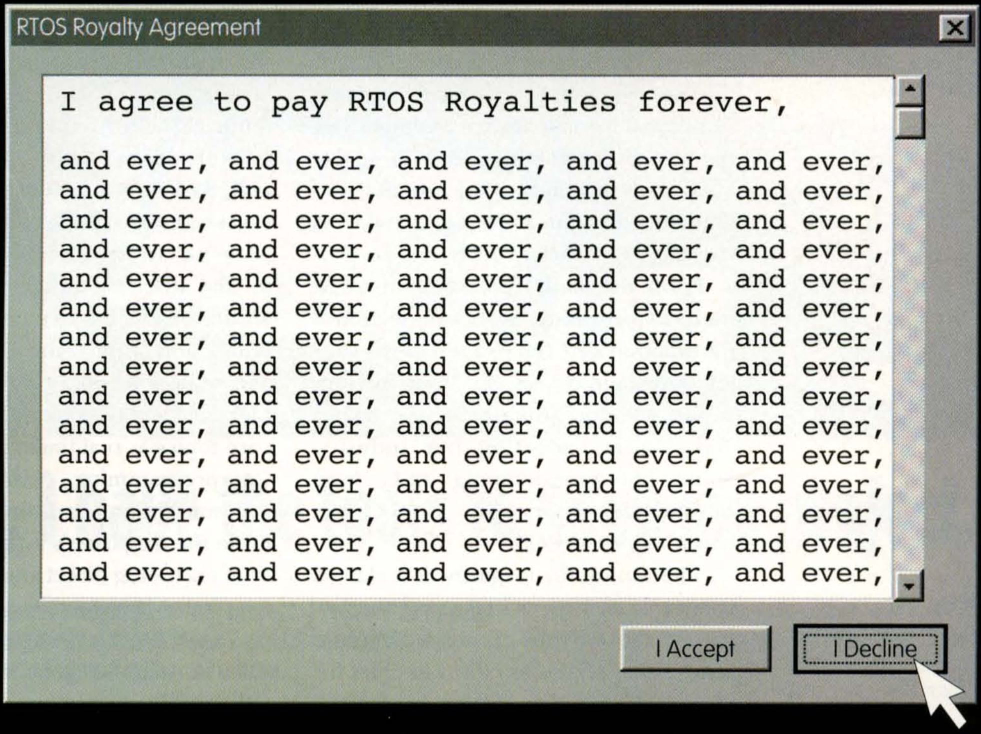
As an enthusiastic long-time reader of *ESP* and an embedded software developer who still likes to get his hands dirty, I know what this magazine is about and why you read it month after month. As technical editor, I've done my best to learn about the business of editing too, and to understand the magazine's editorial philosophy.

The long and short of all this is that I think *ESP* already has an editorial formula that serves the needs of its readers well. So I'm not looking to change our course in any significant way. I will continue my efforts to improve the quality of our articles—both from a technical and a readability standpoint—and to retain and attract the best columnists and writers in our industry. I will also seek to increase our coverage of such core topics as real-time concerns, C and C++ programming, device drivers, and software design. And, finally, I will continue the magazine's long-standing policy of excluding direct and indirect product pitches disguised as articles.

Please don't ever hesitate to tell me what you think—good or bad—about either *ESP* as a whole or a particular issue, article, or column. I personally read and respond to every e-mail I receive from readers and your comments never fall on deaf ears. As far as I'm concerned, this magazine exists for one reason and one reason only: to help you do your job better. So keep reading, and rest assured that the more things change, the more they'll stay the same.

Michael

mbarr@cmp.com



RTOS Royalty Agreement
I Accept
I Decline

Reader's Lament

A fresh copy of *Embedded Systems Programming*; perfect for lunchtime. First thing: flip to the back as usual. (I'll check out the rest of the issue later.)

Well, you know what happened.

Sad lunch.

It's hard to write about a writer, but I'd like to pass on some of my perceptions about P.J. Plauger's writing.

Some writers are controversial just for the sake of calling attention to themselves. Some writers are technically savvy, but leave you suffering from a high reader-effort to concepts-covered ratio. Some writers are personal and breezy at the expense of content. But there are a precious few writers who can show you things you might not have thought about, show you why they are important, let you know how they relate to the author's personal experience, and leave you with the feeling that you just had an important and stimulating conversation with a bright and friendly mentor.

P.J. Plauger is one of the rare writers to put it all together. *Embedded Systems Programming* and its readers have been fortunate indeed to have had the benefits of his columns all these years.

I'll really miss those lunchtime chats he and I used to have, even if I never did get a chance to meet him in person!

Chris Vickery
Queens College of CUNY

Test results: 'C'

While I, a grizzled embedded systems veteran, may not agree with the usefulness of every question in Nigel Jones' article, "A 'C' Test," (May 2000, p. 119), on the whole I found it both interesting and educational. I think that Question 11 (interrupts) is especially meaningful and that this relates to a key area for potentially creating unstable software. As an example of the article's educational value for me: though I have used C for about the last 15 years, I did not know that you could use the `U` modifier on a `#define`'d constant. Well, now I do, and I immediately began using it to improve the self-documenting quality of my code.

While I suspect that several others have already commented on the following "bugs" in "A 'C' Test," for what they are worth, here are a couple of things I observed.

Question 1: While the `#define` compiles just fine, it does not work if you actually try to use the resulting constant with any of the several Borland compilers I used to test this technique. (One of these days, I am going to drag out the old Franklin C Compiler for the Intel 8051 family I have and see how it acts.) As I recall, the error messages were the same in every case for the various Borland compilers. For example, using the BC++ 3.1 IDE, if you code:

```
#define SECS_PER_YR (60 * 60 * 24
* 365)UL
```

the previous line appears to be okay, but later:

```
unsigned long ulA = SECS_PER_YR;
```

creates a `Declaration syntax error` message:

```
unsigned long ulB;
...
int main():
ulB = SECS_PER_YR;
```

This causes a `"Statement missing"` error message. This can be fixed by moving the `UL` to immediately after either of the `60`s or the `24` in the `#define`. I suspect this is related to the fact that 60 * 60 is 3,600 which fits into a 16-bit unsigned integer, while 60 * 60 * 24 is 86,400, which does not. If you put the `UL` immediately after the `365`, then the compiler errors go away, but the result in both `ulA` and `ulB` is 7,615,360 instead of the desired 31,536,000. Interestingly, decimal 86,400 equals hexadecimal 15,180. If you mask this to 16 bits you get, of course, hexadecimal 5,180, which equals decimal 20,864. Finally then, decimal 20,864 times 365 just happens to equal 7,615,360.

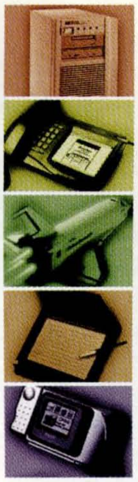
Hmmmmm.

Question 15: If you compile it as printed, you get a `"Misplaced else"` error. I suspect this was a typesetting error and that the author should not be blamed. However, I "corrected" the code by moving the `else` down between the two `puts` lines like this:

```
char cPtr;
if ( ( cPtr = (char *)malloc (0) )
== NULL )
  puts ("Got a null pointer");
 else
  puts ("Got a valid pointer");
```

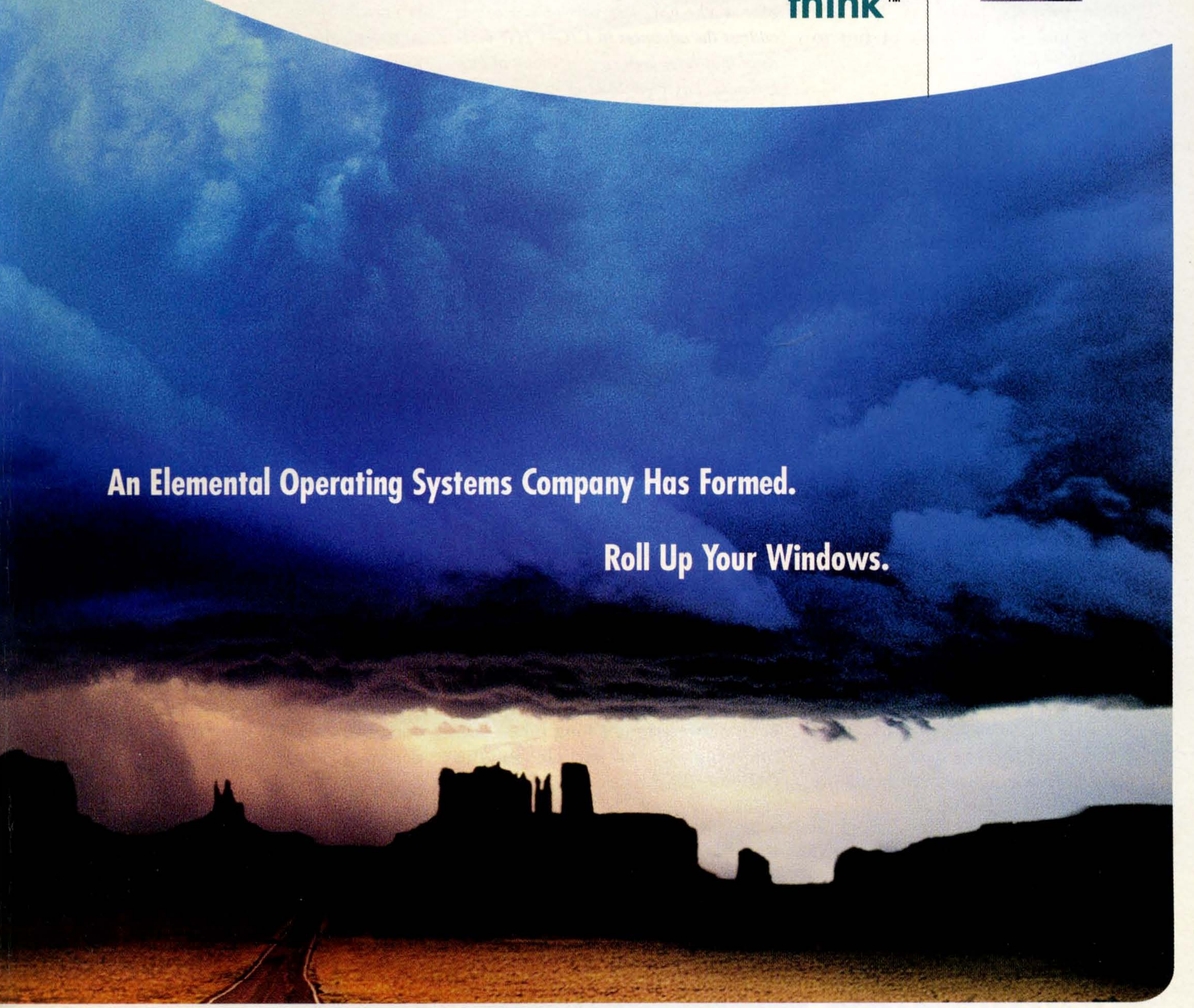
This snippet produces the message `"Got a null pointer"` using several different Borland compilers that I tried. As Mr. Jones said, what is probably more important here is the reasoning used by the applicant to decide which course the library routine should take. Personally, I think I much prefer the `null pointer` result over the other alternative,

There's a new force sweeping across the landscape, ushering in a new era of embedded capability. An era of operating systems and development tools that are balanced, strong and consistent, like nature itself. What's behind this? The merger of Wind River Systems and Integrated Systems, Inc., including DIAB SDS, Doctor Design and TakeFive Software. Separately, these companies were leading the way in software for smart devices. Together, they will offer unprecedented expertise, drawing from a vast pool of resources to craft targeted solutions for their clients. So behold the power, and put it to work for you. Call Wind River at 1-800-545-WIND today.



www.windriver.com

How smart things think™



as I believe you are less likely, in that case, to create unreliable software by storing into "`malloc`'ed" memory that you do not own. I am presuming here that a library routine that returns a non-NULL pointer from the `malloc()` call may well be pointing to a location in the heap that is "owned" by some other function; or worse yet, some other task. Even worse, suppose the "owner task" of the pointed-to location varies as the system runs. I suppose I have seen worse software problems, but this one would not be a lot of fun to debug!

Charlie Carothers
CCarothers@tidel.com

Co-verification methods

In his article "Co-Verification and System Abstraction Levels" (June 2000, p. 90), Bob Morasse correctly identifies co-design and co-verification as the central problems for engineers creating system-on-a-chip products, and the article gives a good overview of the concepts involved.

But I was surprised to find no mention of one of the latest and most effective co-verification methods: high-speed, clock-accurate C simulation of a complete system model, both hardware and software. By modeling the hardware RTL in standard programming languages like C or C++, this type of simulation runs 10 times faster than HDL. And by faithfully simulating complete hardware behavior at each clock, this method captures all the key details of hardware/software interaction, including interrupts, cache misses, pipeline stalls, and so on.

This kind of "RTL C" model is great for software developers, because it's easily integrated with standard IDEs and debuggers and is fast enough to support interactive debugging cycle-by-cycle. The RTL C method was pioneered by a company called CAE Plus in Austin, TX. CAE Plus offers several tools for creating C models for cores and ASICs, including a product called Afterburner that automatically converts RTL Verilog into clock-accurate C. CAE Plus also provides support for software development using the Green Hills tool chain.

Kerry Kimbrough
kkjpr@tidel.com

Author Bob Morasse replies:
The article was adapted from a paper I gave at the ESC last year and does not address the advances in C/C++ HW modeling that have been coming along of late. Intrinsix has been looking closely at the emerging C-based tools for high-level system modeling. We have had some experience with CynApps and System C and are also looking at C-Level. I'm interested in hearing some more about CAE Plus.

There seems to be a bit of a flushing out going on with respect to standardization of C constructs for HW description, as well as conversion from C to HDL. I would expect some great progress in this area in the next 12 to 24 months.

Medieval theology?

Alexander Wolfe's otherwise good article, "Alliances Drive Embedded Linux Towards Prime Time" (June 2000, p. 49) had some errors, particularly in its coverage of the real-time extensions to Linux. He writes, "[w]hether its kernel can be extended to deliver real-time deterministic performance is another question entirely." This question has been answered for a long time—see, for example, my article in the October 1997 issue of *ESP* entitled "Linux as an Embedded Operating System," (p. 96) which describes Real-Time Linux. Even then RTLinux, a project started by Victor Yodaiken and Michael Barabanov at New Mexico Tech, was a fully functioning and mature product. Continuous improvements, primarily in the areas of ports to other architectures, symmetric multiprocessing, and POSIX compatibility have been made since that time. Although it uses an unconventional programming model, RTLinux proves that one can use Linux for real-time applications.

Wolfe also mentions RTAI, another real-time extension that uses the same principles as RTLinux. But he implies that RTAI is produced by Zentropix and that it is not yet complete. In fact, RTAI is an open source project by Paolo Mantegazza and others at the Dipartmento di Ingegneria Aerospaziale Politecnico di Milano. Zentropix and others have contributed to the project, and Zentropix distributes it. Like RTLinux, RTAI is a complete, working system—both are used in real applications today.

Finally, Wolfe wanders into the "open source" vs. "free" software minefield, calling open source "a concept pioneered by the Cambridge, MA-based Free Software Foundation... ." As Richard Stallman never tires of reminding people, the FSF champions "free" software, which emphasizes users' rights. On the other hand, the term "open source" is used by those who believe that allowing users to have access to source code is a superior software development model. These people, of whom Eric Raymond is perhaps best known, downplay the ethical dimension and emphasize the practical advantages of making source code available. While to the outsider this all may sound like a hairsplitting medieval theological debate, a writer commits a *faux pas* like Wolfe's at his peril.

Jerry Epplin
jerry@stereotaxis.com

WHAT'S ON YOUR MIND?

Embedded Systems Programming welcomes feedback from readers. Please send any questions or comments to Editor in Chief Michael Barr at mbarr@cmp.com. We reserve the right to edit all letters for clarity and length.

Send Mail: Building a Gateway is that simple

To: NEO@matrix.com

Priority: Highest

Subject: Building a Gateway is that simple

Dear NEO,

You've heard all the TALK.. But the truth is..

There is ONLY ONE OSGi compliant embedded server software designed for Gateways that has "working" JINI Implementation and UPnP and WAP and DIRECTORY service for security and network management.

So ONLY ONE Question Remains.. Why haven't you called ProSyst?

If you are in the game of device automation then we have the key to the Gateway.
Its simple. Its all we do, and the ONLY ONE who does it RIGHT..NOW

Visit www.prosyst.com or call us 678-366-4607

Regards,
Keith

Sign up for our Early Mover Program and get free development tools!
See a demonstration at the Embedded Systems show San Jose Sept.25-27!

ProSyst USA, Inc.
555 North Point Center East
4th Floor, Suite 450
Alpharetta, GA 30022
http://www.prosyst.com

Keith Sciulli
Marketing Director
Phone 678 366 4607
Fax 678 366 4655
k.sciulli@prosyst.com

ProSyst®

www.prosyst.com

Antennas Predicted to Get Smarter

Big things are in store for antennas in the coming years. According to a report from Allied Business Intelligence, the rapid growth of wireless users worldwide will create demand for "smart" antennas, devices which compensate for transmission interference. This will allow developers to, in the words of ABI analyst Frank Vasquez, "squeeze more capacity out of existing networks," as well as giving them more flexibility in deploying new ones. It would appear that smart antenna manufacturers will be up to the task. They're boasting of capacity gains from 50% to 100% in digital networks. The range within these two figures depends on air interference.

Another good sign for antennas is, according to the report, the increasing integration of GPS applications into everyday life. The GPS antenna industry is already experiencing significant growth due to factors such as sales of in-vehicle navigation systems and to a lesser, but steadily growing, extent, cargo/fleet tracking applications.

Lynx Now LynuxWorks

Lynx Real-Time Systems has changed its name to LynuxWorks to reflect, according to an announcement, the company's "new focus on bringing the benefits of Linux to the embedded market."

"The embedded software world stands to benefit from the standard, open, multi-vendor solutions provided by Linux, and our full commitment to support it," said LinuxWorks chairman and CEO, Inder Singh.

Embedded conference heads north

This September's Embedded Systems Conference in San Jose marks the end of an era. To accommodate industry growth, the 2001 conference will now take place at San Francisco's Moscone Convention Center on April 9-13. The change affects subsequent conferences as well, with ESC Chicago being rescheduled for July 9-11, 2001 at Navy Pier, and ESC Boston to take place September 4-7, 2001 at the Hynes Convention Center.

Briefly Noted...

TASKING plans to integrate real-time operating system and embedded Internet capabilities with its tools portfolio. ✳ **I-Logix Inc.** announced the release of Rhapsody Modeler, a free edition of the Rhapsody development environment. ✳ The following companies have joined the Embedded Linux Consortium: **NETSilicon**, **Accelerated Technology**, **Applied Data Systems**, and **Reasoning**. ✳ **Tundra Semiconductor** announced a record 46% revenue increase from $27.8 million for the year ended April 30, 1999. ✳ **Analog Devices** has added C++ language support to its VisualDSP development environment. ✳ **Scenix** closed a $42.1 million equity investment whose participants included **Cisco Systems**, **Dell Computer**, **JatoTech Ventures**, and **MSD Capital** among others. ✳ **Green Hills Software** announced that it will offer enhanced debug support for **Agilent's** High-Speed E5900B emulation probe.✳ **Taiwan Semiconductor Manufacturing Corporation** (TSMC) has been chosen as the foundry for **Billions of Operations Per Second**'s (BOPS) Manta DSP chip as part of an agreement in which TSMC joined BOPS' Alliance Partnership program and BOPS joined TSMC's Design Service Alliance. ✳ **JEDI Technologies** has joined the **MIPS Technologies** Alliance program and will provide a Java acceleration solution for the MIPS32 and MIPS64 architectures. ✳ **Altera** announced a licensing agreement that enables its customers to access **ARM** core-based embedded processor tools for their programmable logic designs. ARM will supply Altera with the ARM9 Thumb technology-enabled embedded processor cores for use with the Altera APeX architecture. Altera has also licensed **MIPS Technologies**' embedded processor cores for system-on-a-programmable-chip designs. MIPS will supply Altera with a MIPS32 4K processor core. ✳ **Motorola** announced a number of new products that will supply PC manufacturers and the automotive industry with Bluetooth wireless technology. ✳ **Lineo** announced the creation of the Industrial Solutions Group to focus on real-time tools for the industrial control and military marketplaces.

NAMES IN THE NEWS

Centura Software Corp. has appointed **JOHN WISER** vice president of product management, **LEU VASQUEZ** senior product marketing manager for information appliances, **JUN TAKEMORI** president and general manager of Centura's embedded systems office, and **JOHN ARMENAKAS** managing director of the Asia Pacific office. ● Lineo has named **KIM D. CLARK** vice president of engineering and **ALLAN SMART** vice president of the professional services division. ● **JOSEPH NOONAN** has been appointed vice president of worldwide sales and services at Software Emancipation Technology. ● Caliber has made **DAN MCKAY** director of system architecture.

DEVICE SERVER™

TECHNOLOGY

THE MOST **POWERFUL** WAY TO

INTERNET CONNECTIVITY. IT'S NO LONGER A QUESTION OF "WHY". TODAY, YOU HAVE TO FIGURE OUT "HOW". HOW CAN YOU GAIN ACCESS TO VITAL INFORMATION THAT IS PRESENTLY TRAPPED IN UNCONNECTED DEVICES? HOW CAN YOU LEVERAGE THE INTERNET TO SOLVE YOUR BUSINESS PROBLEMS SUCCESSFULLY?

PUT THE INTERNET IN

THE ANSWER IS SIMPLE – PUT THE INTERNET IN YOUR DEVICES. GIVE DEVICES THE POWER AND FREEDOM TO SHARE INFORMATION DIRECTLY OVER THE INTERNET WITHOUT THE NEED FOR BULKY, EXPENSIVE PCs. PLUS, YOU'LL ALSO HAVE THE ABILITY TO ACCESS CRITICAL INFORMATION, MANAGE DEVICES IN REAL-TIME AND CREATE INTERNET-ENABLED APPLICATIONS THAT YOU NEVER IMAGINED POSSIBLE. THE POST PC ERA HAS ARRIVED. ARE YOU READY?

DEVICE SERVER TECHNOLOGY BENEFITS

- Embed the Internet in any device
- Give devices the power and freedom to share critical information
- Fast time to market — weeks instead of years
- Save thousands of dollars in deployment and connectivity costs
- Create Internet-enabled applications that were never before possible
- Gain a new competitive edge
- Develop new business opportunities and generate new revenue streams
- Access, manage and control devices from anywhere in real-time

LEVERAGE THE INTERNET. ←

WITH DEVICE SERVER™ TECHNOLOGY, YOU CAN INTERNET-ENABLE ANY DEVICE.

past

Bring the benefits of Internet connectivity to your installed devices, equipment and software applications with Device Servers, giving them new life and immediate access to the network. Device Servers transparently handle all communications, letting you use existing applications without any modification.

present

Device Servers can be bundled with non-networked devices you are currently deploying for immediate Internet connectivity. Device manufacturers can also embed board-level Device Servers into current products to Internet-enable them in a flash, without spending months with complicated hardware and software integration.

future

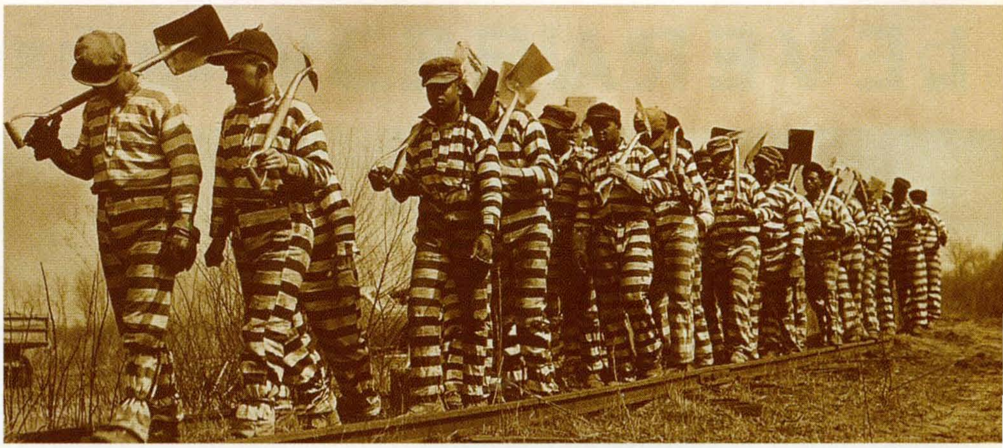
Device Server Technology is the perfect enabling platform for manufacturers facing increasing demand for the Internet in their future products. By embedding board-level Device Servers or licensing Device Server Technology, OEMs can focus on core competencies and bring amazing new products to market faster than ever before.

15353 Barranca Parkway
Irvine, California 92618 USA
949-453-3990
sales@lantronix.com

LANTRONIX®

www.lantronix.com
Europe: +31-76-565-8176
Asia: +65-447-4222

Any other way of developing software is MANUAL LABOR.



Why slave over writing, validating, debugging and documenting your code...when you can actually enjoy designing it?

We feel your pain. That's why we created Rhapsody®. It's the only visual UML compliant real-time application software development environment that simultaneously integrates and automates analysis, design, implementation and test.

Our exclusive Model-Code Associativity guarantees that your model and code are always in sync. Change your model, your code changes. Or–you won't believe this–change your code, your model changes. And you won't even break a sweat.

Frankly, Rhapsody is amazing. It automatically generates readable, deployable, production-quality C, C++ and Java®. Not just code frames, but ALL the code. You can say goodbye to the tedium of manually developing makefiles, state machines, communication infrastructures and the like. You can also move from one RTOS to another with the push of a button. Yes!

With Rhapsody's unique design-level debugging you can visualize, easily detect and instantly correct logic and programming errors. In fact, you can actually test your application as you build it. And, wonder of wonders, you can import and reuse your legacy code. How's that for flexible?

Rhapsody users are already reducing their development cycle by at least 30%. C'mon, what are you waiting for?

Rhapsody will change the way you work. Forever.

Visit us at www.ilogix.com to download a free copy of Rhapsody.

Rhapsody®

I-Logix®

Don Morgan

Motor Rotation Control

Increasingly, motor controls are passed over in favor of DSP control. Here's a primer on the behaviors that can be controlled.

Motors, of all sizes and shapes, are so common that how they are controlled is a subject often taken for granted. Nevertheless, the principles involved in causing a motor to turn and controlling its motion are by no means trivial. This is complicated by the existence of so many different kinds of motors. Not only do these motors sometimes involve different principles of operation, but may possess different physical implementations as well, some quite exotic.

This month, we will begin a description of the Park and Clarke algorithms common in both the analysis and control of DSP-based motion controllers. Because attempting to launch even an elementary description of these algorithms is impossible without knowing some basics about motors and their control, we will have to spend some time studying concepts and mechanisms that some may find unfamiliar.

The DSP is bringing the power of mathematics to a great number of machine tool applications, making it quite popular in this area. For that reason, it is rapidly replacing the older analog controls, providing greater freedom and precision. Today, high-end motion control involves signal processing, in both the time and frequency domains.

The field of motion control is a wide one, and there is not enough room here to describe all of its myriad implementations. So I will start with a simple application involving single- and three-phase permanent magnet motors and, from there, form the basis of operation for an induction motor.

The terms and concepts involved in motion control can be a bit daunting if you are new to the area, so we will take it a step at a time. We will also keep it simple. If you are interested in more detailed discussions, please refer to the Web links at the end of the column.

Magnetic forces

I expect that everyone, at one time or another, has played with a pair of magnets. Most are aware that a magnet has two poles, one called north and the other called south. They have these names because when magnets were used in the earliest compasses, the poles were the parts of the magnet that wanted to align themselves with the poles of the earth. No matter how small the pieces into which we may break a magnet, each one will have a north and south pole.

A magnetic pole creates a field in the space around it that exerts a force on other magnetic materials. This field is usually conceived in the form of lines of induction, which indicate the direction of the field. A popular method of viewing these lines of induction is to place a magnet beneath a piece of paper and sprinkle the surface of the paper with iron filings. Another name for these lines of force is *flux*.

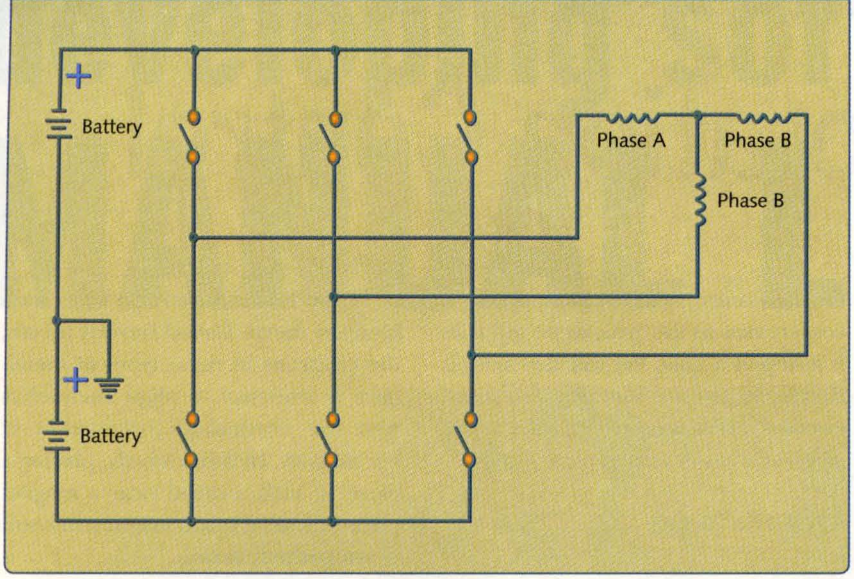
Magnets have an affinity for certain kinds of metal. Called ferrous metals, the electrons in these types of metals have a tendency to align themselves with the north-south orientation of the magnet. In other words, placing a piece of such a metal near a magnet turns the metal into a magnet, in a sort of sympathetic action.

Poles of opposite polarities attract one another. Placing the magnets on a tabletop, you can drag one around by bringing the second one close enough to establish a magnetic link. You can also push the other magnet around by rotating the pair so that similar poles are oriented toward one another.

It's not hard to see that if you fix one magnet on a pivot and bring another close enough to establish a link, you can cause the magnet on the pivot to rotate as you move the other one. This is a basic motor action.

The *rotor* (the magnet on the pivot) may be a magnet or an electromagnet. An electromagnet is a device in which an electric current passes through a wire coil wrapped around a soft iron (ferrous) core to produce a magnetic field. The strength of the magnetic field depends on the number of turns made by the coil of wire, the amount of current passing through the wire, and the magnetic permeability of the core.

FIGURE 1 Three-phase bridge



Electromagnets lose their magnetism when the current is discontinued.

As I have noted, we can cause the rotor to turn by leading its motion with another magnet. To create a motor we replace the other magnet with an electromagnet, which we call a *stator.* The stator forms a ring, or shell, around the rotor and is wrapped with wire to form the electromagnet. The trick is to control the flow of current in the stator so that the flux rotates through it in a circular fashion, taking the rotor with it.

In a single-phase permanent magnet motor, the iron core wrapped with wire forms a shell around a moving rotor. The stator is an electromagnet, and the rotor is a permanent magnet. Current flows in one direction in the stator, forcing the rotor to rotate as it tries to align with the electromagnetic flux. At a certain point in the rotation, the field is reversed and the rotor continues to turn, in an attempt to realign itself with the new field orientation. The switching that causes the current to change in the stator is called *commutation.* Commutation is commonly done with carbon contacts called brushes, or with semiconductors on brushless motors.

There are two ways of conceiving rotational speed. The most immediate is to think of it as a measure of how fast the shaft of the motor can complete one entire revolution, or 360 degrees. The other, more important concept—at least as far as motion control is concerned—is the electrical speed. It may already be apparent from the discussion that a complete electrical revolution occurs when the poles return to their original position relative to the stator. In a single-phase motor with a single pole-pair, this is the entire revolution of the rotor, but on motors with multiple pole pairs, a complete revolution is actually 360 degrees divided by the number of pole-pairs.

Controlling the wave of flux

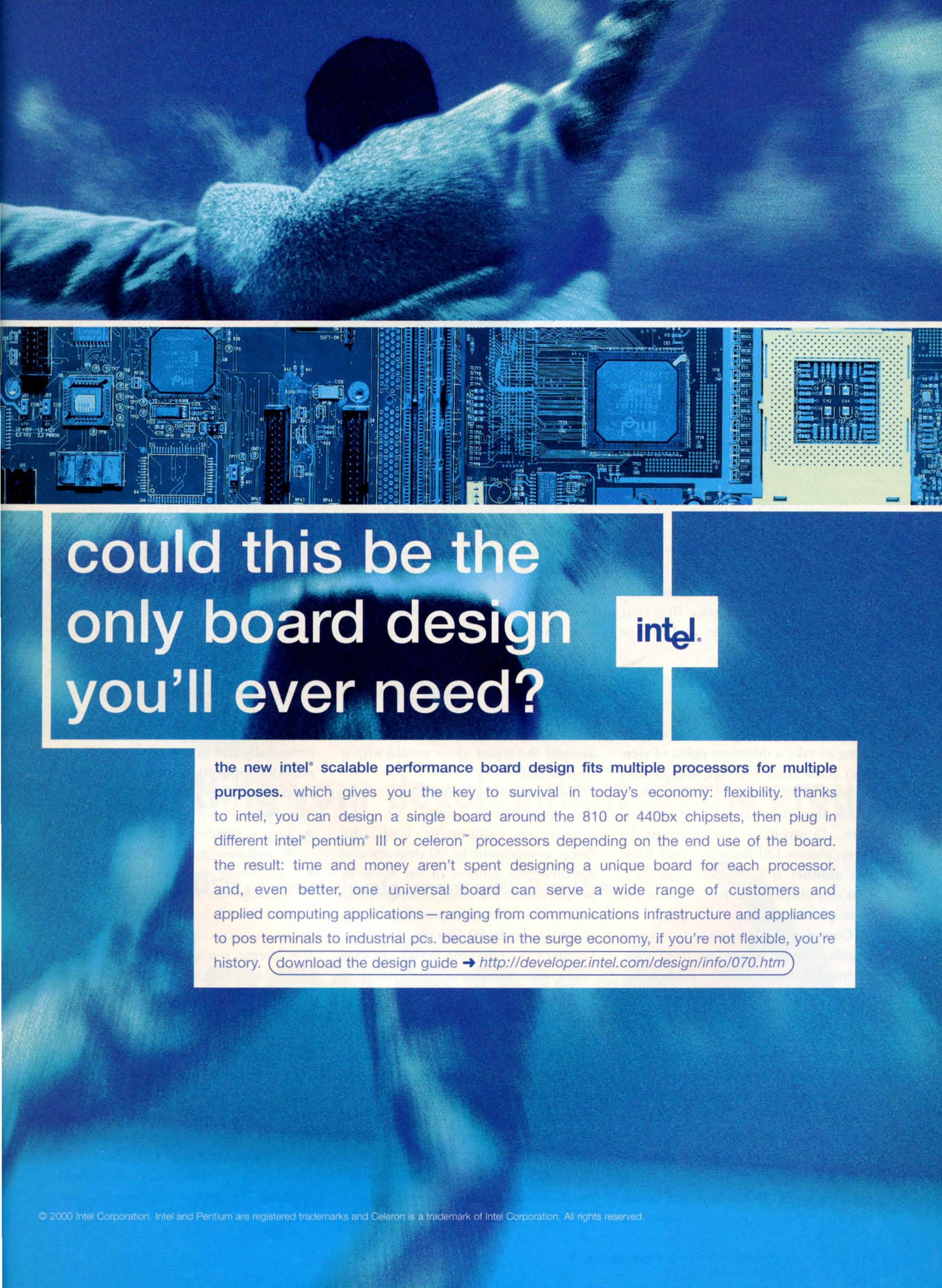
A three-phase motor is much like a single-phase motor, except that instead of a single winding there are three. The windings are connected most commonly—though not exclusively—in the center. As you will see, current in such a system can be caused to flow in as many as eight unique patterns through the windings. In this way, it is possible to gain fine control over the motion of the rotor and to create high torques.

The usual implementation employs a three-phase bridge (also called a power bridge) using transistors instead of mechanical switches. For a more concrete picture, refer to Figure 1. In the figure, we use switches instead of transistors. As you can see, the three-phase bridge is a parallel combination of three *half-bridges* in series with a DC voltage source. Each half-bridge is a series connection of two switches, one above the other. Each of the three windings is connected to the common point of a half-bridge.

In this configuration, a top and bottom switch are never closed simultaneously. This would cause a short circuit with all the current flowing back to the battery instead of to the motor and probably blowing fuses, or worse. Instead, when a top switch is "on"—current is flowing from the positive side of the voltage source into the winding—the bottom switch is forced off and when the top switch is off, the bottom switch is made to be on. The "on" time controls the amount of current that actually flows through the windings. This time is controlled so that only the desired amount of power is delivered to the motor. When the off-time equals the on-time, no current is flowing in the windings of the motor.

The technique used to drive a motor in this configuration is called pulse width modulation (PWM). A PWM signal is a binary signal—it has two states—in which the on-time (the time during which the signal is "on") is used to control a load. To get an idea of how this works, imagine a null state. For PWM, this would be a 50% duty cycle—that is, half the time the signal is one and half the time it is zero. If the on-time is greater than the off-time, the signal is positive and if the off-time is greater than the on-time, the signal is negative.

In the three-phase power bridge in Figure 1, we call the time the upper


intel.

The direct and quadrature components control the behavior of the rotor. On an induction motor, the rotor is also an electromagnet and must be magnetized before it can move or develop torque. But if a permanent magnet motor has magnets on the rotor, of what value is the direct component? Not much.

switch in any phase is closed the "on time." In the same manner, we call the time the lower switch is closed the "off time." In the case of a three-phase motor, each of the phases is controlled by a PWM signal that is 120 degrees out of phase with the other two.

The on-time and off-time are functions of the power you deliver to the motor and the desired position of the flux in the motor. The switching is also sequenced so that the current will flow in the motor windings relative to the desired direction of rotation. By sequencing the switching through the combinations, you can create a wave of magnetic flux that rotates around the stator and carries the rotor with it.

Components of motion

In the next few paragraphs, we are going to take a different point of view than most texts on this subject and describe the operation of the motor first in terms of a static frame of reference. Controlling the currents in three phases 120 degrees apart from one another is a complex operation that can obscure the relatively simple principles involved.

Consider the components of the angular motion of a motor from a static frame whose reference point is the stator flux. In the simplest terms, controlling the rate at which the flux rotates through the stator also controls the angular motion of the rotor or the speed with which it turns.

The goal is to create a sinusoidal current in the coils and generate a rotating field. We can do this with a DC source because (as in Figure 1) the windings are inductive. Modulating the duty cycle of the on-times of the switches (top and bottom) allows the current to build and decay in such a way as to form a sinusoidal current. And, with three phases, we can produce eight combinations, which allows finer control over the flux magnitude and position than with a single-phase motor.

There are two components to this motion of which we need to be aware: the direct and quadrature components. Let's suppose we have a permanent magnet motor (though an induction motor, which we will look at shortly, is only a little different). If we supply a current to the stator, the rotor will turn so that the two fields (stator and rotor) align. The direct component is aligned with the stator flux—it is the component that describes the amount of stator flux aligned with rotor flux. Using this, we can accomplish simple motion by "stepping" the flux around the stator as we would with a step motor.

To obtain torque, we make use of the quadrature component. The quadrature component leads the direct component by 90 degrees and advances the motion of the rotor in proportion to the value we assign it. As a note, if we allow the quadrature component to lag by 90 degrees, the motor becomes a generator supplying current rather than using it. A situation such as this can occur during rapid decelerations, so it's a good idea to place a voltage-sensing circuit and resistor (called a regeneration resistor) across the DC bus. When the voltage on the bus goes high enough to damage the circuit, the resistor will limit that voltage to a reasonable value.

The direct and quadrature components control the behavior of the rotor in a motor. But if a permanent magnet motor has magnets on the rotor, of what value is the direct component? Not much.

The major difference between a permanent magnet motor and an induction motor is the rotor. On an induction motor, the rotor is also an electromagnet and must be magnetized before it can move or develop torque. Here the direct value is used to establish the rotor flux.

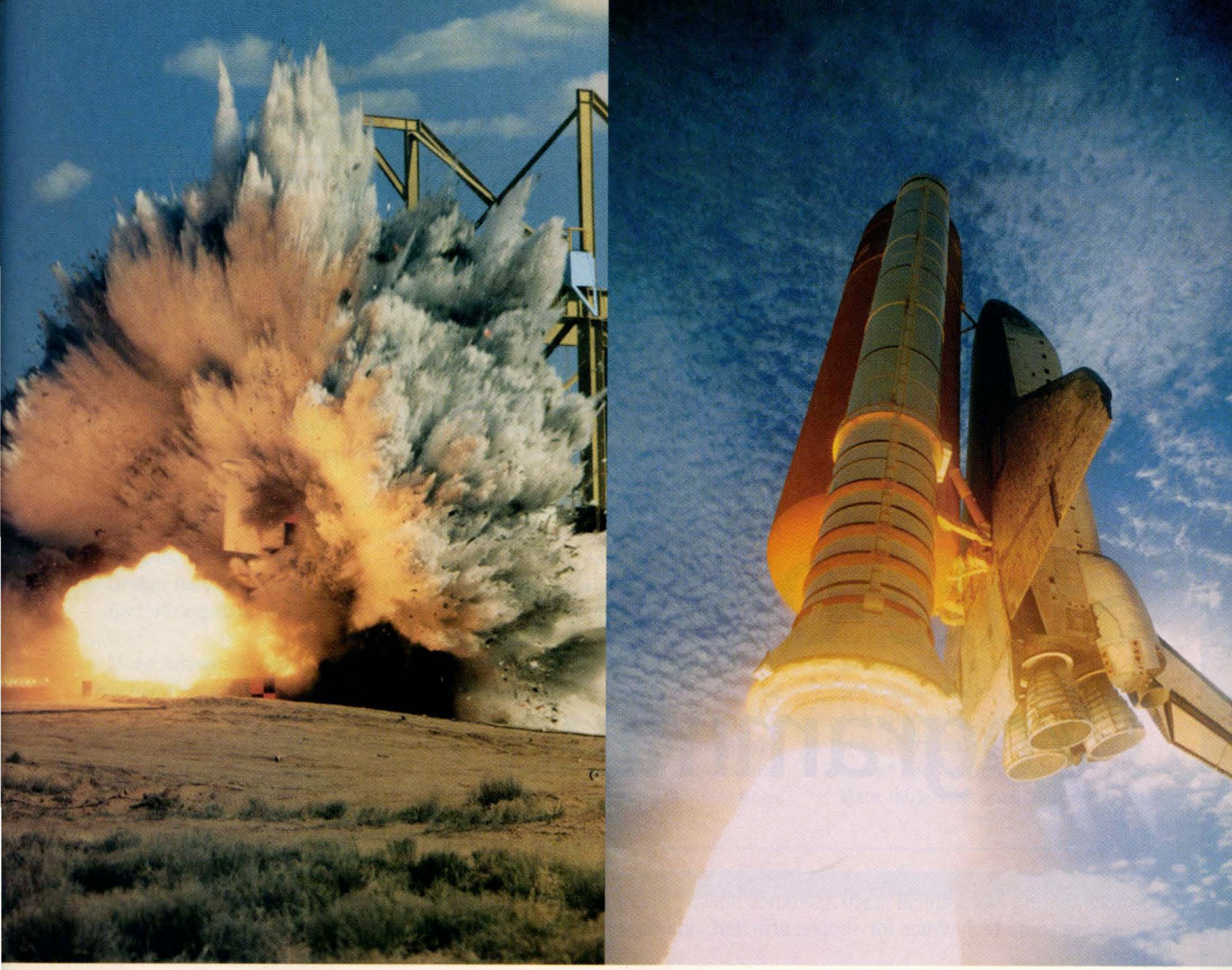
Whatever the type of motor, however, these two components exist, forming a frame that rotates with the flux around the stator. On a permanent magnet motor, the direct component may be set to zero, while the quadrature component is used to generate torque. On an induction motor, the direct component becomes much more powerful in terms of controlling torque and speed.

Next month we will see how these two values translate back into the real world and work with the PI loop and generate PWM for the motor.

The following two links—one for Texas Instruments and one for Analog Devices—deal with this subject more technically and in greater depth. Both provide application notes and code for particular processors:

- *www.ti.com/sc/docs/apps/digital/ac_induction_motors.html#App_Notes*
- *www.analog.com/industry/motor_control/appcode/admc401/pwm401_1.html* **esp**

Don Morgan is a senior engineer at Ultra Stereo Labs and a consultant with 25 years of experience in signal processing, embedded systems, hardware, and software. Morgan recently completed a book about numerical methods, featuring multi-rate signal processing and wavelets, called Numerical Methods for DSP Systems in C. *He is also the author of Practical DSP* Modeling, Techniques, and Programming in C, *published by John Wiley & Sons, and* Numerical Methods for Embedded Systems *from M&T.*



MIRO SAMEK AND PAUL MONTGOMERY

State-Oriented Programming

Implementing hierarchical state machines doesn't mean you have to use code-synthesizing tools. Here are some techniques for simple, efficient, and direct mapping of state machines to C or C++.

The Unified Modeling Language (UML) provides a number of conceptual and graphical views for capturing and conveying designs. Among these views, hierarchical state machines (HSMs), based on Harel statecharts, are of key importance and provide the foundation for automatic code generation from object models.[1] Unfortunately, this creates the impression that the methodology is only accessible through use of code synthesizing tools. This is similar to the common belief that object-oriented programming (OOP) is only possible with object-oriented (OO) languages. However, the key OO concepts of encapsulation, inheritance, and polymorphism may be implemented as design patterns[2] in a non-OO language such as C. Similarly, hierarchical state machines can be viewed as another such fundamental pattern.

From a more abstract perspective, one may view hierarchical state machines as a meta-pattern, in that various structured uses become design patterns (behavioral patterns[3]) in their own right. This is analogous to OO design patterns[7] built on meta-patterns of inheritance and polymorphism. Following this analogy to OOP, we propose the term *state-oriented programming* (SOP) to describe a programming style based on HSMs.

The primary goal of this article is to present a simple and efficient implementation of the HSM design pattern. By providing easy-to-use C and C++ recipes for generating HSMs, we hope to make the major benefits of the technology more accessible to the software community. The proposed implementation techniques are valuable in that they raise the level of abstraction and allow for straightforward mapping of UML statecharts to compact and efficient code in C or C++. We have used the technique extensively in deeply embedded, hard real-time RF receiver applications where both high speed and small memory footprint were crucial.

In an effort to maximize efficiency and minimize implementation complexity, many of the more advanced features of UML statecharts have been omitted. The implemented features form a proper subset of UML statecharts and include:

State-oriented programming raises the level of abstraction and allows for straightforward mapping of UML statecharts to compact and efficient code in C or C++.

- Nested states with proper handling of group transitions and group reactions
- Guaranteed execution of entry/exit actions upon entering/exiting states
- Straightforward implementation of conditional event responses (guards)
- Design that enables inheriting and specializing state models

More advanced features of UML statecharts (such as history mechanisms and orthogonal regions) can be added as behavioral patterns built on top of the implementation presented here.

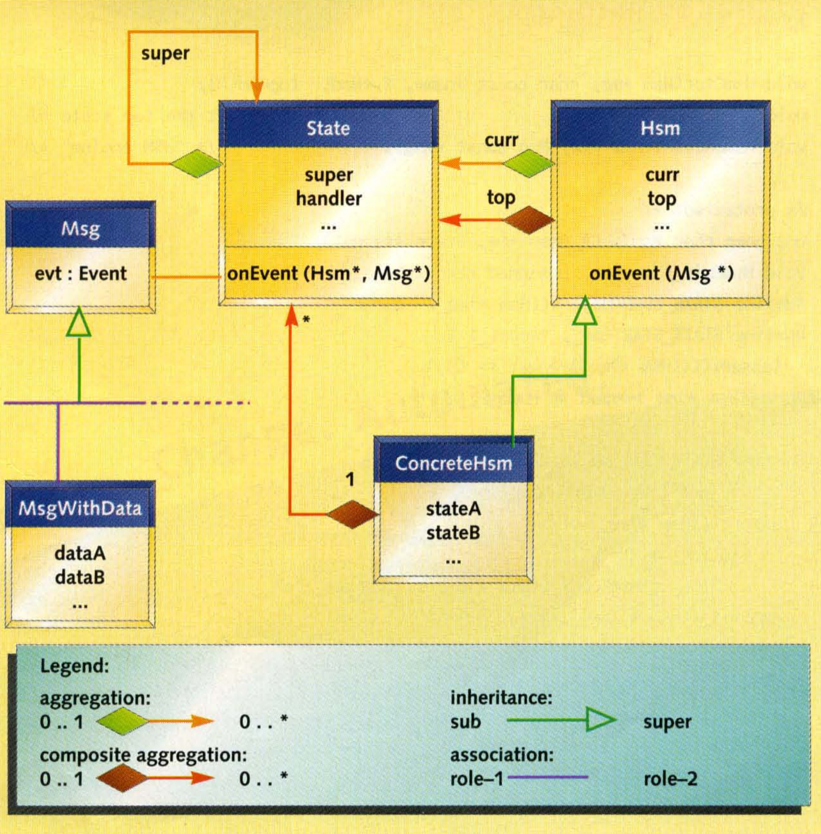
We begin with a brief summary of approaches that have been documented in the relevant literature or implemented in commercial products. We then describe our implementation using UML class diagrams and provide a complete implementation in both C and C++. Taking the example of a simple digital watch, we demonstrate how to map a UML state diagram to code and how to use most features of the HSM implementation in a concrete fashion. We briefly discuss how the HSM pattern, when combined with RTOS facilities, can be used to build a powerful real-time framework. We conclude by drawing a comparison between SOP and OOP and propose some modifications and extensions to UML statecharts. We assume that the reader is familiar with basic concepts of state machines and UML notation.[3,11]

Standard approaches

Typical implementations of state machines in C/C++ include:

- Doubly nested `switch` statements with a scalar "state variable" used as the discriminator in the first level of the switch and event-type in the second.[4] This is, perhaps, the most common technique and works well for classical "flat" state machines and is widely employed by automatic code synthesizing tools.[5] Manual coding of entry/exit actions and nested states is, however, cumbersome, mainly because code pertaining to one state becomes distributed and repeated in many places, making it difficult to modify and maintain when the topology of state machine changes
- Action-state tables containing typically sparse arrays of actions and transitions for each state.[4] Actions (including entry/exit, state reactions, and actions associated with transitions) are most commonly represented as pointers to functions. Representing state hierarchy in a flat action-state table is cumbersome. Also, this approach requires a large (and consequently wasteful) action/state array and many fine-granularity functions representing actions
- Generic "state machine interpreters" driven by typically complex data structures that represent the hierarchy of states together with entry/exit actions and transitions.[6] This is a generalized action-state table approach that attempts

FIGURE 1 Structure of HSM design pattern



RUPERT ADLEY

LISTING 1a hsm.h—C header file

```
typedef int Event;
typedef struct {
  Event evt;
} Msg;

typedef struct Hsm Hsm;
typedef Msg const *(*EvtHndlr)(Hsm*, Msg const*);

typedef struct State State;
struct State {
  State *super;                                   /* pointer to superstate */
  EvtHndlr hndlr;                              /* state's handler function */
  char const *name;
};

void StateCtor(State *me, char const *name,
               State *super, EvtHndlr hndlr);
#define StateOnEvent(me_, ctx_, msg_)\
  (*(me_)->hndlr)((ctx_), (msg_))

struct Hsm {                            /* Hierarchical State Machine base class */
  char const *name;                                 /* pointer to static name */
  State *curr;                                             /* current state */
  State *next;                       /* next state (non 0 if transition taken) */
  State top;                                          /* top-most state object */
};

void HsmCtor(Hsm *me, char const *name, EvtHndlr topHndlr);
void HsmOnStart(Hsm *me);                      /* enter and start the top state */
void HsmOnEvent(Hsm *me, Msg const *msg);                       /* "HSM engine" */

/* protected: */
unsigned char HsmToLCA_(Hsm *me, State *target);
void HsmExit_(Hsm *me, unsigned char toLca);
#define STATE_CURR(me_) (((Hsm *)me_)->curr)
#define STATE_START(me_, target_) \
  (assert((((Hsm *)me_)->next == 0),\
    ((Hsm *)me_)->next = (target_))

#define STATE_TRAN(me_, target_) if (1) { \
  static unsigned char toLca_ = 0; \
  assert(((Hsm *)me_)->next == 0);\
  if (toLca_ == 0) \
      toLca_ = HsmToLCA_((Hsm *)(me_), (target_));\
  HsmExit_((Hsm *)(me_), toLca_);\
  ((Hsm *)(me_))->next = (target_);\
} else ((void)0)

#define START_EVT ((Event)(-1))
#define ENTRY_EVT ((Event)(-2))
#define EXIT_EVT  ((Event)(-3))
```

to represent HSMs with a more efficient data structure than an array. Techniques in this category require each action to be coded as a separate function. They perform relatively poorly if the state machine is complex

- Object-oriented "State" design pattern based on delegation and polymorphism.[7,3] States are represented as subclasses implementing a common interface (each method in this interface corresponds to an event). A context class delegates all events for processing to the current state object. State transitions are accomplished by changing the current state object (typically re-assigning one pointer). This pattern is elegant and relatively efficient but is not hierarchical. Accessing context attributes from state methods is indirect (cannot use an implicit `this` pointer) and breaks encapsulation. The addition of new states requires subclassing and the addition of new events requires adding new methods to the common interface

Implementation

Our implementation of the HSM pattern is, to some degree, a combination of the techniques itemized above. The structure of the pattern is shown in Figure 1. This structure is greatly simplified relative to the standard full-featured UML design.[8]

States are represented as instances of the `State` class, but unlike the "State" pattern as described by Gamma et al.[7], the `State` class is not intended for subclassing but rather for inclusion as is. Accordingly, our approach requires state machines to be constructed by composition rather than by inheritance. The most important attributes of `State` class are the event handler (to describe behavior specific to the state) and a pointer to superstate (to define nesting of the state).

Messages are represented as instances of `Msg` class or its subclasses.

LISTING 1b hsm.h—C++ header file

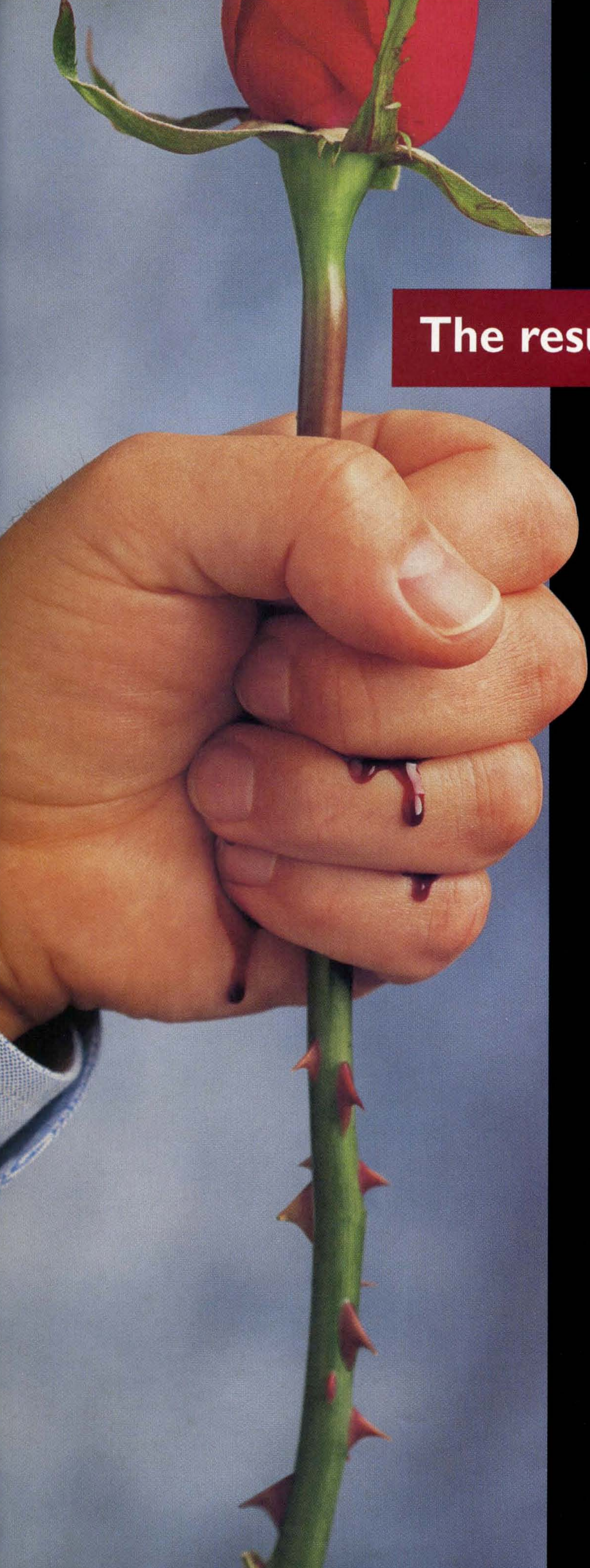
```
typedef int Event;
struct Msg {
  Event evt;
};

class Hsm;                                        /* forward declaration */
typedef Msg const *(Hsm::*EvtHndlr)(Msg const *);

class State {
  State *super;                                  /* pointer to superstate */
  EvtHndlr hndlr;                            /* state's handler function */
  char const *name;
public:
  State(char const *name, State *super, EvtHndlr hndlr);
private:
  Msg const *onEvent(Hsm *ctx, Msg const *msg) {
    return (ctx->*hndlr)(msg);
  }
  friend class Hsm;
};

class Hsm {                         /* Hierarchical State Machine base class */
  char const *name;                                /* pointer to static name */
  State *curr;                                            /* current state */
protected:
  State *next;                      /* next state (non 0 if transition taken) */
  State top;                                         /* top-most state object */
public:
  Hsm(char const name, EvtHndlr topHndlr);                           /* Ctor */
  void onStart();                               /* enter and start the top state */
  void onEvent(Msg const *msg);                       /* "state machine engine" */
protected:
  unsigned char toLCA_(State *target);
  void exit_(unsigned char toLca);
  State *STATE_CURR() { return curr; }
  void STATE_START(State *target) {
    assert(next == 0);
    next = target;
  }
# define STATE_TRAN(target_) if (1) {\
     static unsigned char toLca_ = 0;\
     assert(next == 0);\
     if (toLca_ == 0)\
         toLca_ = toLCA_(target_);\
     exit_(toLca_);\
     next = (target_);\
  } else ((void)0)
};

#define START_EVT ((Event)(-1))
#define ENTRY_EVT ((Event)(-2))
#define EXIT_EVT  ((Event)(-3))
```

All messages carry event-type (attribute **evt** inherited from **Msg**) and possibly arbitrary data (added by subclassing).

Events are handled uniformly by *event handlers*, which are member functions of **Hsm** class (**typedef EvtHndlr**). As shown in Figure 1, a state machine consists of at least one state—the **top**-level state inherited from **Hsm**. Concrete state machines are built by inheriting from **Hsm** class, adding an arbitrary number of states (plus other attributes), and defining event handlers.

Because event handlers are methods of **Hsm** or its subclasses, they have direct access to attributes via the implicit **this** pointer (in C++) or the explicit **me** pointer (in C). Within event handlers, only one level of dispatching (based on event-type) is necessary. Typically this is achieved using a single-level **switch** statement. Event handlers communicate with the state machine engine (see Listing 2) through a return value of type **Msg***. The semantic is simple: if an event is processed, the event handler returns 0 (NULL pointer); otherwise it returns ("throws") the message for further processing by higher-level states. To be compliant with UML statecharts, the returned message is the same as the received message, although return of a different message type can be considered. As we discuss later, returning the message provides a mechanism similar to "throwing" exceptions.

Entry/exit actions and default transitions are also implemented inside the event handler in response to the pre-defined events **ENTRY_EVT**, **EXIT_EVT**, and **START_EVT**. The state machine engine generates and dispatches these events to appropriate handlers upon state transitions. An alternative approach would be to represent entry/exit and start actions as separate methods, but this would require specification and maintenance of three additional (fine granularity) methods and three additional function pointers in each state.



LISTING 2a hsm.c—C implementation

```
static Msg const startMsg = { START_EVT };
static Msg const entryMsg = { ENTRY_EVT };
static Msg const exitMsg  = { EXIT_EVT };
#define MAX_STATE_NESTING 8
                                                          /* Hsm ctor */
void HsmCtor(Hsm *me, char const *name, EvtHndlr topHndlr) {
  StateCtor(&me->top, "top", 0, topHndlr);
  me->name = name;
}
                                       /* enter and start the top state */
void HsmOnStart(Hsm *me) {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  me->curr = &me->top;
  me->next = 0;
  StateOnEvent(me->curr, me, &entryMsg);
  while (StateOnEvent(me->curr, me, &startMsg), me->next) {
    for (s = me->next, trace = entryPath, *trace = 0;
         s != me->curr; s = s->super)
      *(++trace) = s;                             /* trace path to target */
    while (s = *trace )                       /* retrace entry from source */
      StateOnEvent(s, me, &entryMsg);
    me->curr = me->next;
    me->next = 0;
  }
}
                                             /* state machine "engine" */
void HsmOnEvent(Hsm *me, Msg const *msg) {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  for (s = me->curr; s; s = s->super) {
    if ((msg = StateOnEvent(s, me, msg)) == 0) {;
      if (me->next) {                         /* state transition taken? */
        for (s = me->next, trace = entryPath, *trace = 0;
             s != me->curr; s = s->super)
          *(++trace) = s;                         /* trace path to target */
        while (s = *trace )                     /* retrace entry from LCA */
          StateOnEvent(s, me, &entryMsg);
        me->curr = me->next;
        me->next = 0;
        while (StateOnEvent(me->curr, me, &startMsg),
               me->next) {
          for (s = me->next->super, trace = entryPath,
               *trace = 0; s != me->curr; s = s->super)
            *(++trace) = s;                      /* record path to target */
          while (s = *trace )                         /* retrace  the entry */
            StateOnEvent(s, me, &entryMsg);
```

The topology of a state machine is determined upon construction. The constructor of the concrete HSM is responsible for initialization of all participating `State` objects by setting the `super`-state pointers and the event handlers. An example of a state machine and corresponding data structures is shown in Figure 2. In our approach we do not distinguish between composite-states (states containing substates) and leaf states. All states are potentially composite.

State transitions are implemented as macros: `STATE_START()` and `STATE_TRAN()` (see Listing 2). The first macro (inline member function in C++) handles start transitions (transitions originating from a "black dot" pseudostate). The second macro `STATE_TRAN()` implements a regular state transition and is slightly more complex.

Due to the UML-specified order of invocation, all exit actions must precede any actions associated with the transition, which must precede any entry actions associated with the newly entered state(s). To discover which exit actions to execute, it is necessary to first find the *least common ancestor* (LCA) of the source and target states. For example, the LCA of the transition triggered by event `e4` in Figure 2 is `s2` and the LCA for the transition triggered by event `e1` is `top`.

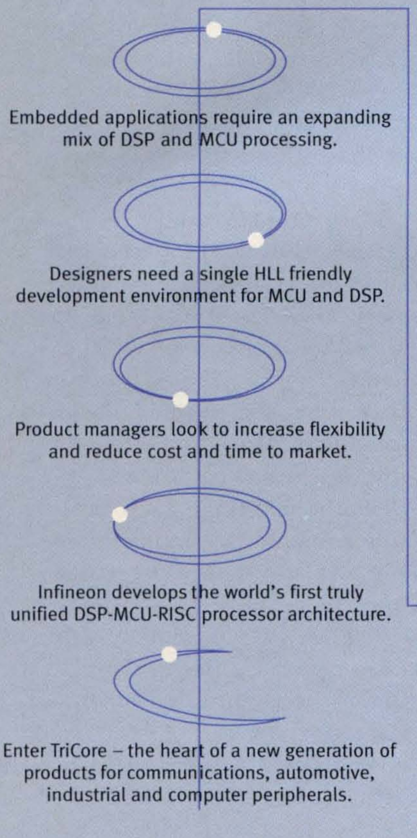
Finding the LCA can be expensive (see method `HsmToLCA_()` in Listing 2). However, for any given transition the LCA needs to be calculated only once. Method `HsmToLCA_()` returns the number of levels from the current state to the LCA rather than a pointer to the LCA state itself. The former is the same for all instances of a given HSM, that is, it is characteristic of the `Hsm` (sub)class rather than individual state machine objects. For this reason it can be stored in a `static` variable shared by all instances.

The `STATE_TRAN()` macro ensures that all exit actions to the LCA will be executed. The user must then explicitly invoke any actions associated with

TRICORE. Three in ONE.

{PERPETUAL THINKING PROCESS}

INFINEON CYCLES



Embedded applications require an expanding mix of DSP and MCU processing.

Designers need a single HLL friendly development environment for MCU and DSP.

Product managers look to increase flexibility and reduce cost and time to market.

Infineon develops the world's first truly unified DSP-MCU-RISC processor architecture.

Enter TriCore – the heart of a new generation of products for communications, automotive, industrial and computer peripherals.

TRICORE is Infineon's award winning architecture unifiying the best of three worlds – real-time capabilities of microcontrollers, computational power of DSPs, and the price/performance benefits of RISC architectures. By enabling efficient single-core DSP-MCU solutions, TriCore reduces complexity and costs, and increases flexibility. Infineon offers TriCore in a wide range of application-specific and general-purpose solutions. The core is also available through Infineon's IP licensing program. TriCore is just the latest example of what happens when a company insists on extending its creative horizons – when its people never stop thinking.

www.infineon.com/tricore

Infineon technologies

Infineon Technologies – A Siemens Company.

Never stop thinking.

LISTING 2a, cont'd. hsm.c—C implementation

```
            me->curr = me->next;
            me->next = 0;
          }
        }
        break;                                          /* event processed */
      }
    }
}
                        /* exit current states and all superstates up to LCA */
void HsmExit_(Hsm *me, unsigned char toLca) {
   register State *s;
   for (s = me->curr; toLca > 0;  toLca, s = s->super)
       StateOnEvent(s, me, &exitMsg);
   me->curr = s;
}
                                  /* find # of levels to Least Common Ancestor */
unsigned char HsmToLCA_(Hsm *me, State *target) {
   State *s, *t;
   unsigned char toLca = 1;
   for (s = me->curr->super; s != 0; ++toLca, s = s->super)
      for (t = target; t != 0; t = t->super)
        if (s == t)
           return toLca;
   return 0;
}
```

the transition and return from the event handler. The framework will then correctly execute any required entry actions.

The state machine engine (method **Hsm::onEvent()** from Listing 2) is small, due mostly to the simple data representation employed. To minimize stack use and maximize performance we were careful to replace potential recursion (natural in hierarchical state machines) with iteration.

Perhaps the weakest part of the implementation lies in execution of entry actions during state transitions. Entry actions must be executed in order from the least deeply nested to the most deeply nested state.[11] This is opposite to the "natural" navigability in our data structure (see Figure 2). This problem is solved by first recording the entry path from the LCA to the target, then "playing it backwards" with execution of entry actions. By applying a technique similar to that described previously for LCA calculation, it is possible to record an entry path only once and avoid repetitive calculation. This optimization trades memory and additional complexity for speed improvement.

Sample application

To illustrate the use of the HSM pattern, consider a simple digital watch (Figure 3). The watch has two buttons—which generate external events—and an internally generated tick event. The different events are handled differently depending upon the mode of operation. The basic watch operates as follows:

- In **timekeeping** mode, the user can toggle between displaying date or current time by pressing the "mode" button
- Pressing the "set" button switches the watch into **setting** mode. The sequence of adjustments in this mode is: hour, minute, day, month. Adjustments are made by pressing the "mode" button, which increments the chosen quantity by one. Pressing the "set" button while adjusting month puts the watch back into timekeeping mode
- While in **setting** mode the watch ignores tick events
- Upon return to **timekeeping** mode the watch displays the most recently selected information, that is, if date was selected prior to leaving **timekeeping** mode, the watch resumes displaying the date, otherwise it displays the current time

A state diagram for the specification given above is depicted in Figure 3b and its partial implementation is shown in Listing 3. We apply the HSM pattern according to the following recipe:

1. Declare a new class, inheriting from **Hsm** class (the **Watch** class)
2. Put into this new class all states (**State** class instances) and other attributes (**tsec**, **tmin**, **thour**, and so on)
3. Declare an event handler method (member function) for every state. Don't forget to declare event handlers for inherited states, like **top**, whose behavior you intend to customize
4. Define the state machine topology (nesting of states) in the new class (the **Watch** class) constructor
5. Define events for the state machine (for example, as enumeration). You can use event-types starting from 0, because the pre-defined events use the upper limit of the **Event** type range (see Listing 1)
6. Define event handler methods. Code entry/exit actions and startup transitions as response to predefined events **ENTRY_EVT**,

EXIT_EVT, and START_EVT, respectively. Provide code for other events using STATE_TRAN() macro for state transitions. Remember to return 0 (NULL pointer) if you handle the event and the initial message pointer if you don't

7. Execute the initial start transition by invoking Hsm::onStart()
8. Arrange to invoke Hsm::onEvent() for each incoming event

Real-time framework

The HSM design pattern is particularly suited for implementing behavior associated with *active objects* (objects that are the roots of threads-of-control in a multitasking system). The concept of active objects exists in many modeling languages under different names: active objects stereotype in UML[9], active instance in Schlaer-Mellor[10], and actor in ROOM[6]. We use the term "actor" because it's most compact.

The typical structure of a framework based on active objects is shown in Figure 4. The Actor class inherits

FIGURE 2 a) State diagram and b) corresponding data structure

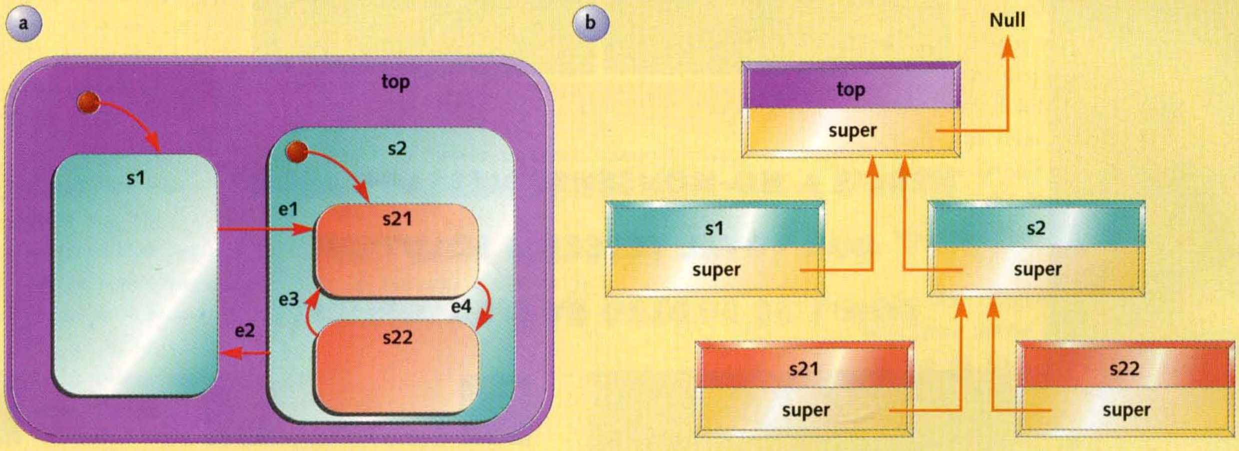
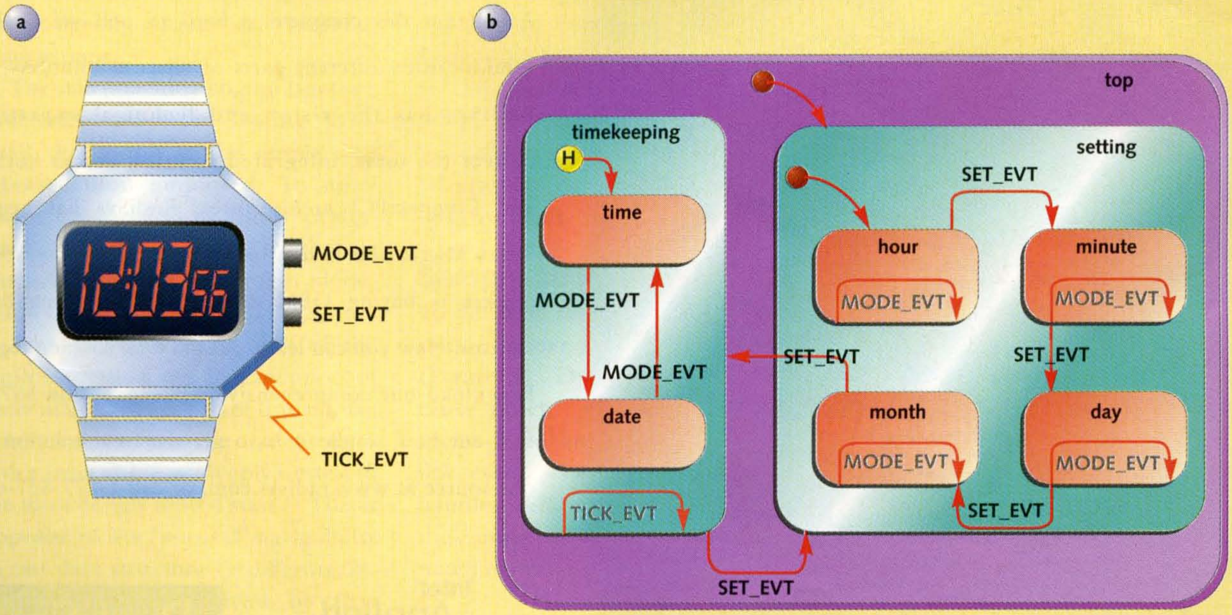


FIGURE 3 a) Simple digital watch events and b) corresponding state diagram



Don´t gamble with success ...

Hitex deals you winning cards!

USB Agent - USB Analyzer

The USB Agent is a full-featured USB-Protocol analyzing instrument ...
A development tool that captures, analyzes and intelligently displays all USB-signals. Right performance - Right price!

In-Circuit Emulators for ...

MX430 family:

For MSP430 microcontrollers, these emulator systems support all members of this TI family of low-power microprocessors, including the ultra-low-power MSP430x11x derivatives. These instruments are proving to be very popular.

C166 family: DProbe167 and DBox167

A modular family of powerful in-circuit emulator systems for the Siemens C16x variants featuring the most advanced adaptation technology: PressOn. Our C161 system with a 64 K trace buffer module is available for less than $3,500. Also supports ST-10 microcontrollers.

68HC12 family: DProbeHC12

For 68HC12 processors; supports maximum speed and all operating voltages; plug and play (no jumpers, no switches); additional BDM interface cable; flash programming built-in; PlugOn trace with 32k frames. Move up to super advanced features with DBoxHC12.

8051 family: MX51/AX51

True real-time emulation for more than 500 variants from all manufacturers. Including derivatives like Atmel 89S8252, Intel 8x931x and Siemens C505C.

and more:

251, 68k, CPU-32, 80x86, 68HC11, Pentium® Processor

hitex

DEVELOPMENT TOOLS

Hitex USA
710 Lakeway Dr., Suite 280
Sunnyvale, California 94086

	Tel.:	(800) 45-HITEX
	Tel.:	(408) 733-7080
	Fax:	(408) 733-6320
	E-mail	info@hitex.com
Hitex Germany	Tel.:	(0721) 9628-133
	Fax:	(0721) 9628-149
Hitex UK	Tel.:	(01203) 69 20 66
	Fax:	(01203) 69 21 31
Hitex Asia	Tel.:	(65) 74 52 551
	Fax:	(65) 74 54 662

www.hitex.com

Advanced technology that brings results...

"The best emulator that I ever used!"

Benefit from our world-wide support:

Austria Tel: +43-1-259727/20 Belgium Tel: +31-77-3078438 PR China Tel: +86-10-62383376 Czech Republic Tel: +420-2-66316661 Denmark Tel: +45-43-424742 France Tel: +33-1-39611414 India Tel: +91-22-8520817 . Israel Tel: +972-3-7657666 Italy Tel: +39-0434-633500 Malaysia Tel: +603-7167591 Netherlands Tel: +31-77-3078438 Pakistan Tel: +92-51-822075 Poland Tel: +48-22-7556983 Russia Tel: +7-812-3252792 Singapore/Indonesia Tel: +65-7496168 South Korea Tel: +82-2-645-0386 Spain/Portugal Tel: +34-91-6401234 Sweden/Norway/Finland Tel: +46-87-404580 Switzerland Tel: +41-1-3086666 Taiwan Tel: +886-2-26983456 Turkey Tel: +90-312-4472700 PBX

LISTING 2b hsm.cpp—C++ implementation

```
static Msg const startMsg = { START_EVT };
static Msg const entryMsg = { ENTRY_EVT };
static Msg const exitMsg  = { EXIT_EVT };
#define MAX_STATE_NESTING 8
                                                             /* Hsm ctor */
Hsm::Hsm(char const *name, EvtHndlr topHndlr)
  : top("top", 0, topHndlr) { this->name = name; }
                                        /* enter and start the top state */
void Hsm::onStart() {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  curr = &top;
  next = 0;
  curr->onEvent(this, &entryMsg);
  while (curr->onEvent(this, &startMsg), next) {
    for (s = next, trace = entryPath, *trace = 0;
          s != curr; s = s->super)
      *(++trace) = s;                            /* trace path to target */
    while (s = *trace )                     /* retrace entry from source */
      s->onEvent(this, &entryMsg);
    curr = next;
    next = 0;
  }
}
                                               /* state machine "engine" */
void Hsm::onEvent(Msg const *msg) {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  for (s = curr; s; s = s->super) {
    if ((msg = s->onEvent(this, msg)) == 0) {                /*processed?*/
      if (next) {                              /* state transition taken? */
        for (s = next, trace = entryPath, *trace = 0;
              s != curr; s = s->super)
          *(++trace) = s;                        /* trace path to target */
        while (s = *trace )                    /* retrace entry from LCA */
           s->onEvent(this, &entryMsg);
        curr = next;
        next = 0;
        while (curr->onEvent(this, &startMsg), next) {
          for (s = next->super, trace = entryPath, *trace=0;
                s != curr; s = s->super)
            *(++trace) = s;                     /* record path to target */
          while (s = *trace )                       /* retrace the entry */
            s->onEvent(this, &entryMsg);
          curr = next;
          next = 0;
        }
      }
```

HSM functionality from `Hsm` and adds to it a thread of execution (for example, via `taskId` attribute and task's `run()` method) and a message queue (via `msgQ` attribute). Actors can only communicate with each other by sending each other messages via message queues. The messages are processed by the HSM in *run-to-completion* steps. Run-to-completion ensures that actors don't have to deal with concurrency issues internally, thereby eliminating a whole class of difficult time-domain problems by design.

SOP vs. OOP

OOP introduces two fundamental types of inheritance: implementation (class) inheritance and interface inheritance.[7] Implementation inheritance defines an object's implementation in terms of another object's implementation. In contrast, interface inheritance enforces only object interface compatibility regardless, of implementation.

Hierarchical state machines introduce a third type of inheritance that is equally fundamental. We call this *behavioral inheritance*. To understand why hierarchy introduces inheritance and how it works, consider an empty (or transparent) substate nested within an arbitrary superstate. If such a substate becomes active it behaves in exactly the same way as its superstate, that is, it *inherits* the superstate's entire behavior. This is analogous to a subclass which does not introduce any new attributes or methods. An instance of such a subclass is indistinguishable from its superclass because, again, everything is inherited exactly.

This property of HSMs is fundamental because it requires only the *differences* from the superstate's behavior to be defined. One observes that all OO design principles (for example, the Liskov Substitution Principle) hold in HSM designs because one deals with inheritance in yet another form. The concept of behavioral inheritance describes the "is-a" ("is-in") relationship between substates

Client-Server Vs. Web Development

by Adam Kolawa

Due to the enthusiastic response of our readers, we have decided to continue our series of columns about software development by taking a look at some of the current trends in the industry. The Web has changed software development forever, and it doesn't take a prophet to know that this trend will continue. As more and more sophisticated applications move online, the Web is no longer just a place for simple, static Web sites. As such, it's time for software developers to take a serious look at the issues surrounding Web development.

The software industry has always been fond of pitting one development technique against another. Nowadays, one of the hottest debates is between supporters of client-server development and Web development. Web proponents sometimes claim client-server is dead; client-server users see glaring weaknesses in the Web paradigm. To make a decision on which to use, it is important to understand the critical differences between the two approaches.

When the industry first began to look at dynamic Web development, they saw the HTML output pages and assumed Web development was limited to providing changing text to viewers. Before long, however, developers realized the full potential of the technology; they could write applications that interacted with databases at the back end, and use a Web browser as a standard GUI at the front end. The problem of teaching people to use a GUI would disappear because any computer owner--and, therefore, any potential software purchaser--knows how to use a Web browser.

It appeared at this point that client-server was in trouble. After all, every client-server application has a learning curve, and often a very steep one, because people must learn to use a proprietary client before they can put the application to full use. When companies realized they could deliver applications with a standard GUI that everyone already knew how to use, they flocked to the Web. Web-based applications offer additional benefits: they give developers the ability to update software in one place--the server--rather than mailing out updates, they streamline maintenance, and they increase the portability of applications. But in our next column, we'll explain why Web development isn't a "silver bullet" solution, and why client-server still has a valuable place in software development

Adam Kolawa, Ph.D., is Chairman and CEO of ParaSoft. You can reach him at ak@parasoft.com

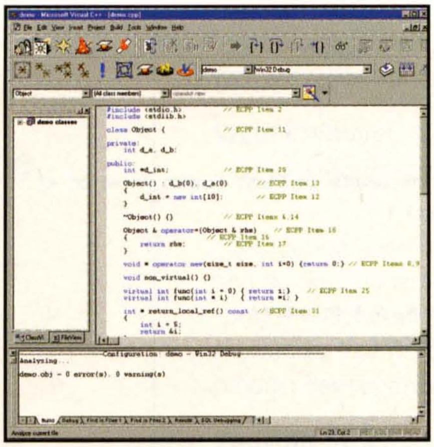
ParaSoft®

ParaSoft®
codewizard®

CodeWizard® prevents C/C++ errors and slashes your debugging time

"CodeWizard is awesome. I am recommending our management to purchase at least 5 copies.... ParaSoft has a lot to offer to the world of C++ development." ***-Piyush Shaw, Arrowsmith Technologies, Inc.***

CodeWizard® is a tool that helps you prevent errors by automatically performing static analysis on C/C++ code. The first thing you'll notice about CodeWizard is that it cuts down on the thing you hate the most: debugging. Preventing errors with CodeWizard is painless and saves you time in the long run.



```
#include <stdio.h>            // ECPP Item 2
#include <stdlib.h>

class Object {                // ECPP Item 11

private:
    int d_a, d_b;

public:
    int *d_int;               // ECPP Item 20

    Object() : d_b(0), d_a(0)       // ECPP Item 13
    {
        d_int = new int[10];        // ECPP Item 12
    }

    ~Object() {}              // ECPP Items 6,14

    Object & operator=(Object & rhs)    // ECPP Item 16
    {                   // ECPP Item 15
        return rhs;           // ECPP Item 17
    }

    void * operator new(size_t size, int i=0) {return 0;} // ECPP Items 8,9

    void non_virtual() {}

    virtual int func(int i = 0) { return i;}     // ECPP Item 25
    virtual int func(int * i)   { return *i; }

    int * return_local_ref() const   // ECPP Item 31
    {
        int i = 5;
        return &i;
```

```
--------------------Configuration: demo - Win32 Debug--------------------
Analyzing...

demo.obj - 0 error(s), 0 warning(s)
```

CodeWizard enforces over 120 sophisticated C/C+ coding standards.

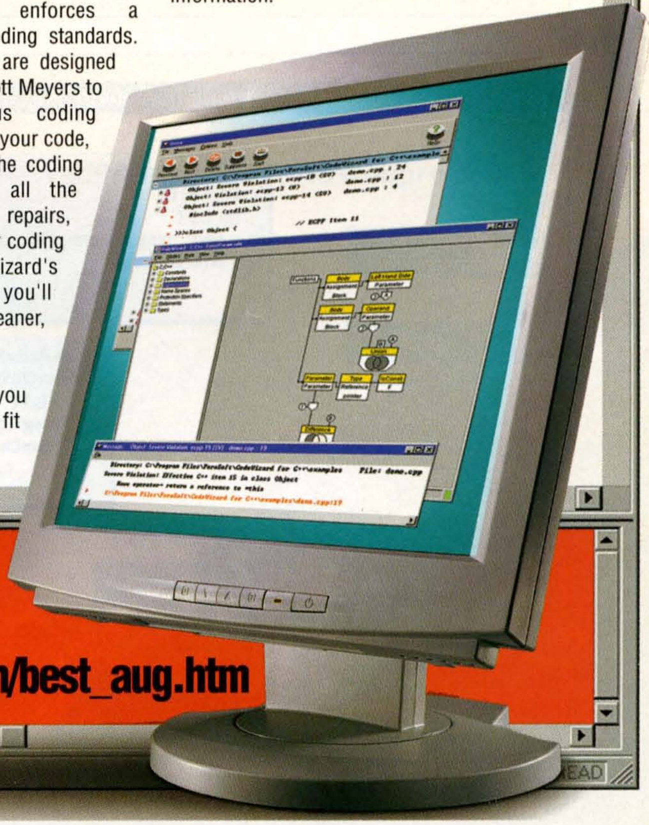
CodeWizard automatically enforces a sophisticated set of C/C++ coding standards. These language-specific rules are designed by industry experts such as Scott Meyers to help you avoid dangerous coding constructs. CodeWizard parses your code, clearly displays violations of the coding standards, and gives you all the information you need to make repairs, including suggestions on better coding constructs. Follow CodeWizard's coding suggestions, and you'll immediately begin to write cleaner, more reliable code.

The key to CodeWizard is that you can customize the program to fit your development schedule.

With a few clicks, you can tell CodeWizard to enforce only the standards that are most important to you at a particular stage of your project. Using the RuleWizard feature, you can point and click to build your own sophisticated coding standards. RuleWizard guides you through the coding standard building process and lets you know when your coding standard is ready for enforcement by CodeWizard.

You won't have any problem adding CodeWizard to your arsenal of development tools, because it installs directly into Windows and UNIX development environments. In Microsoft Developer Studio, you'll begin testing with a single click of the CodeWizard icon. On UNIX platforms, CodeWizard is a wrapper around the compiler. Learn a few simple commands, and you'll be on your way to cleaner code.

If you would like to prevent errors with a customizable tool that conforms to your development cycle, download CodeWizard today at www.parasoft.com/prevention.htm, or call (888)305-0041 for more information.



Download your FREE DEMO at:

www.parasoft.com/best_aug.htm

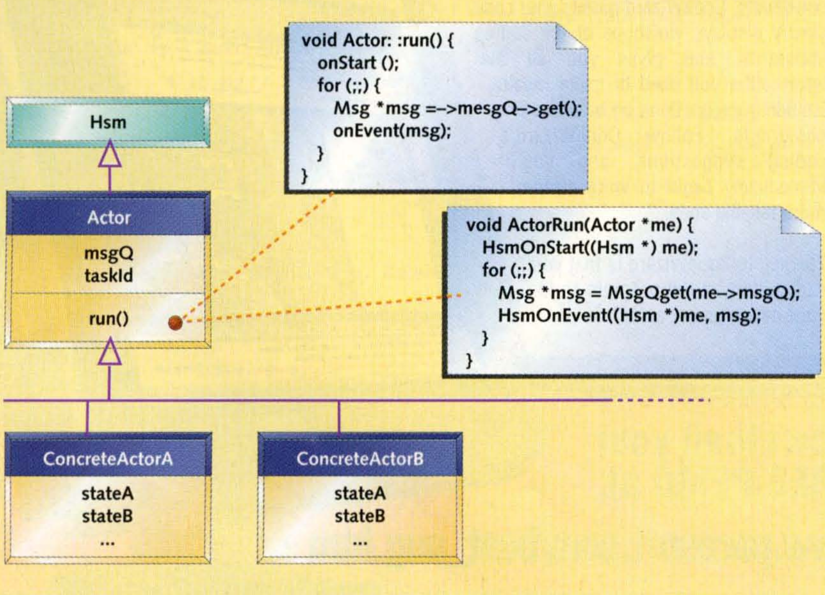
and superstates and should not be confused with inheritance of entire state models.[3]

The analogy between SOP and OOP goes further. Class instantiation and finalization is similar to entering and exiting a state. In both cases special methods are invoked: constructors and destructors for objects, entry and exit actions for states. Even the order of invocation of these methods is the same: constructors are invoked starting from most remote ancestor classes (destructors are invoked in reverse order), and entry actions are invoked starting from the topmost superstate (exit actions are invoked in reverse order).

LISTING 2b, cont'd. hsm.cpp—C++ implementation

```
        break;                                /* event processed */
      }
    }
}
                     /* exit current states and all superstates up to LCA */
void Hsm::exit_(unsigned char toLca) {
  register State *s;
  for (s = curr; toLca > 0;  toLca, s = s->super)
    s->onEvent(this, &exitMsg);
  curr = s;
}
                            /* find # of levels to Least Common Ancestor */
unsigned char Hsm::toLCA_(State *target) {
  State *s, *t;
  unsigned char toLca = 1;
  for (s = curr->super; s != 0; ++toLca, s = s->super)
    for (t = target; t != 0; t = t->super)
      if (s == t)
        return toLca;
  return 0;
}
```

FIGURE 4 Structure of real-time framework based on HSM



```
void Actor: :run() {
  onStart ();
  for (;;) {
    Msg *msg =–>mesgQ–>get();
    onEvent(msg);
  }
}
```

```
void ActorRun(Actor *me) {
  HsmOnStart((Hsm *) me);
  for (;;) {
    Msg *msg = MsgQget(me–>msgQ);
    HsmOnEvent((Hsm *)me, msg);
  }
}
```

A final similarity between OOP and SOP lies in the way they are most efficiently implemented. Although polymorphism can be implemented in many ways, virtually all C++ compilers implement it in the same way: by using function pointers grouped into virtual tables. In view of the deep analogy between SOP and OOP, it is therefore not surprising that arguably the most efficient implementation of HSMs is also based on function pointers grouped into states. These simple state objects define both behavior and hierarchy but are not specific to any particular instance of a state machine. The same holds for virtual functions, which are characteristics of the whole class rather than specific to any particular object instance. For this reason we observe that state objects could (and probably should) be placed in v-tables and be supported as a native language feature.

Improvments to UML?

UML state diagrams do not provide any graphical representation of state reactions, which are reactions to events not causing state transitions. In practice however, reactions are common and sometimes the whole state hierarchy is most naturally designed to reuse group reactions rather than group transitions. In these cases a UML state diagram cannot be properly understood because it is simply incomplete.

We propose to include reactions in state diagrams as directed lines starting and finishing in the same state and lying entirely *within* that state. (An example of this notation is shown in Figure 3 for reactions to `TICK_EVT` and `MODE_EVT`.) Please note that this nota-

LISTING 3a Simple watch HSM: C implementation

```
#include "hsm.h"

typedef struct Watch Watch;
struct Watch {
  Hsm super;                                         /* superclass */
  int tsec, tmin, thour, dday, dmonth;
  State timekeeping, time, date;
  State setting, hour, minute, day, month;
  State *tkeepingHist;
};

enum WatchEvents {
  MODE_EVT,
  SET_EVT,
  TICK_EVT
};
                                          /* timekeeping state handler */
Msg const *WatchTimekeepingHndlr(Msg const *msg) {
  switch (MsgGetEvt(msg)) {
    case START_EVT:
      STATE_ENTER(me->tkeepingHist ? me->tkeepingHist :
                  &me->time);
      return 0;
    case SET_EVT:
      STATE_TRAN(&me->setting);
      printf("Watch::timekeeping-SET;");
      return 0;
    case TICK_EVT:
      WatchTimeTick(me);
      return 0
    case EXIT_EVT:
      me->tkeepingHist = STATE_CURR(me);
      return 0;
  }
  return msg;
}
/*... other state handlerds ... */
                                                 /* Watch constructor */
void WatchCtor(Watch *me) {
   HsmCtor((Hsm *)me, "Watch", (EvtHndlr)Watch_top);
   StateCtor(&me->timekeeping, "timekeeping",
      &((Hsm *)me)->top, (EvtHndlr)Watch_timekeeping);
      StateCtor(&me->time, "time", &me->timekeeping,
            (EvtHndlr)Watch_time);
      StateCtor(&me->date, "date", &me->timekeeping,
            (EvtHndlr)Watch_date);
   StateCtor(&me->setting, "setting", &((Hsm *)me)->top,
            (EvtHndlr)Watch_setting);
      StateCtor(&me->hour, "hour", &me->setting,
            (EvtHndlr)Watch_hour);
```

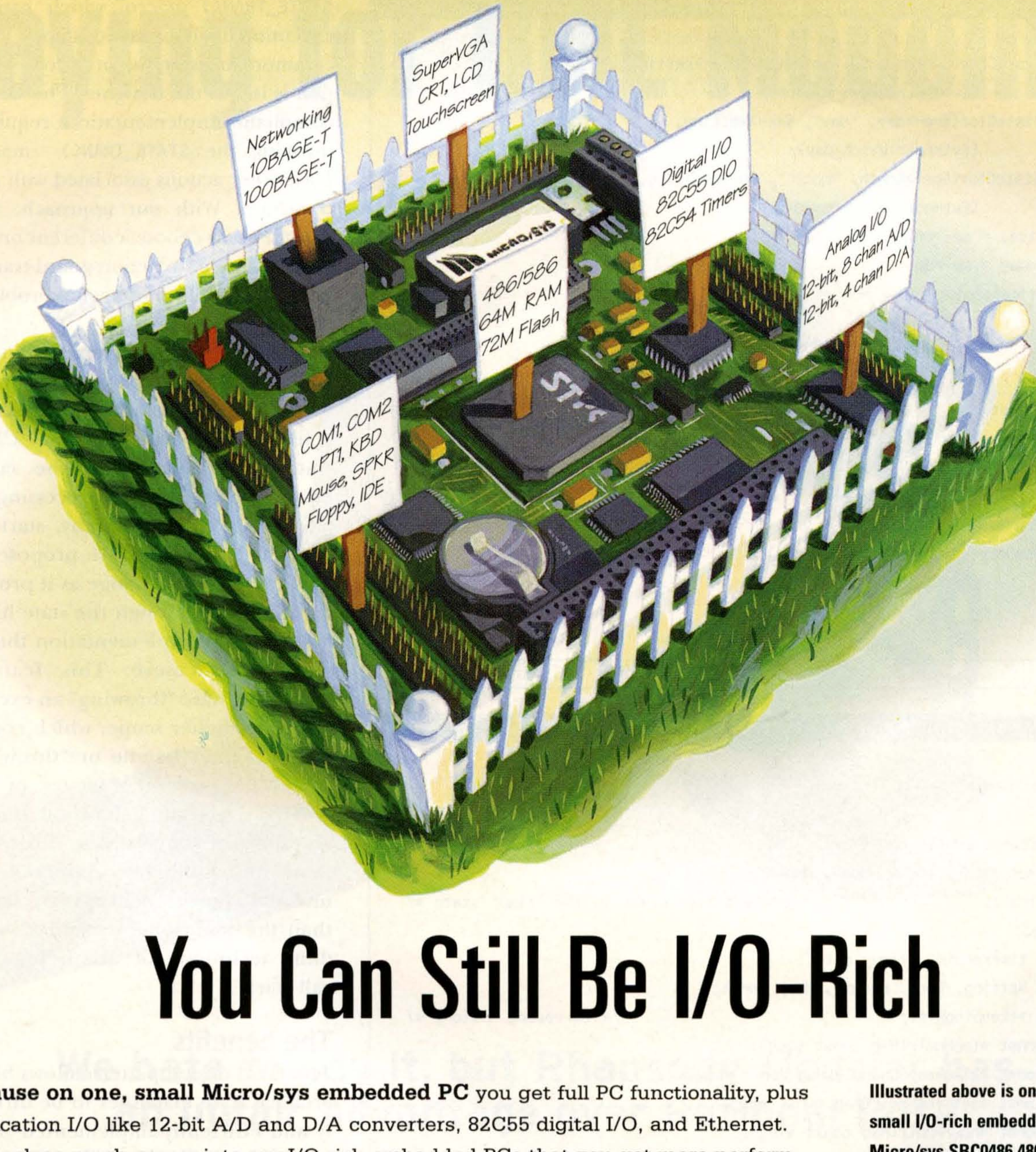
tion is different from self-transitions, which also begin and end in the same state, but lie entirely *outside* the state. Self-transitions are different from reactions because they cause execution of entry and exit actions.

UML statecharts distinguish composite-states (states with substates) from leaf-states (states without substates) in that they only allow leaf-states to be active. This is an unnecessary limitation, which occasionally creates degenerate (empty) substates. To extend the analogy with OOP, this is like saying that a class with subclasses must necessarily be abstract. Our HSM implementation does not distinguish between composite and leaf states; it allows any state to be active.

In our experience with receiver-applications we frequently encountered the following situation: in response to a given event, we wished to perform an action (for example, update a digital filter) and then—depending on the state of the filter—potentially take a (conditional) state transition. We did not want to treat the filter update as part of the guard condition because this would add a side effect to the guard (typically a bad idea[11]). The only alternative is to treat the filter update as an action.

UML requires that actions associated with transitions be executed *after* all exit-actions from the source state but before any entry-actions to the target state.[11] At this point however, it is not yet know if the transition will be required at all. In general, this aspect of UML semantics makes it difficult to mix conditional execution of reactions and transitions. A UML-compliant solution would require specification of a pure reaction (update of the digital filter in this case) and then conditional propagation of *another* event to "self," specifically to trigger a pure state transition.

Because our implementation performs state transitions using the



Networking
10BASE-T
100BASE-T
SuperVGA
CRT, LCD
Touchscreen
Digital I/O
82C55 DIO
82C54 Timers
Analog I/O
12-bit, 8 chan A/D
12-bit, 4 chan D/A
MICRO/SYS
486/586
64M RAM
72M Flash
COM1, COM2
LPT1, KBD
Mouse, SPKR
Floppy, IDE

LISTING 3a, cont'd. Simple watch HSM: C implementation

```
      StateCtor(&me->minute, "minute", &me->setting,
            (EvtHndlr)Watch_minute);
      StateCtor(&me->day, "day", &me->setting,
            (EvtHndlr)Watch_day);
      StateCtor(&me->month, "month", &me->setting,
            (EvtHndlr)Watch_month);
   me->tsec = me->tmin = me->thour = 0;
   me->dday = me->dmonth = 1;
   me->timekeepingHist = NULL;
}

void main() {                                      /* test harness */
   Watch watch;
   WatchCtor(&watch);
   HsmOnStart((Hsm *)&watch);
   for (;;)  {
      Msg *msg = getEvt();                /* block until event arrives */
      HsmOnEvent((Hsm *)&watch, msg);
   }
}
```

LISTING 3b Simple watch HSM: C++ implementation

```
#include "hsm.h"

class Watch : public Hsm {
  int tsec, tmin, thour, dday, dmonth;
  timeTick();                 /* process tick event in the 'time' state */
protected:
  State timekeeping, time, date;
  State setting, hour, minute, day, month;
  State *tkeepingHist;                              /* to record history */
  Msg const *topHndlr(Msg const *msg);
  Msg const *timekeepingHndlr(Msg const *msg);
  Msg const *settingHndlr(Msg const *msg);
  Msg const *hourHndlr(Msg const *msg);
  /*... other state handler methods */
public:
  Watch();                          /* ctor for defining state hierarchy */
};

enum WatchEvents {
  MODE_EVT,
  SET_EVT,
  TICK_EVT
};
```

STATE_TRAN() macro, which causes execution of all exit-actions up to least common ancestor, the order of execution is left to the designer. The UML-compliant implementation requires invoking the STATE_TRAN() macro *before* other actions associated with the transition. With our approach, the designer may choose a different order (for example, reaction-guard-transition), if it would simplify the problem at hand.

A final (proposed) extension to UML statecharts is associated with event handling at different levels of a hierarchical state machine. UML statecharts require that the same event be presented for processing to all levels of the hierarchy, starting with the active state. We propose to allow an event to change as it propagates upward through the state hierarchy. In our implementation this is simple to achieve. This feature would facilitate "throwing" an exception to a higher scope, which could in turn either handle or "throw" it again. Because outer states of an HSM are typically behavioral generalizations of inner states, this technique for handling exceptions is natural and arguably, makes more sense than the traditional exception handling technique of unwinding the call stack.

The benefits

The HSM design pattern allows hierarchical state machines to be directly and efficiently implemented in C or C++ without code synthesizing tools. The event-handler methods provide a concise textual representation of the state model and allow high-level structure and low-level details to be accessed easily. The simplicity of the event-handlers leads to "housekeeping" code[3]—portions of software that can be automatically generated by tools—that is trivial to write by hand.

The HSM pattern is flexible, allowing even fundamental changes

LISTING 3b, cont'd. Simple watch HSM: C++ implementation

```
                                             /* timekeeping state handler */
Msg const *Watch::timekeepingHndlr(Msg const *msg) {
   switch (msg->getEvt()) {
      case START_EVT:
         STATE_ENTER(tkeepingHist ? tkeepingHist : &time);
         return 0;
      case SET_EVT:
         STATE_TRAN(&setting);
         printf("Watch::timekeeping-SET;");
         return 0;
      case TICK_EVT:
         timeTick();
         return 0
      case EXIT_EVT:
         tkeepingHist = STATE_CURR();
         return 0;
   }
   return msg;
}
                                          /*... other state handlerds ... */
                                                          /* Watch ctor */
Watch::Watch()
   :  Hsm("Watch", (EvtHndlr)topHndlr),
   timekeeping("timekeeping", &top,
      (EvtHndlr)timekeepingHndlr),
   time("time", &timekeeping, (EvtHndlr)timeHndlr),
   date("date", &timekeeping, (EvtHndlr)dateHndlr),

   setting("setting", &top, (EvtHndlr)settingHndlr),
   hour("hour", &setting, (EvtHndlr)hourHndlr),
   minute("minute", &setting, (EvtHndlr)minuteHndlr),
   day("day", &setting, (EvtHndlr)dayHndlr),
   month("month", &setting, (EvtHndlr)monthHndlr)
{
   tsec = tmin = thour = 0;
   dday = dmonth = 1;
   tkeepingHist = NULL;
}
                                                        /* test harness */
void main() {
   Watch watch;
   watch.onStart();
   for (;;) {
      Msg *msg = getEvt();                 /* block until event arrives */
      watch.onEvent(msg);
   }
}
```

in state machine topology to be accomplished easily, even late in the process. Due to their hierarchical nature, models can be developed in incremental steps and remain executable throughout the development cycle. By requiring only specialization of behavior to be coded at nested levels of the state machine, common policy mechanisms (for example, exception handling) can be handled naturally. The state machine engine can easily be instrumented (an example is available in code accompanying this article at *www.embedded.com/code.html*) to produce execution trace, message sequence charts, or even animated state diagrams. In practice, however, we found it most useful to use a standard debugger to step through interesting parts of the code.

This implementation of the HSM pattern is no more complex than an internal implementation of inheritance or polymorphism in C++ and, in fact, has many similarities (both are based on function pointers). Given the many parallels, it seems reasonable to suggest that state-oriented programming should be directly supported by a (state-oriented) programming language in the same way that OOP is supported by object-oriented languages. We see this as beneficial for a couple of reasons. First, the compiler could place state objects in a virtual table and perform memory and performance optimizations at compile time (for example, computation of LCAs for state transitions). Second, the compiler could check consistency and well-formedness of the state machine, thereby eliminating many errors at compile time. In our view this is one direction in which C/C++ could evolve to better support future real-time applications. **esp**

Miro Samek is lead software architect at IntegriNautics Corp. He holds a PhD in *physics from Jagiellonian University in Cracow, Poland. Miro previously worked at GE Medical Systems where he developed real-time software for diagnostics X-ray equipment. He may be reached at miro@integrinautics.com.*

Paul Y. Montgomery is software group leader at IntegriNautics Corp. He holds a PhD in Aero/Astro engineering from Stanford University, specializing in control systems applications based on the Global Positioning System. He may be reached at paulm@integrinautics.com.

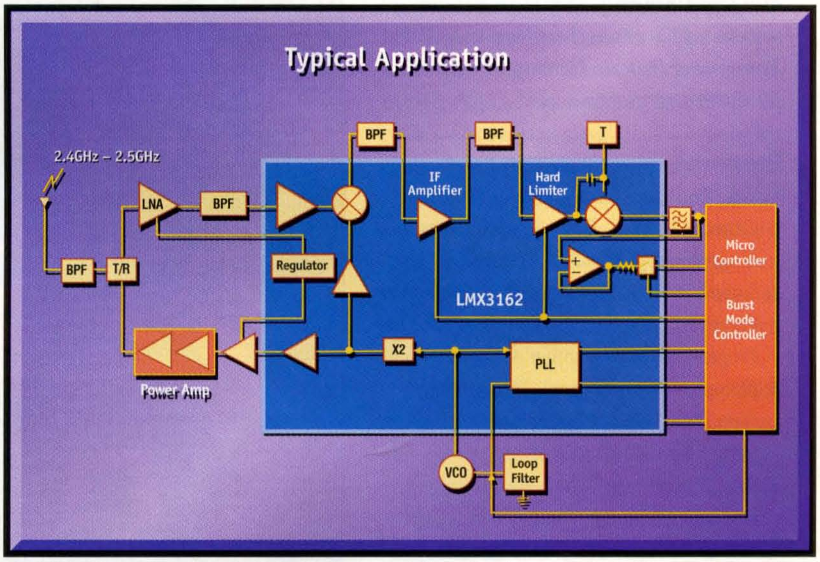
Resources

1. Bell, Rodney, "Code Generation form Object Models," *Embedded Systems Programming*, March 1998, p. 74.
2. Samek, Miro, "Portable Inheritance and Polymorphism in C," *Embedded Systems Programming*, December 1997, p. 54.
3. Douglass, Bruce Powell. *Doing Hard Time*. Reading MA: Addison-Wesley Longman, 1999.
4. Douglass, Bruce Powell. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Reading, MA: Addison-Wesley, 1998.
5. *www.i-logix.com*
6. Selic, Bran, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. New York City: John Wiley & Sons Inc., 1994.
7. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
8. *www.omg.org/docs/97-96-02.pdf*, UML 1.1 Alpha—Behavioral Elements: State Machines.
9. Douglass, Bruce Powell and Srini Vason, "Temporal Models in UML," *Dr. Dobbs Journal*, December 1999.
10. Levkoff, Bruce, "A Schlaer-Mellor Architecture for Embedded Systems," *Embedded Systems Programming*, November 1999, p. 88.
11. Douglass, Bruce Powell, "UML Statecharts," *Embedded Systems Programming*, January 1999, p. 22.

National Delivers Industry's First 2.4GHz Integrated ISM Transceiver

LMX3162 Provides Low Cost Solutions for Emerging Wireless Applications

WOW
- System RF Sensitivity to –93dBm; RSSI Sensitivity to –100dBm
- System Noise Figure 6.5dB (typ.)
- 70µA (max.) Power Down Mode
- Works with Unregulated 3.0V – 5.5V Supply
- Two Regulated Voltage Outputs for Discrete Amplifiers
- High Gain (85dB) Intermediate Frequency Strip



Ideal for 2.4GHz Voice/Cordless Applications, Wireless Networking Proprietary Protocols, and Products Utilizing Bluetooth, HomeRF, 802.11b, or 2.4GHz US DECT

For More Information on LMX3162:
www.national.com
1-800-272-9959

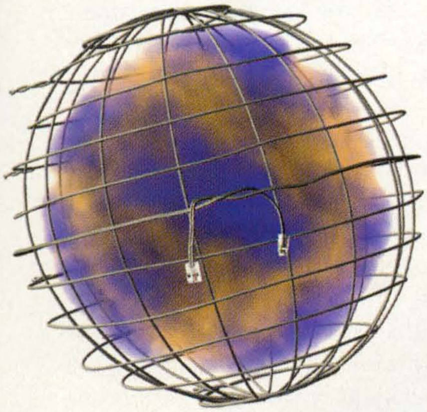
Evaluation Board Available

Free CD-ROM Data Catalog Available at freecd.national.com

HomeRF Bluetooth.

National Semiconductor®

Michael Barr



Schedule Slips

Internet Appliance Design

Have you ever had one of those days where nothing seems to go your way? Well, with regard to my own UDP/IP development work, I've just had "one of those months." By now, I had hoped to have a pretty solid body of code to share with you, including the protocol stack and some client and server application software. I hadn't expected to be done with the development altogether, and I'm far from being done describing the code to you, but I had at least hoped to have my stack up and running on the µC/OS-II operating system, with a fake network driver passing the UDP/IP packets back and forth in local RAM.

In fact, I was actually quite close to achieving that milestone prior to writing last month's column. However, in the time since, I've had absolutely zero chance for any further development. Admittedly, part of the problem was that I was on the road for two weeks of the four and tried to take some vacation during a third. But even when I did find time to sit down at my computer to program, my efforts were thwarted. Reflecting on my difficulties has led me to some broader concerns.

Progress?

How have we allowed ourselves to become so dependent on a technology that is so unreliable? The technology of which I write is, of course, the personal computer. And by that I mean not only Wintels, but also Macs and even Unix workstations. They're all built on the same hardware these days anyway.

Now don't get me wrong, I understand that a modern computer is a terribly complex and wonderfully adaptable system, whose hardware and software requirements are primarily driven by dollars and cents. But maintaining one of these things can be a nightmare and a substantial hidden cost of ownership.

Try as I might to avoid installing new software unless absolutely necessary; run the most reliable operating systems (my three machines run a combination of Windows 2000 and Linux), backup religiously; scan every new file for viruses; and never, ever fiddle with PC hardware (all rules adopted after painful experiences with the alternatives), it's a good week when something major doesn't go wrong on my network.

This week it was a hard disk failure that kept me from my work. The drive that failed was only about one and a half years old, as is the machine in which it was installed. Of course, I let this machine (my primary development platform and "gateway to the Internet") run 24 hours a day, 365 days a year. But the machine is plugged into a UPS that regulates the quality of the power signal and is only shut down and restarted about three or four times a month. Is 13,000 hours of life the best this hard disk manufacturer (a big one) can do? Of the eight or nine drives I've owned personally over the years, three have failed, two of them before their warranties had expired!

As a group, we tend to choose proven processors over new ones, the same programming languages over and over again, internally developed operating systems, and so on. We are slow to incorporate the latest and greatest the research community has to offer. And that is a good thing.

Remember the first color TV you or your family owned and how it lasted for 20 years before you bought a second? Now we go through TVs as if they were candy bars. We go through PCs at a similar rate, both at home and at work. Sure, more of us than ever have these gadgets. But are we really making progress?

A major flaw of our current social system is that we measure our overall success and progress by an increase in the total amount of stuff produced per year (if I were writing for *The Economist*, I'd simply say GDP). But are we really better off individually or as a society with a new TV, PC, PDA, cell phone, car, house, (and spouse?) every three to five years, especially if the replaced items are, by then, broken or of little use to others?

Upgrade cycles

The most serious problem on the horizon is that more and more products look like computers on the inside. Of course, this is great news for our industry; embedded software developers should have no shortage of job oppotunities. The problem, though, is the price we'll pay as a society if every one of those embedded computers isn't built to be reliable and long lasting. I bet every *ESP* reader could tell a story or two about a product within their company that was doomed to failure from the start. In particular, projects not given enough time, money, or engineers to do the job right are likely to develop products that look great in the store but won't last long in the hands of an actual user.

Embedded systems designers have traditionally been conservative in their technology choices. As a group, we tend to choose proven processors (with multiple sources only, please) over new ones, the same programming languages over and over again, internally developed operating systems, and so on. We are slow to incorporate the latest and greatest the research community has to offer. And that is a good thing.

When I look around my office and my home I do see some products that are built around microprocessors yet remain reliable. I count among these: a 10BaseT network hub, a fax machine, a stereo receiver (the tape drive and CD player bought at the same time are long since dead), an alarm clock, and a bread machine. All of these products have been with me for as long as I can remember and show no signs of aging.

What sets these products apart is their specificity of purpose and simplicity of design. Each of them does one thing and does it very well. Where mechanical components are required (in the fax and bread machines, in particular), it is clear even to a casual observer that the designs have been optimized for decades of heavy use. In all of these products, a minimum number of components are used. Only what is needed is there. There is none of the "upgradeable this," "replaceable that," and "updateable firmware" that is becoming all too common with all sorts of products these days.

As an engineer, I tend to be particularly skeptical of products whose firmware can be upgraded. Perhaps I've just worked at one too many companies where the attitude (of both managers and developers) has been: since we can upgrade the firmware after we ship, we don't have to get the product completely working before we start manufacturing. That attitude, combined with pressure to get products out the door before some other company beats you to it, is a recipe for disaster (or a lawsuit).

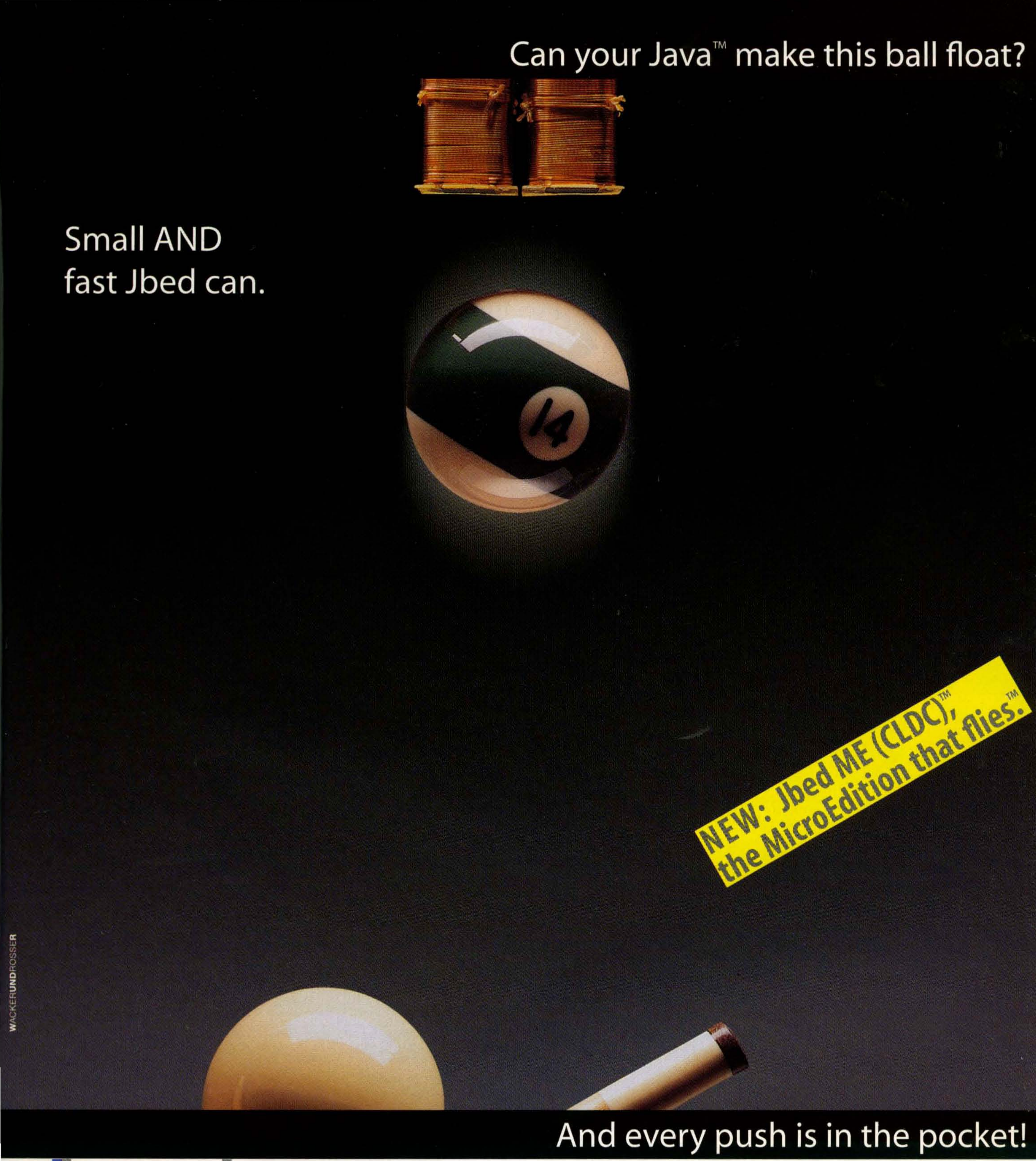
I don't want to upgrade my alarm clock, I just want it to tell me the time. Build systems that work and people will buy them, use them, and love them. Build systems that are updatable, yet unreliable, at your peril.

A new milestone

In the original plan, this month's column was supposed to be about the IP layer of my protocol stack. Unfortunately, some of my UDP/IP code was among the data not recently backed up and has gone to the big bit bucket in the sky. The ARP code, most of which was printed in the magazine, is fine. And the IP code had already been copied to my template for this month's column before the crash, so that's fine too. But I will have to spend some time over the next few weeks getting back to where I was overall.

I'm going to postpone our discussion of IP until I can sit down with my code and make sure that what I'm showing you is reliable. I take that aspect of my work very seriously, and I hope you do too. If all goes well, I'll get to IP next month. In the meantime, try to stay connected. **esp**

Michael Barr is the editor-in-chief of Embedded Systems Programming. *He holds BS and MS degrees in electrical engineering from the University of Maryland. Prior to joining the magazine, Michael developed embedded software and device drivers. He is also the author of the book* Programming Embedded Systems in C and C++ *(O'Reilly & Associates). Michael can be reached via e-mail at mbarr@cmp.com.*


Can your Java™ make this ball float?
Small AND
fast Jbed can.
NEW: Jbed ME (CLDC)™,
the MicroEdition that flies.™
And every push is in the pocket!

WACKERUNDROSSER

Jbed

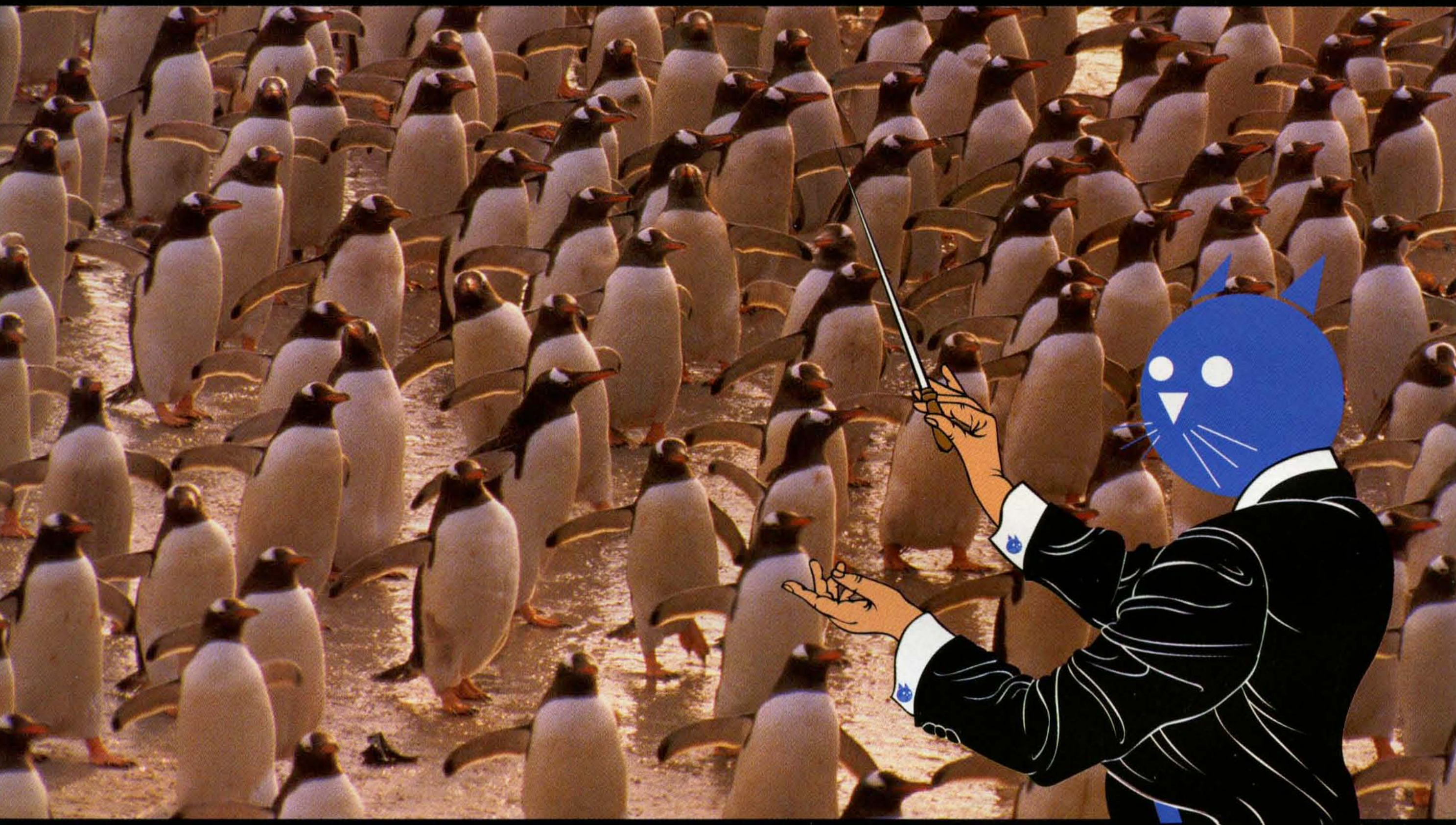
The embedded real-time Java™ Virtual Machine

Jbed runs faster than other Javas because it always compiles, never interprets. Its small memory footprint allows low-cost hardware with less memory. You save time and cost because Jbed offers the productivity of the Java™ language to the embedded world. **Included features are:** •Integrated development environment (IDE) •Deadline-driven/time-triggered scheduling with admission testing •Full support of the Java™ language as specified by Sun Microsystems •Incremental Garbage Collector compliant with hard real-time •Reflection, serialization •Dynamic loading of Java™ byte code •All standard communication protocols supported •RTOS and JVM (Java™ Virtual Machine) in one!

esmertec inc., Technoparkstrasse 1, CH-8005 Zürich, Switzerland
www.esmertec.com, info@esmertec.com, Tel. +41 1 445 37 30, Fax +41 1 445 37 34

esmertec
your partner for embedded real-time Java™

ORCHESTRATING EMBEDDED LINUX SOLUTIONS



It's been true in music since the first duet. And today, it's just as true in the world of Linux. A technology may play beautifully, but for a solution to be complete, this technology must also play in harmony. With compelling complementary technologies. With worldwide, world-class consulting, support, and training services. Without one false note.

That's what we at LynuxWorks bring to the performance. As Lynx Real-Time Systems, our company developed LynxOS® RTOS, a true virtuoso in developing embedded real-time applications and the closest RTOS to Linux today. Now, with BlueCat™ Linux, we've expanded our repertoire to embedded Linux. Embedded systems designers can use either of these superb soloists, or tap the natural synergies of both, to speed new product development and more precisely match product and market requirements.

To mark this change, our company has taken a new name. One that integrates our past with our future. And, like a seasoned maestro, we've brought all our resources – experience, technology, and support – with us.

Don't miss a beat. Find out more about LynuxWorks. The company that orchestrates embedded Linux solutions. The company that can help you hit new high notes in embedded systems design.

Lynx Real-Time Systems is now

LYNUXWORKS™

LynxOS™
FROM LYNUXWORKS

BlueCat™
FROM LYNUXWORKS

www.lynuxworks.com

THOMAS BESEMER

Linux, Interrupted

Every operating system has different mechanisms and approaches to handling interrupts. This article lays out the specifics of handling interrupts in Linux. It presents an overview of the Linux runtime environment, discusses how to communicate with hardware peripherals, and provides a working example of servicing an interrupt from application code.

A significant difference between the Linux execution environment and typical real-time operating systems is the memory model. Linux, natively, executes with protected memory space: processes are isolated from other processes through the kernel and underlying hardware memory management unit (MMU). Processes are also isolated from the underlying hardware—application code can't directly read and write peripheral registers. Figure 1 provides a visual representation of the Linux execution environment.

Almost all embedded operating systems can be depicted as in Figure 1; the primary difference with Linux is that each "layer" in this figure is truly isolated from the others. With just a handful of exceptions, embedded operating systems typically use a "flat" memory model, in which all software components in the system (OS, application tasks, device drivers, and so on) are able to read and write from and to any location in memory. Device drivers may or may not be employed in typical environments, because the developer has the choice of either using a device driver or communicating directly with hardware devices from within the application code itself.

In a Linux environment, application code runs as a set of processes and/or group of POSIX threads within one process. In order for the processes and threads to communicate with underlying hardware, the developer must install one or more device drivers to support the various hardware peripherals. There are two basic schemes for communication with hardware in a Linux environment. The first is to place the peripherals on a PCI bus. The second approach is to memory- or I/O-map the peripherals. Both approaches require a device driver.

When PCI is not employed, the system designers must map the hardware's registers into a space that is accessible to device driver code.

FIGURE 1 Linux execution environment

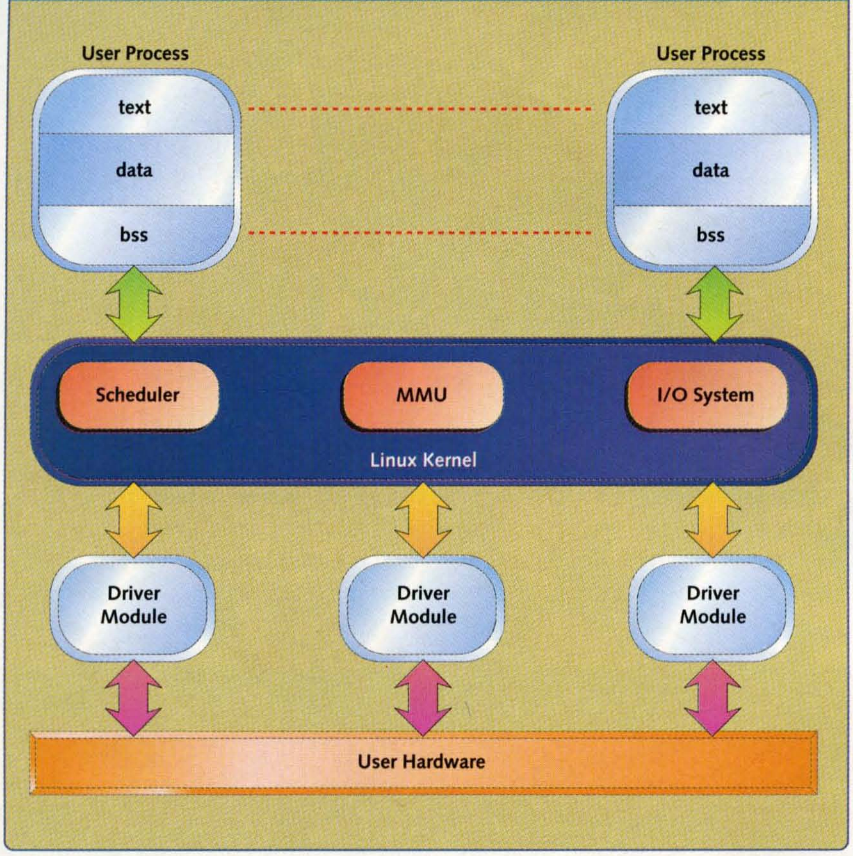
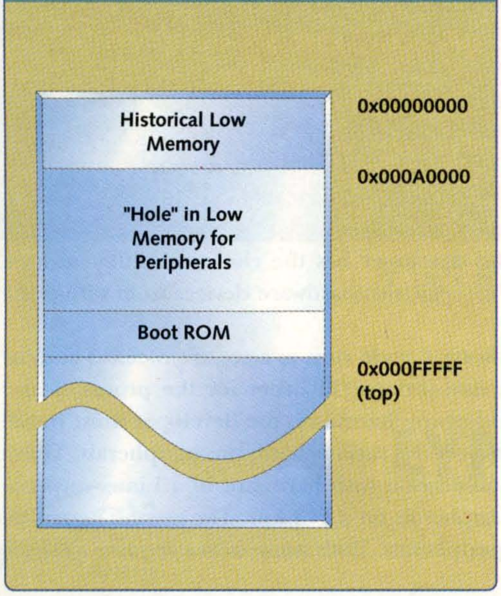


FIGURE 2 PC memorylayout



With PCI-based peripherals, the application-level processes or threads may share an underlying PCI driver to communicate with one or more peripherals, supporting and interfacing with them through this common driver. In this configuration, the PCI driver provides support for interrupts from the peripherals. Application processes or threads read or write the peripheral, and may be blocked within the kernel space until the driver services an interrupt.

With memory- or I/O-mapped peripherals, the developer must design and implement a specific driver to support each peripheral. Interrupts from peripherals are handled at the driver level, and it is important that the application developer understand how to approach servicing the interrupt from the application (process) level.

This article discusses how to work with memory- or I/O-mapped peripherals. It concentrates on the basic structure of a device driver that supports interrupts.

Hardware interface

When PCI is not employed, the system designers must map the hardware's registers into a space that is accessible to device driver code. The registers may be mapped into memory space, with the peripheral appearing as a chunk of "memory" in the system. Or, if the processor architecture supports it, the registers may be mapped into a dedicated I/O space. For example, x86 processors provide an I/O space that can be accessed through the assembly language instructions `in` and `out`. A basic understanding of both memory and I/O-mapped peripherals is extremely helpful.

The PC platform provides a convenient foundation for experimentation with Linux. All driver and code examples in this article were tested on a PC running Red Hat Linux.

Figure 2 provides a block diagram of the basic PC platform memory layout. The most notable aspect of this block diagram is the memory region below 1MB. This region has ties back to the original IBM PC and 8086 architecture, which had only 20 bits of address space. A "hole" sits in memory, between 640K and the start of the boot ROM (also known as the BIOS). This "hole" is where VGA cards and other memory-mapped PC peripherals placed their shared buffers in the pre-PCI era.

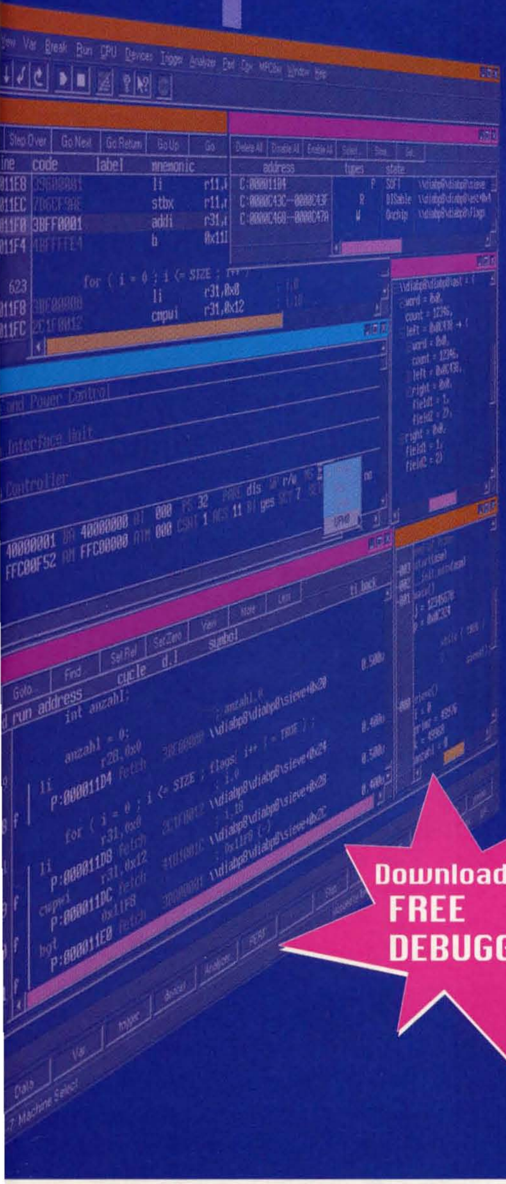
This is important because designers planning to use a standard PC-style platform for their product must keep the following in mind:

TRACE32®

IN-CIRCUIT-EMULATORS & DEBUGGERS

ONE IDE SERVES ALL!

TRACE32-PowerView



TRACE32-ICD
the highly cost effective In-Circuit Debugger

TRACE32-FIRE
the Fully Integrated Risc Emulator at highest speeds

TEST THE BEST!

- *Professional tools exceed highest demands*
- *The ultimate in technology and speed*
- *Greatest variety of supported CPUs (ARM, C167, H8S, PowerPC, SH2, ST10 and more)*
- *Unique modularity and open concept*
- *Test and compare our comfortable tools by downloading our simulators*

Embedded Systems Conference
San Jose, CA
Sept. 26-28, 2000

Download
FREE
DEBUGGER!

Lauterbach, Inc.
4, Mount Royal Ave
Marlborough, MA 01752
Fax: 508-303-6813
Phone: 508-303-6812
e-mail: info_us@lauterbach.com

Lauterbach, Inc.
13256 SW. Hillshire Drive
USA-Tigard, OR 97223
Fax: 503-524-2223
Phone: 503-524-2222
e-mail: info_us@lauterbach.com

LAUTERBACH

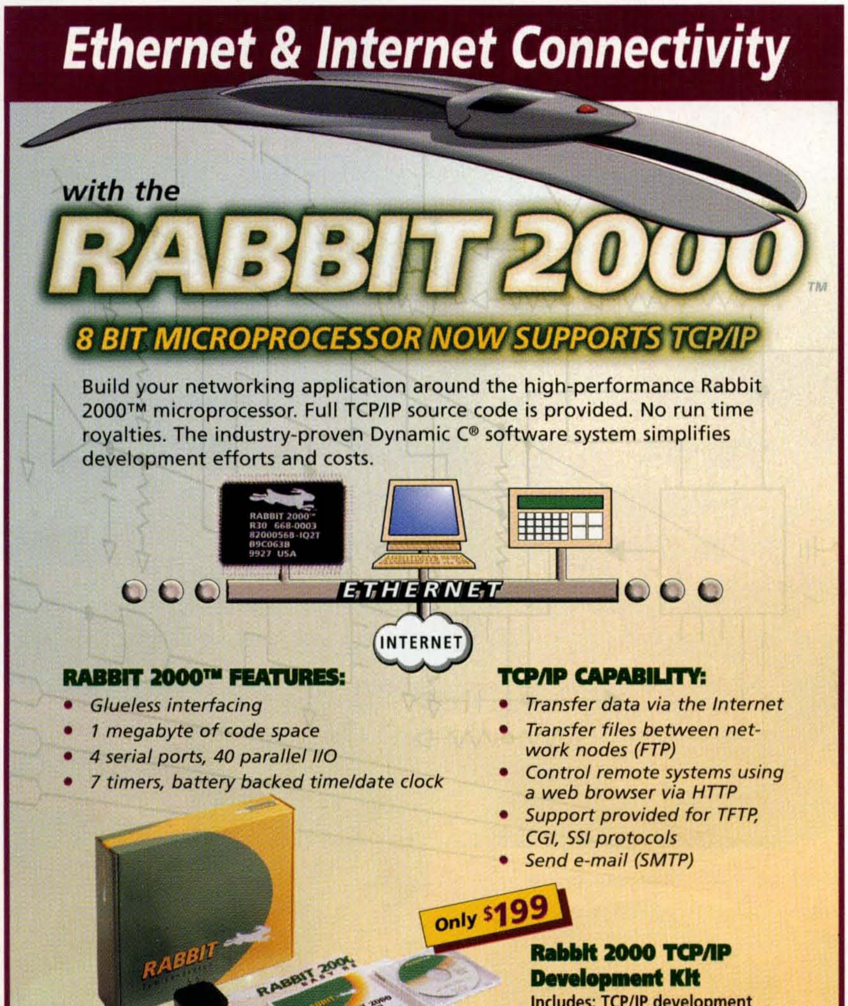
Free simulators available in the Internet:

http://www.lauterbach.com/download.html

LISTING 1 Output from lsmod

```
Module    Size     Used by
ctr05     1688     0 (unused)
tulip     25252    1 (autoclean)
```

- The ISA/EISA bus has only 20 address bits, so memory-mapped peripherals must reside below 1MB
- Memory-mapped I/O must fit into the location between 640K and the bottom of the boot ROM, and must not overlap in memory space with any other hardware present in the system



The Linux kernel ignores this region of low memory in that it does not try to use it. However, device drivers are able to read and write from and to this space, allowing communication with any potential hardware devices that decode in this region of memory.

When working with custom hardware, or processors such as the PowerPC, hardware designers will be required to implement specialized decode and mapping circuitry to map peripherals into memory space. Additionally, there may be requirements to change the Linux kernel MMU page tables and mappings to allow addressing of these memory regions.

For example, if the hardware designer designs a PC-compliant motherboard, yet places specialized peripherals at 0xA0000000, the MMU tables will need to be updated to allow access to that physical location. Since Linux is distributed with source code, this is a relatively minor issue. It's just important that the developer be aware of it.

Another approach, when using a PC-style platform, is to map peripherals into I/O space. I/O space is for peripherals that respond to special bus cycles from the processor. These cycles are generated through special-purpose assembly language instructions. This article is based on a hardware environment that resides in I/O space.

Figure 3 provides a block diagram of the hardware and configuration used for the examples in this article. This hardware test bed consists of a pushbutton that is debounced through a 74LS221 Monostable Multivibrator on a breadboard, and an off-the-shelf counter/timer board (CIO-CTR05) from Computer Boards (*www.computerboards.com*). The CIO-CTR05 contains an AMD 9513 counter/timer device, which has five 16-bit counter/timers. Additionally, the CIO-CTR05 provides eight digital inputs and eight digital outputs, as

Put Linux Anywhere™

No, this isn't an ad for a cruise line. We just thought you might find it interesting to see some of the places embedded Linux can be used. Because Linux is open source, it offers unmatched control, flexibility and reliability that allow it to precisely match your unique system requirements and constraints. And as a leader in the real-time embedded Linux market, Lineo provides OEM tools, support and services to help you build solutions across the full spectrum of embedded systems. So when you partner with Lineo you can put Linux anywhere, even in an ad for a cruise line.

Visit us at www.lineo.com or call 1.801.426.5001

LISTING 2 CIO-CTR05 module initialization

```
static struct file_operations ctr05_fops = {

        NULL,               /* ctr05_seek      */
        ctr05_read,         /* ctr05_read      */
        ctr05_write,        /* ctr05_write     */
        NULL,               /* ctr05_readdir   */
        NULL,               /* ctr05_select    */
        ctr05_ioctl,        /* ctr05_ioctl     */
        NULL,
        ctr05_open,
        NULL,
        ctr05_close,
        NULL                /* fsync            */
};

int init_module( void ) {
int     err;
BYTE    bReg;

    /* Register as a device with kernel.
     */
    err = register_chrdev( MajorNumber, "ctr05", &ctr05_fops );

    if ( check_region( CTR05_BASE, BOARD_SIZE ) != 0 )
    {
        unregister_chrdev(MajorNumber, "ctr05") != 0);
        return( -ENODEV );

    } else
    {
        request_region( CTR05_BASE, BOARD_SIZE, "ctr05" );
    }

    /* Register interrupt handler
     */
    request_irq( CTR05_IRQ, ctr05_interrupt, SA_INTERRUPT, "ctr05", NULL) );

    /* Initialize the 9513 chip; Master Reset, Select
     * Master Mode Register
     */
    outb_p( MASTER_RESET, CMD_REG );
    outb_p( MASTER_MODE_REG, CMD_REG );

    /* Write lower byte of MMR, followed by upper
     */
    bReg = 0x70;
    outb_p( bReg, DATA_REG );
    bReg = 0x00;
    outb_p(bReg, DATA_REG);

    return 0;
}
```

well as the ability to assert an interrupt to the system.

In my experiments, I used one of the 9513's 16-bit counters and the logic on the CIO-CTR05 to generate an interrupt. When the pushbutton is depressed, the Monostable generates a 300ms pulse. This pulse, on its active edge, asserts an interrupt to the processor and starts the timer counting. The counter is clocked at 1MHz.

A Linux device driver services the interrupt. When its interrupt handler is called—from the Linux kernel's "low-level interrupt handler"—it notes the value in the counter and then potentially alerts application code that the interrupt has been asserted. If the application code has provided an "interrupt" handler, that handler will execute in process space and perform a read of the device driver to stop the counter and read it's value. Through this scheme, the following is accomplished:

- A driver is implemented to support the interrupt and counter, and provides a means to alert application level code
- Instrumentation is put in place to characterize and provide timing analysis of interrupt operation in Linux

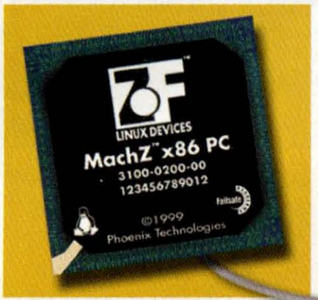
Driver basics

Linux device driver constructs and implementation details are beyond the scope of this article. This information may be found in books such as *Linux Device Drivers* by Alessandro Rubini (O'Reilly & Associates, 1998).

As a foundation for this article, a driver for the CIO-CTR05 was located at a site at North Carolina State University (*ftp://lx10.tx.ncsu.edu/pub/Linux/drivers*). This driver provided me with a head start for putting the test fixture in place. (One of the great things about working with Linux is the tremendous amount of source code that's already out there to supplement it.)

Finally

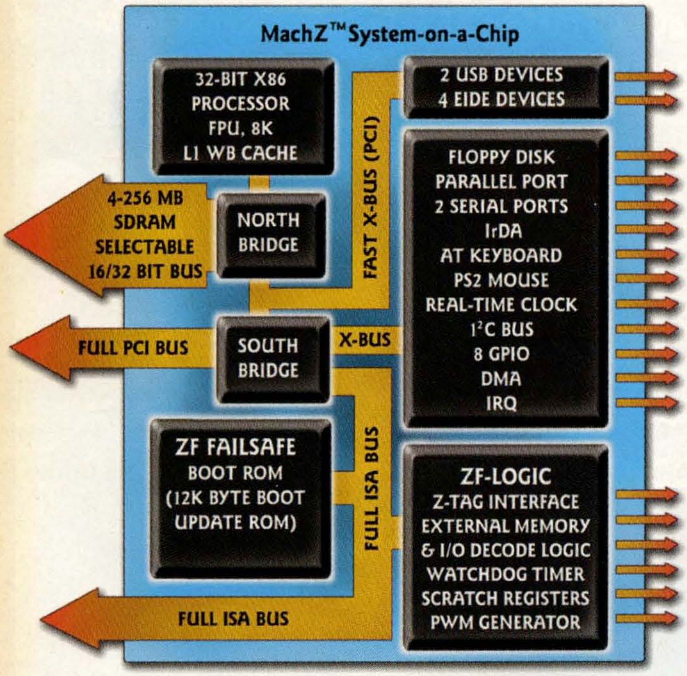
a *real* x86 System on a Chip



(actual size)

Introducing the "FailSafe MachZ" for the next generation of embedded systems

You can design the MachZ system into virtually anywhere your imagination can take you. It's very small and at a 1/2 watt it actually runs cool to the touch. Think about it: you could embed this little power house in a system that is completely sealed. Consider the possibilities, consider the facts. Then put your engineering brain to work.



MachZ™ System-on-a-Chip

Ultra Low Power <500mW@100MHZ

Auto-Boot FailSafe System allows crash-immune Internet upgrades - FailSafe Boot, Flashless Operation, Dual WD-Timer, Z-tag 2M-bit/s Flash download, Scratch Registers

Fully PC Compatible runs all x86 code natively

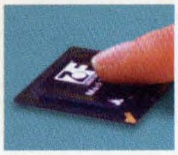
Easily Interface HomePNA, Ethernet, Modems, Graphics Controllers, PC/104, MPEG, XDSL, ISDN, IrDA, Flash, CAN, GPS

Includes BIOS and Linux O/S or WindRiver® VxWorks RTOS

phoenix™ technologies

Oh, and did we mention that it runs cold?

Failsafe EMBEDDED

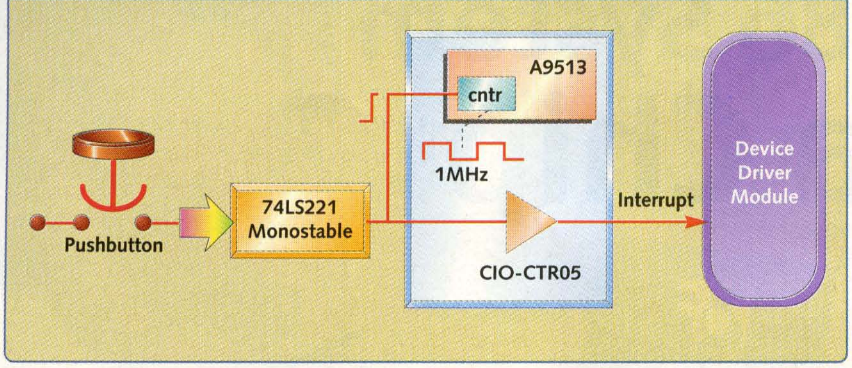


388 BGA Runs at a cool 1/2 Watt not a typical 8 to 10 Watts

ZF LINUX™ DEVICES

800.683.5943 • www.zflinux.com

FIGURE 3 Test fixture configuration



The driver from NCSU is fairly robust, supporting many of the functions of the CIO-CTR05 board. In order to present the concepts within the bounds of this article, the basic driver was scaled down to support only the topics discussed within. As each aspect of the driver is discussed, working source code is presented to support the discussed concepts. It is also important to note that several types of drivers exist in the Linux environment character, block, and network. The driver discussed in this article is a character device driver.

Linux device drivers are considered "modules," and these modules may be loaded dynamically at runtime. This means that the developer is able to implement, compile, load, test, and then unload the driver. This cycle may be repeated over and over without rebooting the Linux machine, providing a suitable environment for developing a driver. The list of installed modules can be obtained with the `lsmod` command. This command displays what modules are loaded, and what components in the system are using each module. Listing 1 provides the output from `lsmod` in the system on which the driver in this article was developed.

The module `tulip` is the device driver for a PCI network card, while the module `ctr05` is the driver for the CIO-CTR05. The module `ctr05` was installed with the following command:

```
/sbin/insmod -f ctr05.o
```

This module can be removed during

Small Footprint... Huge Impact!

Extending data beyond the corporate enterprise. PointBase, the first pure Java data management solution designed to support applications for the mobile wireless device market.

PointBase

*Managing data **anywhere, anytime.***

Meet the demands of the small application environment... test-drive the PointBase data management solution with synchronization...discover the keys to high-power, small-footprint databases...pick-up your FREE 30-day evaluation copy now!

www.pointbase.com/esp

Questions? Give us a call
1.877.238.8798 toll free

LISTING 3 CIO-CTR05 module removal

```
void cleanup_module(void)
{
    if ( MOD_IN_USE )
    {
        printk("%s: device busy, remove delayed.\n", ADAPTER_ID);
        return;
    }

    release_region( CTR05_BASE , BOARD_SIZE );

    if ( CTR05_IRQ >= 2 && CTR05_IRQ <= 7 )
    {
        free_irq( CTR05_IRQ, NULL );
    }

    if ( unregister_chrdev( MajorNumber, "ctr05" ) != 0 )
    {
        printk("%s: cleanup_module failed.\n", ADAPTER_ID);
    }
}
```

LISTING 4 CIO-CTR05 device files

```
crw-r—r—    1 root     root      50,   0 May 18 08:23 /dev/ctr05_DIO
crw-r—r—    1 root     root      50,   1 May 18 08:23 /dev/ctr05_CTR1
```

runtime with the following command:

```
/sbin/rmod ctr05
```

Every device driver module has two key functions:

- `init_module()`—Responsible for installing the driver in the kernel; called by the kernel when the module is loaded
- `cleanup_module()`—Responsible for doing any hardware shutdown or internal cleanup when a module is removed from the kernel

The procedure `init_module()` registers the driver with the kernel, telling the kernel about other internal functions such as `read()`, `write()`, or `ioctl()`. Additionally, if the driver supports interrupts, it requests that the kernel call an internal function when the specified interrupt is asserted. Finally, this procedure is responsible for configuring and initializing the hardware. Listing 2 shows the initialization function for the CIO-CTR05. Note that to keep the listing short for this article, most of the error checking has been removed. However, this listing does present the basics of the initialization of the module. The following key steps are performed:

1. Registering the module as a character device driver. After this, application code running in user space may perform `open()`, `close()`, `read()`, `write()`, or `ioctl()` operations on the module
2. Checking for and requesting a "region." This allows the driver to communicate with the physical hardware device (which is, in this case, the CIO-CTR05 board)
3. Registering an interrupt handler with the kernel. This operation associates a physical interrupt level with a module-based handler
4. Configuring the hardware

Removing a module from the kernel requires that the reverse be done: release the region, release the interrupt, and un-register the device.

Listing 3 shows the procedure `cleanup_module()` used in the CIO-CTR05 driver. When reviewing this listing, note the function `printk()`. This function is identical to `printf()`, with the exception that it operates at the kernel level. Diagnostic messages generated with `printk()` are sent to the Linux console. For all practical purposes, this is the most useful debug tool for implementing Linux drivers and interrupt handlers.

Application code

To this point, we have discussed the concept and basics of a Linux device driver. The important points to note are as follows:

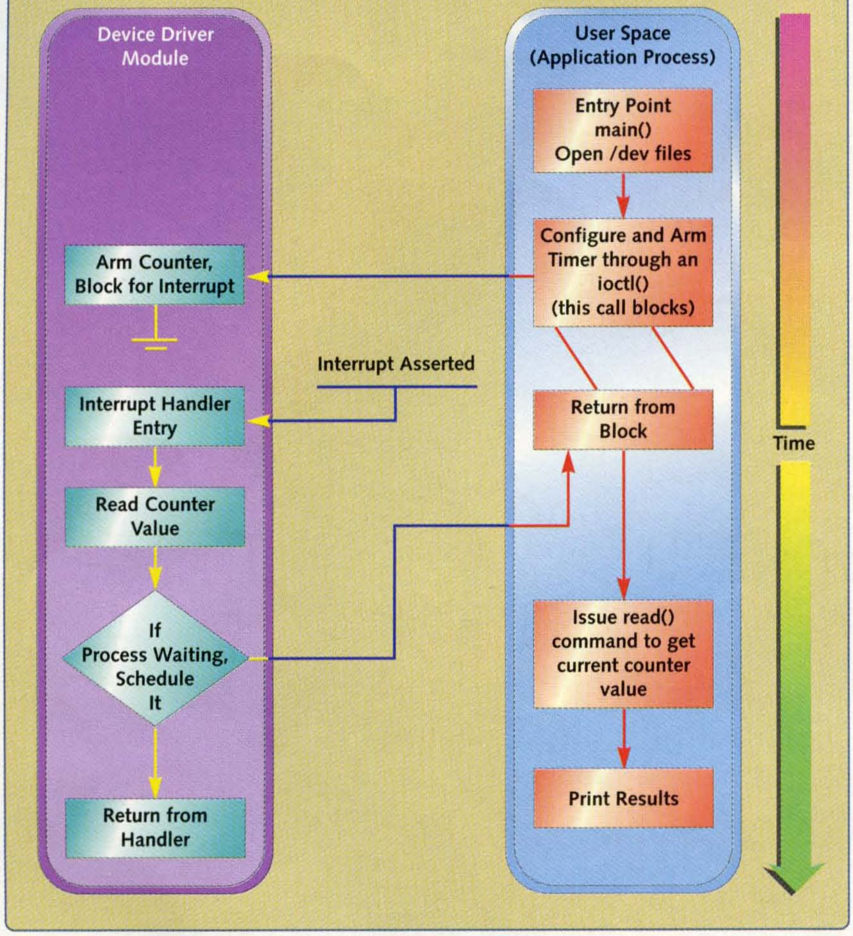
- Interrupts must be serviced by a driver
- Hardware may only be communicated with through a driver

Specific operation on an interrupt will be discussed in a bit. First, we need to understand how application-level (process) code running in user-mode interacts with the kernel-mode driver.

Device drivers look like files to application code. Their primary interface is through special files called device files, which are generally located in the `/dev` directory in the Linux file system. Device drivers have associated with them major and minor device numbers. The major device number is used to associate a device file with a device driver (module).

The Future
Is Ahead
of Schedule

FIGURE 4 Sequence of events


Device Driver Module
User Space (Application Process)
Entry Point main() Open /dev files
Configure and Arm Timer through an ioctl() (this call blocks)
Arm Counter, Block for Interrupt
Interrupt Asserted
Interrupt Handler Entry
Return from Block
Time
Read Counter Value
If Process Waiting, Schedule It
Issue read() command to get current counter value
Print Results
Return from Handler

LISTING 5 CIO-CTR05 module interrupt handler

```
static void ctr05_interrupt(int irq)
{
  int data;

    data = read_counter( 1 );
    intCntValue = data;

    if ( waitqueue_active( &int_wq ) )
    {
        wake_up_interruptible( &int_wq );
    }
    else
    {
        printk("Wait Queue is NOT ACTIVE!!\n" );
    }
}
```

Listing 2 shows that when the module is registered with the kernel, it passes in a major device number.

Device files in the /dev directory are created by the module developer using the Linux command mknod. Listing 4 shows the device files for the CIO-CTR5 module. This listing was made by issuing the command ls -rtl /dev/ctr*.

The first field indicates that the module is a character device (c) that can be both read and written (rw-) by users or processes running as root, but only read (r--) by other users. The fifth field, 50 in this example, is the major device number, while the sixth field, 0 and 1 for these device files, is the minor device number. Minor numbers may be selected in any way that makes sense to the driver developer. In the CIO-CTR05 driver, they represent and specify one of the five counters in the AMD 9513 contained on the I/O board.

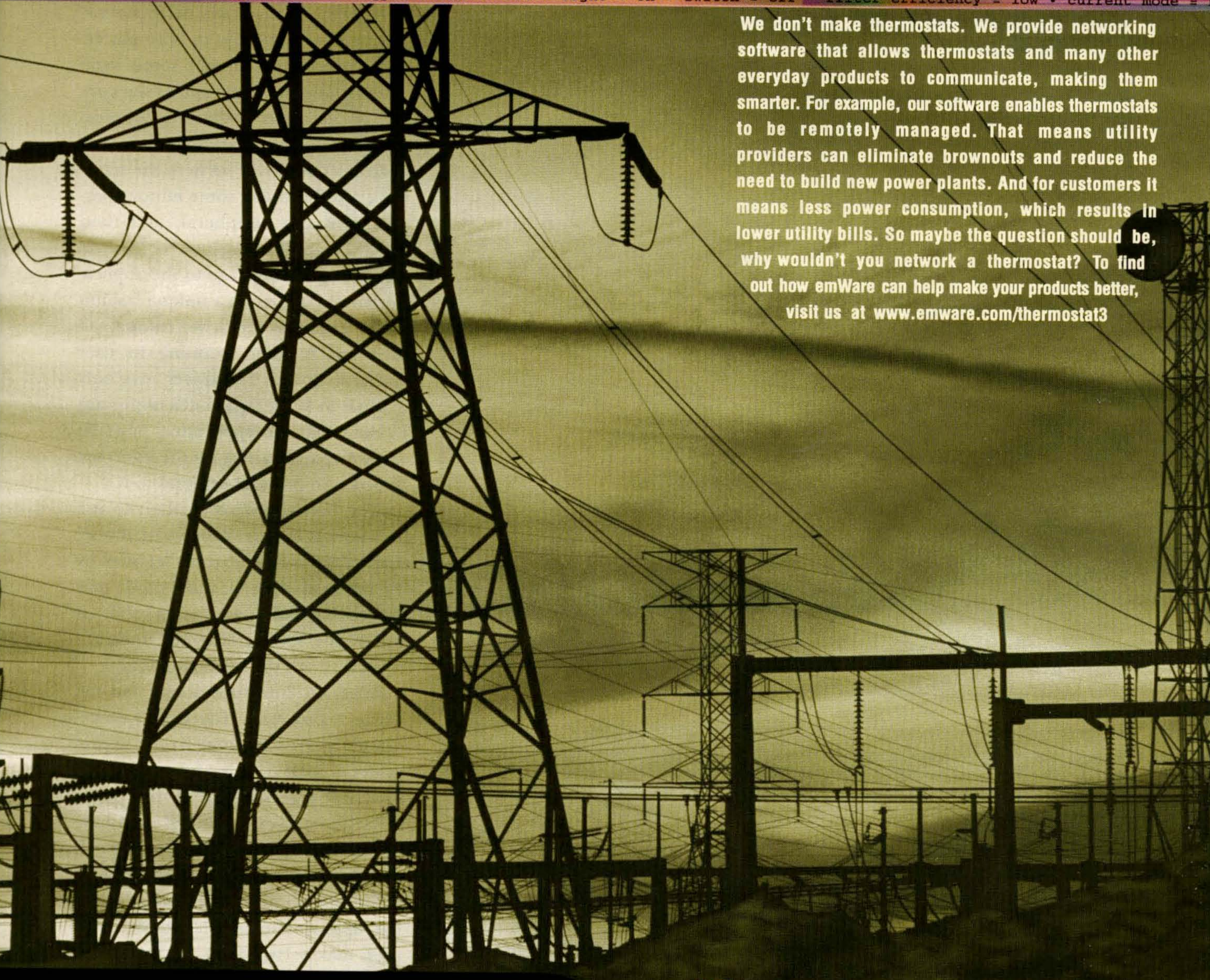
Finally, /dev/ctr05_DIO and /dev/ctr05_CTR1 are device file names. These names generally represent something meaningful to the developer. In this case, ctr05 identifies the module, while DIO states that this file is for operating on the digital I/O portion of the driver, and CTR1 is for counter 1 control. These device files were created by executing the following commands:

```
/bin/mknod /dev/ctr05_DIO c 50 0
/bin/mknod /dev/ctr05_CTR1 c 50 1
```

Application code operates on these by first performing either an open() or fopen() on the device file. After that, the returned file descriptor or stream pointer may be used to read(), write(), ioctl(), or close() the specified device (module). Internal to the module are procedures to support each of these functions. These procedures behave in a consistent way through standardized calling conventions, but internally perform operations specific to the peripheral.

why would you want to network a thermostat?

ure = 72 degrees • power consumption = 4 kilowatts per hour • regulate = auto • high set point temp = 72 degrees • low set point
r drop = 4.2 degrees • fan = low • energy discount = 18% • light = on • switch = off • filter efficiency = low • current mode =



We don't make thermostats. We provide networking software that allows thermostats and many other everyday products to communicate, making them smarter. For example, our software enables thermostats to be remotely managed. That means utility providers can eliminate brownouts and reduce the need to build new power plants. And for customers it means less power consumption, which results in lower utility bills. So maybe the question should be, why wouldn't you network a thermostat? To find out how emWare can help make your products better, visit us at www.emware.com/thermostat3

to conserve energy.

emWare®
ANY DEVICE, ANY NETWORK™

eti
ALLIANCE™

e-smart™

device networking software that makes products better

go to www.emware.com/thermostat3 or call toll-free 1.877.436.9273

LISTING 6 CIO-CTR05 module ioctl() function

```
static int ctr05_ioctl(struct inode *iNode, struct file *filePtr,
    unsigned int cmd, LONG arg)
{
int size = _IOC_SIZE(cmd);
int err = 0;

    if (_IOC_TYPE(cmd) != IOCTL_MAGIC) return -EINVAL;
    if (_IOC_NR(cmd) > IOCTL_MAXNR)  return -EINVAL;

    if (_IOC_DIR(cmd) & _IOC_READ)
    {
        err = verify_area(VERIFY_WRITE, (void *)arg, size);
    }
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
    {
        err = verify_area(VERIFY_READ, (void *)arg, size);
    }
    if( err ) return( err );

    if ( cmd ==  WAIT_FOR_INTERRUPT )
    {
        arm_counter( 1, 11 );
        interruptible_sleep_on( &int_wq );
        return( 0 );
    }
    else
    {
        return( -EINVAL );
    }
}
```

LISTING 7 Application process for interrupt

```
void waitForInt()
{
unsigned int allData;

  printf( "waitForInt(): called, doing ioctl()\n" );
  ioctl( fd_ioctl, WAIT_FOR_INTERRUPT );
  read( fd_read, &allData, 1 );
  printf( "waitForInt(): Intererupt count = %d\n", (allData & 0x0000FFFF) );
  printf( "waitForInt(): Application count= %d\n",
                ((allData & 0xFFFF0000) >> 16) );
}
```

Interrupt delivery

When an interrupt is asserted, the developer has two choices for processing it:

- In the driver, at kernel level
- In user space, in the application process or processes

Most applications will perform processing of an interrupt in both places. The driver must contain some logic for handling the interrupt, as the context of the driver is the only location that the interrupt handling code may execute. The driver services the interrupt, possibly doing some minor operations on the peripheral, and then alerts the application.

Getting the message to the application must be done in a unique fashion under Linux, as the driver cannot call interrupt handlers running in user (process) space. The driver may only execute in kernel space. It's important to note that the application must be expecting and waiting for the interrupt to occur, in order to respond to it.

The best approach is often to use an *interrupt dispatch* process or thread that waits for all interrupts serviced by the driver module (as the module may service hardware devices capable of generating several different interrupt types through the same IRQ line).

When an interrupt occurs and the dispatch process is placed into execution, it sends a message to the appropriate process or thread, based on the interrupt type serviced by the driver ISR. The process or thread then handles the interrupt, while the dispatch process waits for the next interrupt. In practical operation, this process opens the appropriate module device file, then makes (typically) an `ioctl()` or `read()` call that "blocks," suspending the caller and allowing other processes and threads to run while the "dispatch" process waits for the interrupt.

Figure 4 shows the sequence of events for the following discussions. Each block in this figure is described in detail in subsequent paragraphs.

Vision available. Hardware not included.

ESPIAL™

BUILD A SMART FUTURE™

espial.com

LISTING 8 open() on /dev devices

```
void DoOpenDevices()
{
    fd_ioctl = open( "/dev/ctr05_DIO", 0666 );
    if ( fd_ioctl < 1 )
    {
        perror( "can't open DIO" );
        exit(2);
    }

    fd_read = open( "/dev/ctr05_CTR1", 0666 );
    if ( fd_ioctl < 1 )
    {
        perror( "can't open counter" );
        exit(2);
    }
}
```

LISTING 9 CIO-CTR05 read() function

```
static ssize_t ctr05_read(struct file *filePtr, char *buf,
                 size_t count, loff_t *off)
{
unsigned int  allData;
int             minor;
WORD            data;

    struct inode *iNode = filePtr->f_dentry->d_inode;
    minor = MINOR( iNode->i_rdev );

    data = read_counter( minor );
    allData   = ((data << 16) | intCntValue);
    put_user(allData, (unsigned int *) buf);

    return 0;
}
```

The interrupt handler in the driver module executes when an interrupt is asserted. It does minor processing of the interrupt, then checks to see if an application process is waiting for that interrupt. If one is waiting, the interrupt handler alerts the process, and the Linux scheduler places it into execution based on its priority within the system.

Listing 5 shows the interrupt handler used in the CIO-CTR05 device driver module. It is extremely important to note that this interrupt handler executes in the context of the kernel.

This interrupt handler performs the following functions:

- Reads the counter value (the counter started counting at the moment the interrupt was asserted to the processor)
- Makes a call to kernel function `waitqueue_active()`. This function checks to see if an application process has blocked on a user-specified Wait Queue called `int_wq`
- If a process is blocked on the Wait Queue, waiting for the interrupt, a call is made to `wake_up_interruptible()`. This function unblocks the process, and the kernel will place it into execution based on its specific scheduling rules (regular Linux process vs. real-time Linux process)

Listing 6 shows the CIO-CTR05 module logic (which allows user processes to block) waiting for the interrupt. This logic is implemented as an `ioctl()` operation in the driver. Like the interrupt handler, this procedure executes in the context of the Linux kernel.

The core logic in Listing 6 follows the `if` statement to see if the `ioctl()` operation specified by `cmd` is `WAIT_FOR_INTERRUPT`. In this case, the following occurs:

1. The counter is armed and its value is reset to zero, and it is configured to wait for the edge transition from the monostable, which occurs when the pushbutton is depressed. When this edge transition occurs, the counter begins counting at a 1MHz rate
2. The counter makes a call to `interruptible_sleep_on()`, a kernel function that blocks the calling process until `wake_up_interruptible()` is called

After returning from `wake_up_interruptible()`, the `ioctl()` call returns to the user's process, resuming execution in the user's process at the location following the `ioctl()` call. Listing 7 shows the application process that makes the `ioctl()` call. This code fragment executes in the context of Linux user space.

In the function `waitForInt()`, the `ioctl()` function is called, which blocks the caller until the interrupt occurs. Upon return (interrupt asserted), a `read()` is done on the module to get the current value of the counter. This integer data contains two fields; the counter's value at the time the interrupt handler executed, and the value at the time the `read()` function was called.

The file descriptors used for the `ioctl()` and `read()` calls were

You Need **TimeSys Linux/RT™** The Power of Real-Time in Pure Linux™

Why TimeSys Linux/RT?

TimeSys Linux/RT is the first complete and open-source solution that operates directly within the Linux kernel to provide predictable real-time response entirely from the Linux OS.

TimeSys Linux/RT vs. Standard RTOS		
	TimeSys Linux/RT	**Standard RTOS** (average)
Run-Time Licenses	$0*	Expensive ($.10 to $50)
Developers Exposed to API	100,000 +	5,000 to 50,000
Device Drivers	50,000 +	5 to 5,000
Applications and Utilities	100,000 +	Limited
Source Code License	$0*	>$200,000 per seat

TimeSys Linux/RT vs. Standard Linux		
	TimeSys Linux/RT	**Standard Linux**
Footprint	75KB to 1.2MB	1 to 2MB
Scheduler	Fine-grained (256 priority levels) + enforced CPU reservations	Simple fixed priority
Timer Resolution	Very high (10µs or less)	Very low (10ms or higher)
Tools for Embedded/RT Support	Comprehensive (TimeWiz, TimeTrace TimeBench, TimeWarp)	Limited
Real-Time POSIX Extensions	Yes	No

Features:

- Embedded, diskless, headless operation
- Support for x86, PowerPC, ARM/StrongARM and (soon) MIPS
- A large variety of BSP's is available
- Complete set of development tools: TimeTrace™, TimeWarp™, TimeWiz® and TimeBench™
- Support for real-time extensions to POSIX
- 256 fixed-priority levels
- Support for priority inheritance to avoid problems from priority inversion
- High-resolution timers (1ms or less)
- CPU reservations for QoS guarantees for real-time and non-real-time applications
- Support for periodic processes
- RTOS abstraction layers (pSOS)

Benefits:

- Open source
- $0 run-time royalties*
- Use any and all Linux applications, utilities and tools
- Use Linux device drivers as is**
- Satisfy embedded, real-time and/or QoS requirements
- Spatial and temporal partitioning to isolate application groups from one another
- From the leading provider of complete embedded real-time solutions.

TimeSys Linux/RT™

*for base platforms and capabilities
**small number of exceptions may apply

TimeSys Linux/RT Configurations

TimeSys Linux/RT supports two configurations to provide you with a complete range of options for your embedded and/or real-time system.

Configuration A upgrades the core Linux kernel with the Resource Kernel (RK), adding real-time capabilities in Linux.

Pros:

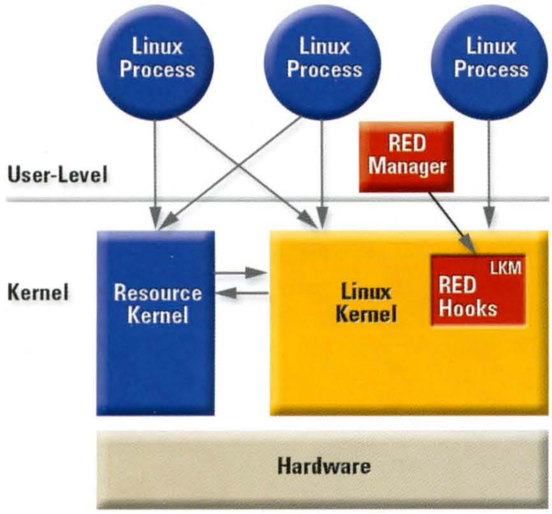
- Embedding support
- Real-time POSIX extensions
- QoS guarantees
- Use Linux drivers as is**
- Use all Linux applications, utilities and compilers
- Support for Real-Time Java™
- Excellent timer resolution
- Open source

Cons:

- Performance not as high as Configuration B

TimeSys Linux/RT Configuration A

The Power of Real-Time in Pure Linux



Configuration B adds a thin kernel below the Linux kernel providing high performance alongside Linux.

Pros:

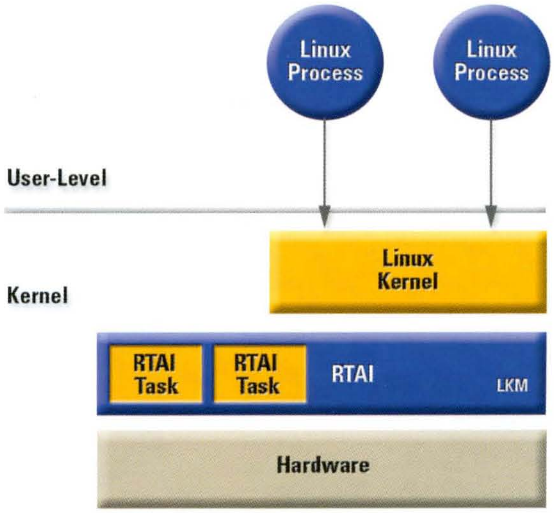
- Very high performance
- Smaller footprint
- Open source

Cons:

- Task failure leads to system crash
- Need custom device drivers
- Custom API for applications

TimeSys Linux/RT Configuration B

Real-Time Alongside Linux



Get the Power

If you are building systems with embedded, real-time and/or Quality of Service requirements, you need the power of TimeSys Linux/RT. To order, or for more information, call us at 888.432.TIME or visit us at www.timesys.com.

TimeSys Corporation

Real-Time... Real Solutions.™

TimeSys Provides the Complete Embedded Real-Time Solution

TimeSys Linux/RT™

Open and powerful embedded real-time OS with QoS features

JTime™ (1)

Embedded Real-Time Java Virtual Machine compliant with Sun's Real-Time Specification for Java

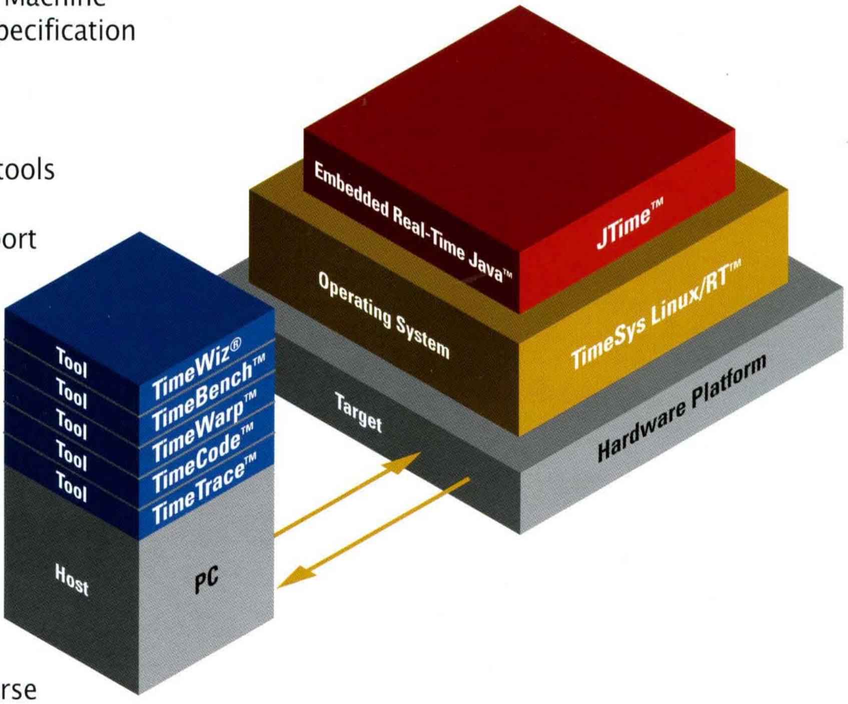
SuiteTime®

Suite of integrated development tools

- **TimeTrace™**
  Profiling and visualization support
- **TimeWarp™** (1)
  Integrated development environment
- **TimeCode™** (1)
  Application profiling, charting and visualizing tool
- **TimeWiz®**
  Modeling, analysis and simulation for embedded real-time systems
- **TimeBench™** (1)
  Real-Time UML tool for automated generation and reverse engineering for embedded software

Professional Services

We also provide consulting, training, customization, porting services and turnkey project development. Tell us how we can help you.



[1] *Available to select customers for evaluation*

TimeSys Corporation
Corporate Headquarters: 4516 Henry Street ▪ Pittsburgh, PA 15213
888.432.TIME ▪ 412.681.6899 ▪ Fax: 412.681.5522
www.timesys.com ▪ info@timesys.com

acquired from a standard `open()` call. Listing 8 shows the application code that opened these descriptors.

Listing 9 shows the `read()` function as implemented in the CIO-CTR05 module. Like the `ioctl()` and interrupt handler fragments, this fragment executes in the context of the Linux kernel.

The function `ctr05_read()` performs the following functions:

- Reads the current value of the counter through the function `read_counter(minor)`. The minor number is 1 in this case from the device file `/dev/ctr05_CTR1`, indicating Counter 1
- Combines the data with the counter value saved when the interrupt handler executed
- Copies the data into user process space (isolated from kernel and other process space by the MMU) and returns

It is important to note that this call does not block, and returns immediately to the calling application process after transferring the data into the user buffer identified as `buf`.

TABLE 1 Timing data, no load

Assertion #	Count at ISR execution	Count from read()
1	10	154
2	9	50
3	9	55
4	8	54
5	9	64

TABLE 2 Timing data, system loaded

Assertion #	Count at ISR execution	Count from read()
1	226	17404
2	624	705
3	21793	47086
4	18	1269
5	14	152

Interrupt characterization

The test fixture presented in this article provides for basic characterization of response times within Linux. The examples were tested on a 100MHz Pentium processor with 80MB of RAM. Table 1 shows the response times for five back-to-back interrupts (initiated through the pushbutton) with no load on the system. The numbers represent the number of ticks from the time the interrupt was asserted to the processor. The ticks increment at 1MHz, representing time in microseconds.

Listing 10 shows a simple shell script that was used to place a load on the processor.

The command:

```
ping -f  tbcorp1 &
```

causes the local machine to ping the machine `tbcorp1` as quickly as the network interface is able to send and receive packets. In the test environment, this causes heavy PCI activity, heavy interrupt activity, and mild CPU loading.

The "while" loop `tars` up `/usr` into a file, then removes it, over and over. The `tar` command operates verbose, which causes a considerable amount of text information to be sent to the console. Additionally, the `tar` command itself causes lots of disk activity, which causes interrupts and heavy traffic on the bus.

Table 2 shows the results of five back-to-back interrupts through the test fixture while this load executed. The lack of consistency is notable. With some interrupts, such as Assertion 5, the numbers are almost as low as when the tests were run without load. In other cases, such as Assertion 1, interrupt latency was not terrible, but the amount of time before the application process executed was lengthy. The worst case measurement is Assertion 3, which shows long interrupt latency and long latency to process execution.

These numbers provide some starting points for analysis. These were taken using standard Red Hat Linux, with no real-time extensions in place. The application process was a regular Linux process vs. a real-time Linux process. (Real-time Linux processes execute at a higher priority than any other process. Because of this, interrupt latency is an issue, whereas latency time in process execution is not.) Additionally, a real-time Linux project is under way which solves interrupt off-time problems, and has improved scheduling.

LISTING 10 Processor loading script

```
# !/bin/sh
#
#  Simple Processor Loading Script

ping -f tbcorp1 &

while [ true  ]
do
     tar cvf tmp.tar /usr; rm tmp.tar
done
```

Regardless of what version or flavor of Linux you use, if you have hardware devices that your application code must communicate with, and these devices have interrupts, you will need to design, implement, and install a driver module as discussed in this article. **esp**

Thomas Besemer has been developing embedded software since 1980, using a variety of tools, processors, environments, and approaches. He is the president of a consulting firm, a supporter of public radio, and a photographer, who spends as much time as he can enjoying life with his 4-year-old son. Contact him at tbesemer@tbcorp.com.

WE CALM THE SEAS OF TECHNOLOGY

SO YOU
CAN RIDE
THE WAVE
OF SUCCESS

WHY PACIFIC SOFTWORKS ?

• **Protocols:**	TCP/IP, UDP/IP, DHCP, BOOTP
• **Applications:**	FTP, TFTP, TELNET, DNS, SMTP, POP3, RPC
• **Internet:**	PPP, SLIP, CSLIP, CHAP, PAP
• **Web:**	Web Micro Browser, Web Server, Graphics Toolkit/GUI
• **Network Management:**	SNMP v1/v2/v3, MIB Compiler
• **Routing:**	RIP/RIP2, OSPF, BGP4, NAT
• **Security:**	SSL Support
• **RTOS:**	ThreadX™

Since 1982, Pacific Softworks has been delivering the quickest and most stable Internet Protocols available in the industry. We're proud of our reputation as the leading supplier of protocol software.

Royalty Free or Royalty based licensing
End-to-End support from the Protocol Specialists
Ported and Tested Compatibility with any Processor, Kernel or RTOS
Flexible and Scalable solutions to protect your investment
Demonstrated reduction of time to market
Leading the industry with next generation products and solutions
World-wide presence
...and, most importantly, we have Happy Customers!

Visit our website for more in-depth information on Fusion, the "off-the-shelf" FastTrack and other leading embedded solutions. www.pacificsw.com , e-mail: sales@pacificsw.com

Pacific Softworks provides a single supplier solution ensuring that today's successes are the building blocks for tomorrow.

California Corporate Headquarters:
tel: 800.541.9508 fax: 805.499.5512
European Headquarters:
tel: 44.1494.432.735 fax: 44.1494.432.728
Japan/Asia-Pacific Headquarters:
tel: 81.3.5669.7722 fax: 81.3.5669.7723

Pacific Softworks
our wave... your Success.

YASHVANT JANI

Network Talk: Voice Over IP

Voice over IP (VoIP) has a big place in the future of the Internet. This article tells you what software is required to support VoIP and how the system should be architected.

In traditional telephony applications, speech is digitized and delivered over a circuit-switched network. During call setup, a dedicated amount of bandwidth is reserved for each phone call. In a VoIP system, voice signals are transported as packets over a packet network with no bandwidth dedicated to their delivery. The processing involved in a VoIP system is, therefore, quite different from the circuit switched delivery operation.

This article looks at a typical VoIP system architecture and describes the various software modules that must be included. We will also look at estimates of code size and required processing power for each module, as well as some guidelines for partitioning the system between a traditional CPU and a DSP.

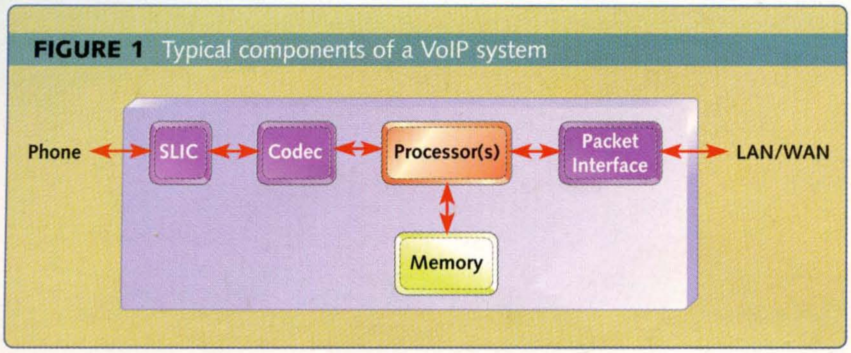
VoIP technology offers several advantages in the telecommunications market:

- Effective bandwidth utilization, because the same channel can be used for voice packets going to different destinations
- A lower bit rate per voice channel through vocoders. For example, G.729a results in a sub-15Kbps rate (including overhead) versus a 64Kbps A/µ-law based voice bit stream
- Simultaneous voice/data/fax transmission
- Reduced costs for long distance voice calls and facsimile transmissions

Architectural considerations

A typical VoIP system has two interfaces to the outside world, as shown in Figure 1. The first interface is to a telephone or handset and the second to a packet net-

FIGURE 1 Typical components of a VoIP system



work (LAN or WAN). The telephone interface consists of a subscriber line interface card (SLIC) and a coder/decoder (codec). The SLIC handles the interaction with the handset, while the codec performs the analog-to-digital and digital-to-analog conversion of the voice signals. The packet interface can either be a WAN interface, such as ISDN, Packet Cable, or DSL, or a LAN interface, such as Ethernet.

Before sending voice data as packets, the analog signal from the handset microphone is sampled at 8kHz to generate a PCM-encoded digital data stream. Depending on the codec hardware setting used, the resulting bit stream has a bandwidth of either 64Kbps or 128Kbps. The digital signal is then processed by an echo canceller to remove the echo of the far-end received signal. This echo-free bit stream is then converted into frames of constant length by collecting samples for a definite time period, say 10ms. The frames are passed to the application layer, where a speech coder (also known as a Vocoder), such as G.729a, compresses the data. With sufficient digital signal processing, a 64Kbps bit stream can be compressed into a 6.4Kbps stream.

The TCP/IP protocol stack is then invoked to create the actual IP packets that will travel across the packet network. The compressed voice frame becomes the payload, with appropriate headers added at the real-time protocol (RTP), UDP, and IP layers. Once the packets are created, they can be transported over any suitable WAN or LAN data link and physical layers—Ethernet, for example.

On the receiving side, voice packets arriving at the network interface are depacketized, decompressed, and then played out at the handset speaker via the same codec/SLIC interface. This entire sequence of operations must be executed in real time for one or more voice channels, depending on the features of the product.

A VoIP system requires real-time processing at two points: when the samples are collected and when they are played back. If the sampling or playback period varies even slightly, the listener will hear something other than a continuous voice signal. To reduce the load on the CPU, a buffer is used for collecting the samples and transferring them into frames. A typical frame size is 80 samples (10ms of data). This frame must be compressed and converted into an IP packet during the 10ms before the next frame is created. Because voice packets are being received at the same time, those must be depacketized and decompressed for playback within the same time slot. Thus, both DSP and CPU functions must take less than 10ms for a complete cycle of compression/decompression and packetization/depacketization for one voice channel. When multiple voice channels are provided, the system becomes even more constrained. For example, a product with two voice channels must complete each set of tasks within 5ms maximum.

Since packet networks are asynchronous and the network bandwidth might be higher than the voice collection rate, a buffer must exist to store the received packets. These packets also need to be reordered, since they may arrive at different times, as the result of network congestion. Multiple voice channels require multiple buffers, making buffer management necessary.

Software components

The overall processing needed for VoIP is a combination of signal processing and protocol-based control processing. Tasks such as speech compression and decompression, echo cancellation, and tone detection require signal processing capability, while network packetization and depacketization and call setup and teardown tasks require protocol-based control processing capability. The required processing capability can be realized with two different processors, a DSP for the signal processing tasks and a traditional CPU for protocol implementation. A hybrid CPU/DSP processor can also be used.

VoIP systems require simultaneous execution of at least the following software modules:

- Collection of digitized voice samples, as well as playback at regular intervals
- Call setup/teardown and call control using H.323 or H.245 standards
- Vocoder algorithm such as G.723, G.729 and G.729a, G.711, or G.728 (interoperability with other VoIP systems may require availability of many vocoders in one system)
- Telephony support modules such as DTMF, call progress tones, and line and acoustic echo cancellation
- TCP/IP protocol stack
- LAN/WAN interface driver
- RTOS that provides services such as interrupts, timers, and buffer management

A variety of tones are used on the public switched telephone network for

TABLE 1 Popular VoIP vocoders with bit rate and mean opinion score

Vocoder	Bit rate (Kbps)	Score (MOS)	Sample size (ms)	MIPS
G.711	64	4.4	0.75	<1
G.723.1	6.3/5.3	3.6	30	30
G.729	8	3.9	10	30
G.729a	8	4.0	10	17

dialing-dual tone multi-frequency (DTMF), call setup/tear-down, and caller ID. Tones are also used to indicate the call status and line conditions. A VoIP system must recognize and perform these tones to support a phone call. A VoIP system also needs to emulate central office functions such as the generation of call-progress tones used for conveying conditions such as ringing, busy, and dialing. The ITU standards Q.23/Q.24 specify the DTMF tones, while ITU standard E.180 specifies call progress tones for North America.

Echo cancellers are required to cancel echoes that arise as the result of impedance mismatches at the hybrid networks that connect two-wire links to four-wire links. Echo is a problem in VoIP networks because the round-trip delay is significant (longer than 50ms) and unpredictable. In designing an echo cancellation function, one must consider the echo tail length, speed of convergence, double-talk performance, and algorithm complexity. ITU G.168 defines performance requirements for echo cancellers.

Vocoders—paired speech compression and decompression algorithms used in IP telephony—are ITU standards. Examples include G.711, G.723, G.729, and G.729a. The differences between these algorithms reflect trade-offs between speech quality, network bandwidth utilization, computation complexity, and latency. Other features, such as voice-activity detection and comfort noise generation, need to be included to enhance speech quality and conserve bandwidth. Table 1 shows the performance of some of the popular speech codecs.

The ITU has developed a set of standards for multimedia communications over packet-based networks. These standards are aggregated under recommendation H.323. This recommendation describes terminals, equipment, and services for multimedia communications over packet networks.

Voice quality

The voice quality achieved by a VoIP system depends on a number of factors including the quality of the phones, network congestion, and signal processing algorithms used. In particular, it depends on the following factors:

- Vocoder—The choice of the speech codec affects the intelligibility of the speech. The choice is based on a trade-off between voice quality, bandwidth computation, complexity, and latency
- Echoes—Imperfections in the network give rise to echoes that, in turn, affect the intelligibility of the speech. Echo cancellers are required to cancel the echoes and enhance the quality of speech
- Latency—Because delays in packet networks are significant and unpredictable, they can have a significant impact on speech quality. Delays arise because of buffering, processing, and congestion

For consumer applications, not only must a VoIP system's performance be adequate, but the price of the product must be affordable. System cost has to be limited to a reasonable value to stimulate demand. Therefore, VoIP implementations must utilize cost-effective components.

TABLE 2 VoIP software modules

Software module	MIPS	Code size
DTMF, Call Progress Tones, and Caller ID support	2 to 3	15KB
Line Echo Cancellation	5 to 14 (4ms to 32ms tail length)	7KB
Vocoders		
G.711 Compander	<1	<1KB
G.721 ADPCM	10	2.1KB
G.722	10	2.6KB
G.723.1	29.7 (VAD off)	56KB
G.729	29.5	35KB
G.729a	16.5	32KB
Voice Activation Detection	approx. 3 to 4	approx. 8KB

Hardware requirements

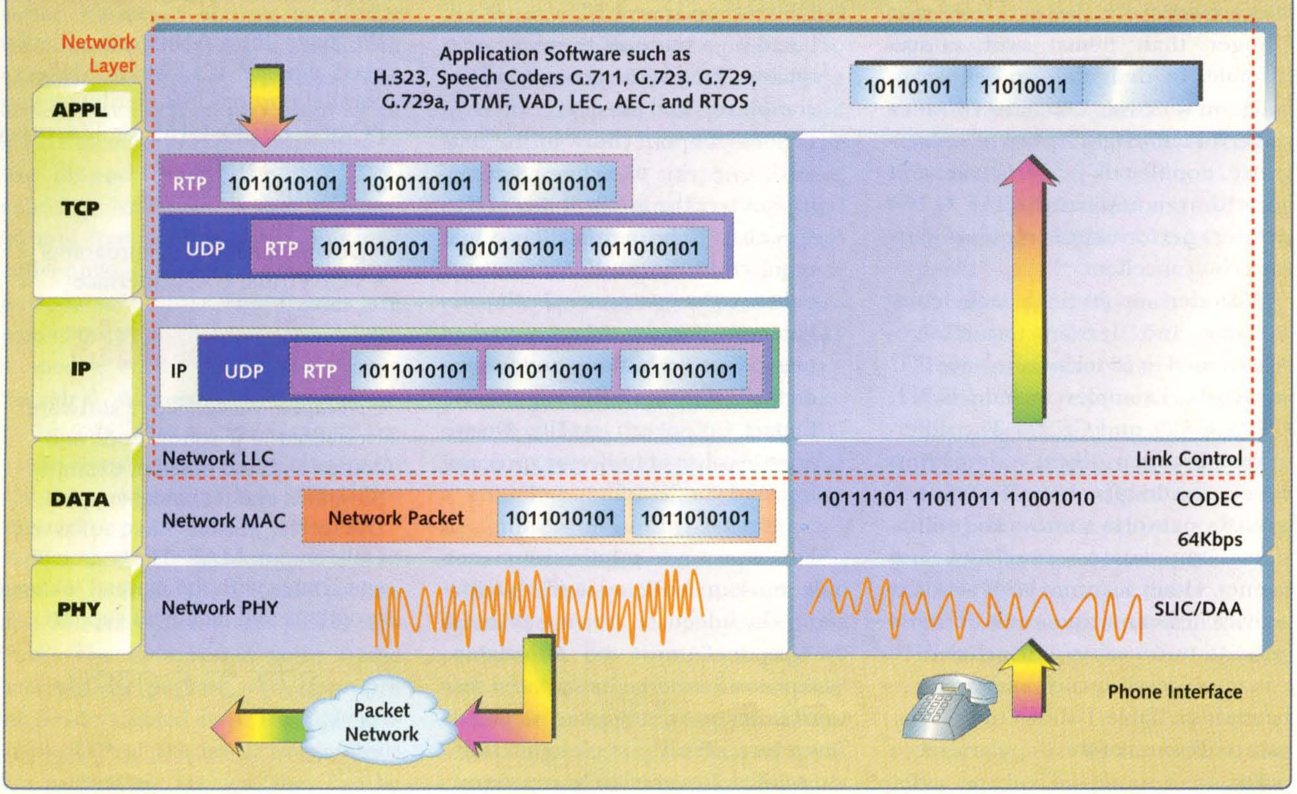
Several software modules that would typically be run on a DSP, including Vocoder middleware, are shown in Table 2 with their MIPS requirement and code size.

The PCM interface module captures the voice samples at the desired rate and stores them into the memory. It also supplies the voice samples to the CODEC for playback. In our current implementation, voice samples are generated at a rate of 8,000 per second. Eight voice samples are stored in the buffer and transferred into memory every millisecond. A timer set during initialization for this purpose provides an interrupt to the RTOS to start the transfer task. This task requires less than 1 MIPS. Since the RTOS also handles other interrupts such as network packet transfer and ocoder initialization, it requires about 1 to 2 MIPS. This software is not included in the table, since it would typically be run on a traditional CPU, not a DSP.

When a call is initiated by the handset, the RTOS monitors the off-hook signal and starts the appropriate DTMF task. When a call comes from the packet network, it provides the ring signal to the handset and monitors the off-hook signal again. Tasks in this group continue to provide monitoring of other calls. In total, these tasks require 1 to 2 MIPS.

The line echo cancellation (LEC) task requires 5 to 14 MIPS depending on the tail length, which can be from 4ms to 32ms of the echo. Voice samples coming from the handset include the echo of the played voice. This echo needs to be cancelled at this point, otherwise it will play back at the origin. The tail length depends on the sending and receiving points. It is generally determined during the initial set up and then remains constant during

FIGURE 2 VoIP network processing


Network Layer
APPL
TCP
IP
DATA
PHY
Application Software such as H.323, Speech Coders G.711, G.723, G.729, G.729a, DTMF, VAD, LEC, AEC, and RTOS
10110101 11010011
RTP 1011010101 1010110101 1011010101
UDP RTP 1011010101 1010110101 1011010101
IP UDP RTP 1011010101 1010110101 1011010101
Network LLC
Link Control
Network MAC
Network Packet 1011010101 1011010101
10111101 11011011 11001010
CODEC 64Kbps
Network PHY
SLIC/DAA
Packet Network
Phone Interface

BIG FEATURES
SMALL FOOTPRINT

With thousands of embedded applications and 25 years in the business, U S Software continues to build cutting-edge, yet compact development tools.

Our software has a history of acclaimed performance and features:

- Compact – Even with industry-leading innovations our small footprint saves you valuable space.
- Fast – Optimized design ensures quick, efficient performance.
- Customizable – Source code is included to give you complete control.*

In the beginning there was silicon. Then, there was U S Software. And throughout, our engineers have crafted software and provided technical support designed to ensure your success.

Call us today at **800-356-7097** and entrust your next embedded project to the engineers and products at U S Software.

*Netpeer™ is delivered as a library.

U S SOFTWARE®
EMBEDDED EXCELLENCE

7175 NW Evergreen Pkwy. Suite 100 Hillsboro, OR 97124
TEL: 503-844-6614 • FAX: 503-844-6480 • EMAIL: info@ussw.com • www.ussw.com

U S Software products are compatible with:

x86
386/486PM
680x0/683xx,
DragonBall EZ
PowerPC
ARM
StrongARM
SH 1, 2, 3, 4
SPARCLite
i960
NEC V85x
M*Core
ColdFire
MIPS
C166
CR16
8096/196
68HC11
68HC16
80251
Z80/180

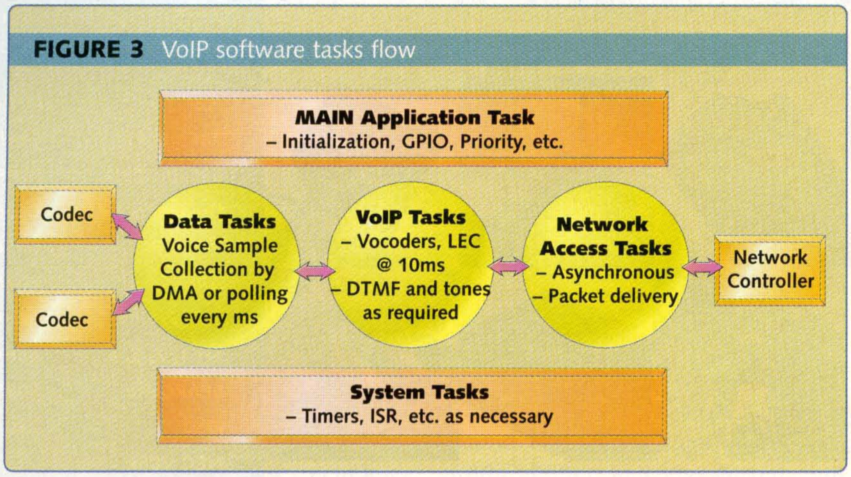
See our website for additional microprocessors we support.

- **USNET®**
  Embedded TCP/IP protocols, Internet Access Package, SNMP and Embedded Webserver.
- **SUPERTASK!™**
  Multitasking RTOS development suite. ITRON 2.x/3.x API available.
- **USFILES®**
  Windows/DOS compatible file system with CompactFlash and CD-ROM support.
- **NETPEER™**
  Distributed networking platform with comprehensive desktop interface including GUI, database, multimedia and more.
- **SUPERPEG™**
  Embedded GUI for SuperTask!®
- **IRPRO™**
  IrDA protocol stack with SIR and FIR

800-356-7097

WWW.USSW.COM

FIGURE 3 VoIP software tasks flow



the conversation. When tail length is significant, the LEC module needs to process more samples, and this requires more MIPS.

When a call is made, the call setup module determines which vocoder is to be used. Vocoders perform the compression and decompression of the voice samples as necessary. Each vocoder uses a special method to compress the voice. Thus MIPS requirements vary. For the SH3-DSP, the MIPS required for vocoder functions range from less than one to over 30. All other MIPS data shown in Table 2 are measured quantities on the SH3-DSP hardware and its cycle accurate simulator. The voice activation detection (VAD) module is included in this table because it requires significant DSP processing. In the current implementation, G.729a is executed every 10ms after collecting voice samples every 1ms.

The TCP/IP stack converts the voice frames into IP packet for the transmission over any packet network. In our case, it prepares the IP packets for the Ethernet transmission. It requires about 1 MIPS to 2 MIPS, and Ethernet control task takes less than 1 MIPS. When the VoIP application is running, overall control of the platform functions is maintained by several interrupts and by synchronous tasks using timers. An additional 1 MIPS is allocated for this overall control flow. For an incoming call, the IP packet received by the Ethernet initiates the application, and processing is performed accordingly. The call set up procedure properly starts the voice processing after all steps for timers and interrupt priority are performed.

The relationship between all tasks, input data, and output data is shown in Figure 3. The voice samples come from the codec. These samples are compressed, packetized using TCP/IP, and then delivered to the network interface for transfer over the LAN/WAN. Processing begins when the off-hook condition is detected from the code. Samples are collected every millisecond, compressed at 10ms intervals, and delivered asynchronously.

The total MIPS needed for a single channel depends heavily on the type of vocoder used and on the tail length of the line echo. Assuming G.729a and a 4ms tail length, the total MIPS required by one voice channel is about 27 MIPS. When G.711 is used, less processing power is needed, about 10 MIPS. At the worst (assuming a 32ms tail length and adding some MIPS for coordination), one channel generally requires a maximum of 35 to 38 MIPS.

These estimates can be used to design a system with adequate DSP and CPU processing power to support from one to any number of VoIP channels. The breakdown of software modules should also help you to understand what you'll need to implement to include VoIP support in your system.

If you already have an RTOS and TCP/IP stack, I think you'll find that the additional code space requirements are not a major issue. However, the complexity of adding a DSP to a system and managing the flow of data between that processor and the CPU can make for some long hours in the lab. **esp**

Yashvant Jani received his PhD in physics from University of Texas-Dallas in 1976. He currently works for Hitachi on residential gateways and applications of SH3-DSP, SH4, and similar devices. Yashvant has also supported embedded controller designs using SH and H8 microcontrollers, and developed embedded system architectures for PDAs, hard disk drives, and voice pagers. You can e-mail him at yashvant.jani@hsa.hitachi.com

References

Goncalves, Marcus. *Voice Over IP Networks*. New York: McGraw-Hill, 1999.

"Voice Over IP (VoIP)" technology guide at *www.techguide.com*.

Bear, Eric, "Designing an Embedded Voice-over Packet Network Gateway," *Communication Systems Design*, October 1998, p. 21.

The International Telecommunication Union (ITU) Web site has information available about all standards mentioned here—*www.itu.int*. For all relevant information regarding G-series standards, go to *www.itu.int/publications/itu-t/itutg.htm*.

The following Web sites have detailed information regarding voice over IP:
www.telogy.com
www.vocaltec.com
www.voxware.com
www.iptelephony.org
www.micom.com
www.itmag.com

Make Your Embedded Platform Soar! with Treck Inc.

Real-Time Internet Protocols for Embedded Systems

We know that schedules are always tight, and that you do not have time to "port" Internet protocols into your embedded environment (let alone write them from scratch). That does not stop marketing people inside your company from asking for Internet functionality in your new product line that is rolling out in the next few weeks. Now you have someone to turn to.

High quality, high performance Internet protocols are what makes us the #1 choice for designers of embedded systems who need Internet connectivity. We design and implement our products specifically for embedded systems. That is why we are three times faster than our fastest competitor. Of course our Internet protocols do not require any porting to your specific platform. This means that you will never need to change a single line of our ANSI "C" code. You do not need an RTOS to use our Internet protocols, but if you have one, we will certainly work with it.

We are very committed to supporting you in your development effort. We know that you do not have time to wait for someone to call days later to get a support answer. That is why we provide the "finest" support in the industry. You can always count on Treck Inc. to help you every step of the way.

Our base product (Treck Real-Time TCP/IP), includes Sockets, TCP, UDP, ICMP, IP, Ethernet, ARP, SLIP and Ping. Treck also provides support for PPP, DHCP, BootP, FTP, and Telnet. All of our Internet products arrive at your door in source code form and are always royalty free.

Call us today to see how Treck Inc. can help you!

Treck Inc.

(800)340-6648 or (513)688-0553

or visit our web page at www.treck.com

Solutions Today... For Tomorrow's Embedded Technology

Let EBSnet help you build your next embedded device!

- All Source Code is Provided
- 100% ANSI "C"
- Royalty Free
- Competitive Prices

Prompt Competent Support

- Consulting
- Satisfaction Guaranteed!

For More Information,
Visit Our Web Site

www.ebsnetinc.com

1-800-428-9340

P.O. Box 873 • 39 Court Street
Groton, MA 01450
Phone: 978 448 9340 • Fax: 978 448 6376

http://www.ebsnetinc.com • email: sales@ebsnetinc .com

EMBEDDED TCP-IP NETWORK STACK

The EBSnet Cross Development System provides comprehensive TCP/IP protocols and network applications for embedded CPUs. RTIP 3.0-MPC provides support for Motorola microprocessors including PowerPC, MPC860, 68K, 68360, and Coldfire. Systems are also available for 808x, 80186, 386, ARM, SparcLite, Mips, Hitachi, Mitsubishi, Philips, and Infineon.

INTERNET PROTOCOLS

SUPPORTS UDP, TCP, ARP, RARP, BOOTP, ICMP, IGMP, DNS

DEVICE DRIVERS

Ethernet, 100 Base-T, Uart, PCMCIA based, PCI based

KERNEL DRIVERS

AMX®, CMX®, SMX®, Nucleus®, RtKernel®, Pharlap®,
TNT®, ETS®, TNTRT®, PSOS®, RTXC®, RTPX,
Polled (no kernel), UCOS®, ECOS®, VRTX®, Threadx®

PORTING LAYER

Ports easily to any realtime OS/CPU

DIAL-UP

SLIP, CSLIP, PPP, modem support

APPLICATION PROTOCOLS

SNMP, DHCP, NFS, FTP, TFTP, Telnet, Mail (SMTP, POP3, IM AP), RIP, and Virtual/Memory File Systems

WEB SERVER

Full featured embedded Web Server (with diskless option)

EMBEDDED WEB BROWSER

I-BROWZR, EBSnet's Embedded Web Client, is a complete tool for developing WEB browser interfaces for interactive internet appliances and other embedded systems that wish to use HTML to present a user interface.

I-BROWZR is written in portable C++, will port to any target that can support the PEG Graphics Library and uses standard socket calls to access the network.

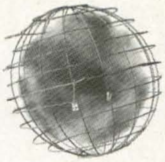
I-BROWZR is compatible with EBSnet's RTIP but can be standalone.

ERTFS-DOS COMPATIBLE FILE SYSTEM

Comprehensive, portable, high performance
DOS compatible file I/O for embedded systems.

ebSnet
INC

Y2K



Embedded Internet Tools

Internet processor

The Au1000 Internet Edge processor is based on the MIPS instruction set and is designed for high-performance at low-power. It runs at speeds of up to 500MHz. Its peripheral set targets communications market segments whose products operate on the "outer edge" of the Internet network, such as wireless palm-sized PCs, third-generation data devices, VoIP telephony devices, firewalls, and routers. The Au100's architecture consists of a 32-bit core and includes a five-stage scalar pipeline, 16K instruction cache, a 16K non-blocking data cache, a high-speed multiply-accumulate (MAC) unit and an R400 class memory management unit (MMU). Its peripherals include 10/100 Ethernet controllers, fast IrDA, multiple UARTs, AC 97 controller, and I2S.

Alchemy Semiconductor
Austin, TX
(512) 421-6200
www.alchemysemi.com

Client software

EmWeb/Client is software that allows devices to initiate basic HTTP transactions (GET, PUT, and POST) with any web server on the Internet for communications such as the transmission of new configuration files or firmware images, notification of web-based alarms, and distribution of configuration information to other networked devices. It supports persistent connections (meaning it reuses TCP connections, rather than establishing them each time a URL is to be fetched). Used in conjunction with EmWeb/Server, EmWeb/Client can implement applications through send and receive communications between different devices and web-based applications. The idea is that were you to have a networked printer attached to the Internet by means of a unique web page address, it would be possible to automatically upload a document residing on a different web page to the networked printer.

Agranat Systems Inc.
Maynard, MA
(978) 461-0888
www.agranat.com

Information Linux

Based on Red Hat Linux, NeoLinux is an OS designed for business-to-business information appliances such as cash registers, firewalls, routers, interactive web-kiosks, thin clients, security devices, and wireless appliances. Its features include ezCompress, which allows Linux to be installed on small systems without a hard drive; ezOFF, which allows people to turn off their device without going through the "shut down" process required by a desktop OS; ezSecure, which makes the OS read only; and ezManage, which allows appliances to be remotely managed, configured, and updated across a LAN or WAN. NeoLinux is designed to operate on Intel X86 or compatible processors.

Neoware Systems Inc.
King of Prussia, PA
(610) 277-8300
www.neoware.com

Tools for Bluetooth

The Bluetooth Extension to WinDK is a toolkit that enables developers to attach Bluetooth hardware to a tested Windows Bluetooth stack. It may be used to develop hardware and protocol drivers for Bluetooth implementations based on the Digianswer A/S Bluetooth software stack. Also featured by the Bluetooth Extension to WinDK is support for the development of hardware drivers for USB, PCI, and PCMCIA buses, as well as serial interfaces. Additionally, it lets you access the hardware using any combination of port I/O, memory I/O, interrupts, and DMA. It's available now for $14,995.

BSQUARE
Bellevue, WA
(888) 820-4500
www.bsquare.com

Design kit

Micro WebTarget is a hardware/software prototyping technology designed for Web-enabled cell-phones, PDAs, pagers, and other mobile devices. It's based on the connected limited device configuration component (CLDC) of the Java 2 platform, Micro Edition. The Micro WebTarget features a single processor core from Hyperstone Electronics that combines a 32-bit RISC processor with a 16-/32-bit DSP unit. The processor core provides 200 MOPS of processing power at 50MHz. Also included is a built-in Ethernet controller and interface transformer support for a direct network connection. In addition to 4MB of DRAM and 1MB flash, the MicroWebTarget utilizes HyNetOS, a real-time network operating system designed specifically for the Hyperstone RISC/DSP architecture. The MicroWebTarget is available now for $2,350.

Smart Network Devices
Dussëldorf, Germany
+49 (213) 122-3267
www.smartnd.com

DAVID HINERMAN

Pardon Me, Do You Have the Time?

Many systems need to know the current time. Take note of these helpful hints from a seasoned timekeeper.

Time is a fascinating concept that has kept scientists and philosophers busy for centuries, debating its nature and purpose. While we may not fully understand time, we still need to measure it. Thousands of years ago, hunters and farmers marked the seasons in order to know when to begin the hunt or to plant crops. Many ancient stone and earthen structures around the world appear to have had timekeeping functions.

In modern systems, timekeeping needs range from providing a CPU clock for instruction timing to reporting current time with nanosecond accuracy for navigation purposes.

Fortunately for developers, building a system with timekeeping capability no longer requires moving yards of earth or tons of stone. Electronic timekeeping requirements are so widespread that components and subsystems that go a long way toward filling them are available off-the-shelf. A little forethought can extend those capabilities even further. This article will present a few ideas for answering the age-old question, "What time is it?"

For purposes of this article, let us define *timekeeping* as "the measurement of the passage of time with sufficient accuracy, and in a suitable representation, to meet the requirements of the system."

The passage of time is typically measured by counting repetitive events that occur at consistent intervals (a *timebase*), or by measuring a

Numerous ways of measuring time are available, from watching the sun rise and set to counting vibrations of atoms at a specific state.

condition that varies in a regular way over time.

The earliest timebases were the Earth's rotation about its axis and its revolution around the sun. Early man's timekeeping requirements were fulfilled by counting days and years. Short durations were later measured by observing phenomena such as the flow of water or sand, or the position of the sun in the sky.

During the Renaissance, global sea navigation created a need for accurate timekeeping that didn't rely on the Earth's motion. Mechanical clocks were invented to fulfill that need.

When the age of electronics arrived, methods for developing highly accurate periodic signals were developed, many based on the oscillations of a quartz crystal. It wasn't a difficult step to apply them to timekeeping, first by driving electromechanical counters, later electronic ones. Now methods exist that are accurate enough to uncover the inaccuracies in the Earth's motion.

Timekeeping requirements

Numerous ways of measuring time are available, from watching the sun rise and set to counting vibrations of atoms at a specific state. As embedded system developers, we will usually select a method that is electronic in nature, and (we hope) inexpensive enough that our users can afford it. But how will we know if a method is good enough? What kind of specifications may we be expected to meet?

The specification most commonly given as a timekeeping requirement is *accuracy*. Accuracy is commonly expressed as a maximum allowable error over a unit of time or as a percentage of total time. For example, a wristwatch or real-time clock chip may have a guaranteed error of 30 seconds or less per month. Since the majority of electronic timekeeping methods rely on an oscillator of some sort as a timebase, the accuracy may be given in typical oscillator terms, for example, parts per million.

Another specification that may be given is *resolution*. One wouldn't use a calendar to time a marathon runner, or a stopwatch to decide when to plant corn. If resolution is specified, it will usually determine the unit of measure you will use in your timekeeping system.

Some timekeeping applications will specify the *representation* of time. One of the oldest legacy systems in existence is our method of expressing time: seconds, minutes, hours, days, months, and years. You've probably noticed that the progression of these units doesn't fit conveniently into a decimal or binary numbering system. Nevertheless, sometimes we must present time in the way that our human clients understand it, instead of a more convenient electronic representation.

Time can be represented in many different ways. You should select a method based on the answers to these questions:

- How will the measurement be taken? By counting periodic events such as oscillator cycles, or simply reading the date and time out of a real-time clock chip?
- How will the measurement be used? Will it be used in calculations, scheduling activity, or simply displayed?
- How will the measurement be displayed? Is it an indication of the current date and time, or is it a measurement of duration? What units does the user expect?

A requirement that isn't strictly related to timekeeping is *synchronization*. When systems interact, they are often expected to have a similar view of current time. While it may be acceptable to allow the user to set the time and hope for the best, other applications will require much closer adherence to a common reference. If your system requires synchronization to an external reference, it will directly affect your timekeeping methods.

Time measurement

Perhaps the simplest method of measuring time in an electronic system is to count periodic events such as oscillator cycles. Inexpensive quartz crystals or even less expensive ceramic resonators or RC oscillators can give sufficient accuracy for most applications. With an oscillator operating at frequencies of a megahertz or more, timing resolution of a microsecond or better is possible. One may signal the start and end of a duration with hardware or software, and the resulting measurement is a simple number that may be recorded, compared, and used in calculations without dealing with modulo-60 issues.

Real-time clock chips are available that combine an oscillator, divider chain, and 32-bit counter in a convenient package. The Dallas Semiconductor DS1602 Elapsed Time Counter is an example. The oscillator and divider provide a one-second tick to the counter, which may be read or set via a serial interface. The many variants of the Unix operating system keep real time as the number of seconds since midnight, January 1, 1970 (in a signed 32-bit integer), so this sort of hardware could be a good timekeeping solution for a Unix-like system.

Of course, almost anything hardware can do, software can do cheaper. A software counter may be used to count ticks, as long as the CPU is

HANNAH BLAIR

guaranteed not to miss any. If time must be maintained when the system is powered down, software probably isn't the best solution.

The common method of expressing time is a little more complicated than counting ticks, but a lot of the work is already done. Real-time clock chips are available that use the oscillator and divider of the chip described above to increment a set of nested counters to provide seconds, minutes, hours, days, months, and years. These chips also take into account months of varying length and leap years. Dallas, Motorola, and other semiconductor manufacturers offer these types of products.

Again, software can perform the same functions as these hardware counters. Instead of incrementing a single counter, one must keep a series of counters and increment higher-order counters when the lower-order ones roll over. Also, the value at which a day-of-month counter rolls over varies depending on the month and year. A software clock may still not be a good solution if timekeeping must be maintained while power is off.

Even if time is kept as a 32-bit seconds counter, routines are available in run-time libraries or freely available source that will convert to and from the real-world units. Standard C run-time library functions such as `time()` and `ctime()` can be used to deal with such conversions.

Time operations

Regardless of the method (hardware or software) and representation (single counter or multiple) one chooses to keep time, certain basic operations must be supported. Timekeeping systems must be *initialized*, *set*, and *read*. In some cases, they must also be *synchronized* to external signals. In many cases an external signal may be used as a timebase to increment the time, so a *tick* routine or input may be necessary.

Initialization is self-explanatory. Most systems must be brought into a known state before they may be used. This could include zeroing counters, or at least removing illegal values (such as a random startup value of 29 in an hour counter). It may also prepare inputs and outputs to be ready for operation.

A set operation places a value into the timekeeping system. While this is usually done in response to a user command, it may also be done automatically by external systems. Sometimes the set and initialize operations may be combined into a single operation. A read operation gets the contents of the timekeeping system.

Synchronization is the process of getting the timekeeping system to agree with an external time reference. Most timekeeping systems, no matter how accurate, will drift apart if left to themselves. For systems that must operate for long periods yet maintain agreement on the current time, a method of synchronization is necessary. An early example is the clocks used on railroads across the United States almost a hundred years ago. These clocks were all connected to the telegraph lines that paralleled the railroads. Every day near noon, the clocks would wait for a pulse to come down the telegraph line. When the pulse arrived, the clock would immediately jump to exactly 12:00 and pick up timekeeping there. That way, rail stations across the country were all on the same time, within practical limits.

The *tick* process is part of almost every timekeeping system. It may be internal to the system, in the form of an oscillator, or it may come from an external source. This is one of the most accurate ways to synchronize timekeeping, by the way—in effect, every tick of the timebase resynchronizes the local system. Many household and even industrial clocks use the AC power line frequency as a timebase, and take advantage of that common signal to maintain synchronization.

Time usually must be treated as a shared variable. It is common to use a periodic interrupt, generated by a timer or other signal, to trigger an interrupt service routine that increments the time. If a higher-priority interrupt should occur and its ISR should read the time while it's being updated, it may get the wrong value. This can even be a problem with hardware clocks, although most modern parts have circuitry to prevent collisions between reads and writes.

Practical procedures

Developing a timekeeping system need not be a difficult thing, although at times it's not as simple as it seems. If all that is required is to be able to display the current time of day, a simple set of nested counters updated by software in response to some periodic signal, such as a timer interrupt, is an inexpensive and straightforward solution. If time must be maintained in the absence of power, a battery-powered clock chip is an effective solution, in which case a simple driver routine to read and write the chip is the only software necessary, especially if the clock chip has the battery built in. Such parts are easier to design in than a separate battery and power switching circuitry.

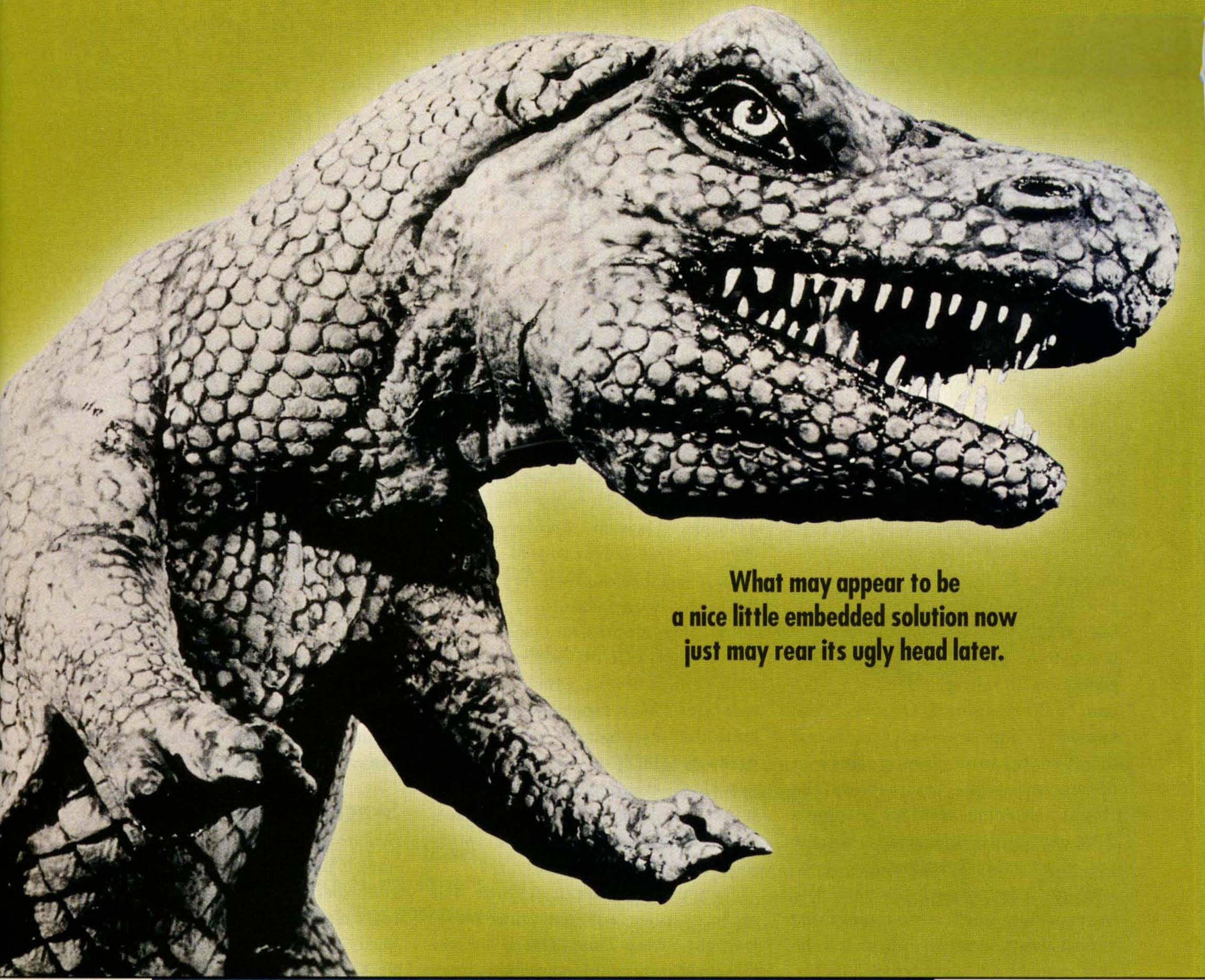
If the system is required to perform certain actions in response to a schedule, or if events will be recorded and stored or presented in chronological order, perhaps a single counter (hardware or software) to count seconds is a better choice. It's easier to do a date and time comparison by doing this:

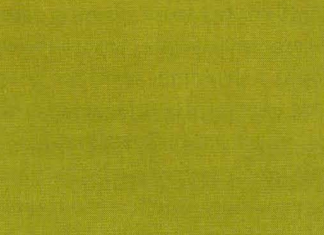
```
if (presentTime == nextAlarmTime)
{...
```

than by doing this:

```
if ((presentSecond ==
  nextAlarmSecond) &&
  (presentMinute ==
  nextAlarmMinute) && ...)
{...
```

This makes sorting of dates and times easier, as well as detecting a scheduled time. If date and time must be displayed while using this time representation, conversion routines are available—many C run-time libraries


What may appear to be
a nice little embedded solution now
just may rear its ugly head later.


FTP
Email
Web Server
Web Browser
Telnet
SNMP v1,v3
DHCP Server
DHCP Client
ARP Ethernet
RIP
NAT Routing
PPP
TCP/IP

include them already—to go to and from individual second, minutes, hours, month, day, year, and day-of-week representation.

One of the things that should be considered when deciding whether to use hardware or software for timekeeping is the resolution required. If your system must resolve time to microseconds, software timekeeping with a periodic interrupt to increment it probably isn't going to work. An off-the-shelf clock chip probably won't do either, since their timebases are typically a one-second tick.

Reading time from a clock chip is usually straightforward, as long as you have a proper driver for the chip's interface. Memory-mapped registers are usually simple—read and write them like RAM. A serial interface clock chip, if it uses a standard serial bus protocol like I²C, is almost as easy if you have an I²C interface on your processor. If not, you'll have to bit-bang a driver using I/O lines. Serial interface chips are slower to access than their byte-wide counterparts, so if the amount of time it takes to read the chip is a problem, consider keeping a RAM copy of the time, and letting the rest of the system read that.

When a system includes timekeeping capability, one feature request that almost always seems to appear is to be able to make an "alarm clock," that is, a feature that schedules some operation to occur in the future. Whether the request makes sense or not is up to you and your marketing people, but if you have to do it, here are a couple things to keep in mind:

- Consider using a single counter for timekeeping. One comparison is easier to make than six
- When comparing, test for present time being greater than or equal to the schedule. If you happen to miss making the comparison at the exact time called for in the schedule, you'll catch it on the next second. Otherwise, you may never catch it

Potential problems

Setting the time has essentially the same problems as reading the time—as a shared data item, it's important to avoid writing the new time while a read is in progress. A hardware clock chip has two sources of writes to the current time: the set operation in software, and the normal tick update which is driven by hardware. When applying a hardware clock, check the vendor's documentation to see it provides a built-in synchronization method, or a suggestion for preventing simultaneous writes.

A common problem that can affect a timekeeping system is when multiple copies of time exist in the system. For example, a system may keep time internally as a 32-bit count of seconds, but convert the counter's contents to a structure with seconds, minutes, hours, days, months, and years for display purposes. If the two representations get out of sync, which is the correct one? This is especially true of systems that use battery backup on a hardware clock—at system power-up the clock chip may be the master source only long enough to initialize a software timekeeping system, which then becomes the master.

You may require a feature that amounts to an "automatic time set"—Daylight Saving Time. This is a tricky feature from an implementation standpoint—the date on which Daylight Saving Time starts and stops are set by law, and may change at the whim of the legislature. Clock chips that have a Daylight Saving Time shift algorithm built in are in danger of being made obsolete with the stroke of a pen.

If you must handle Daylight Saving Time automatically, consider running your core timekeeping on Standard time, or even Universal Time (also known as Greenwich Mean Time) and adjust it for Daylight Saving Time when the time is read. This requires that the timekeeping system be able to determine when it is in Daylight Saving Time, and when it is not. The "first Sunday in April until last Sunday in October" period currently used in the United States can be determined with a moderately simple algorithm. Or, the system could keep a list of dates on which Daylight Saving Time starts and stops. This method has the advantage that it can be more easily changed if the law is changed or if your product is to be marketed outside the U.S.—a simple table update, instead of re-coding an algorithm.

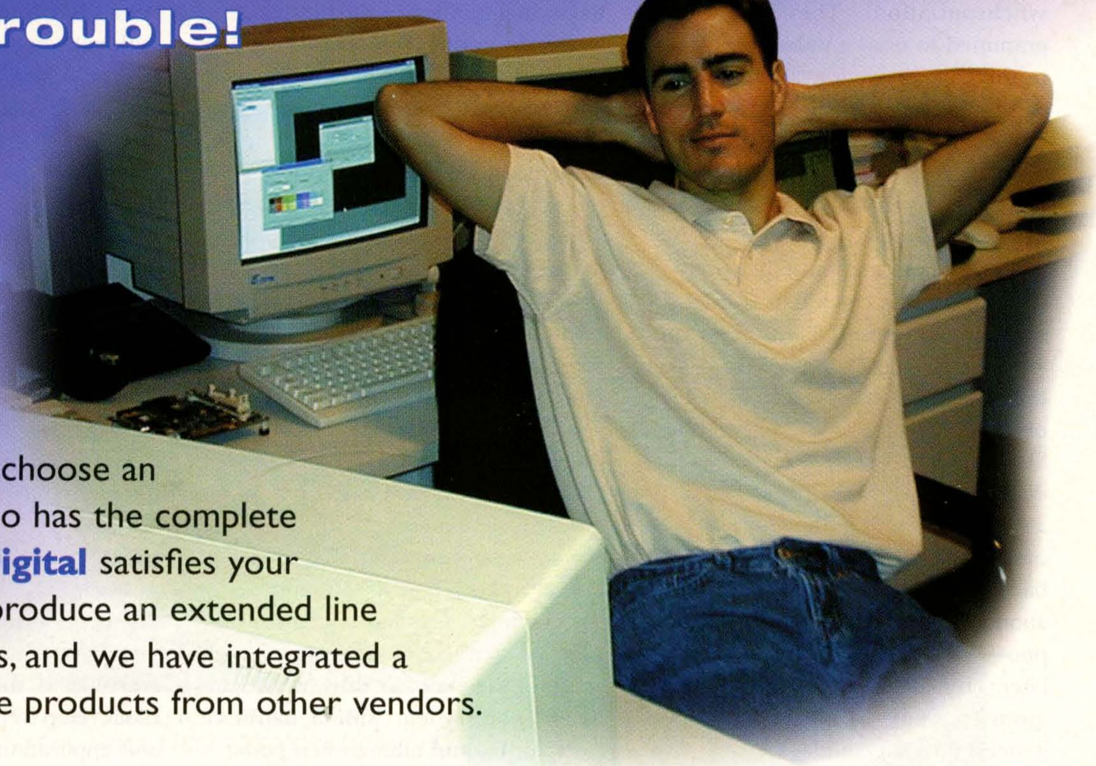
If you *must* adjust your base clock for Daylight Saving Time, be very careful of the autumn ("fall back") shift. For example, in the U. S. the hour from 1:00 a.m. to 2:00 a.m. is repeated—the first time as Daylight Saving Time, the second as Standard Time. If your system waits until 2:00 a.m. and sets the clock back, it will see 2:00 a.m. again in one hour. Unless your system remembers that it has already changed the time, it will set the time back again. (I still cringe at the thought of going to work on the Monday after the autumn time change, even though I learned this lesson over 10 years ago.)

Synchronization

Synchronizing to an external reference is probably one of the most difficult aspects of developing a timekeeping system. If the desired source is a periodic signal that is guaranteed to be available, you can use it to trigger the tick process that will update your time counter. However, if the reference isn't always available (for example, it occurs only once per day, or it may arrive when your system is powered down), this may not be an option.

In its simplest form, a synchronization signal is a time set operation. The user (or a host system, or some other authority) sends a time set command to the system. If the uncertainty of when the command is executed (which will contribute to timekeeping inaccuracy) is acceptable, this is probably the easiest method of synchronization.

A sync signal can be thought of as a time set command with a default time value implied. The railroad clocks



mentioned previously used this sort of synchronization. They were programmed to treat a pulse that arrived in the correct window as a command to set the time to the nearest noon. (Be careful of the distinction between "nearest noon" and "next noon"—if the clock already reads 12:00:03, next noon is tomorrow.)

Additional care must be given to synchronizing a clock that is used for scheduling events. If a sync command arrives that causes the clock to skip over a scheduled event, decide ahead of time how it should be handled. Should the event be executed immediately? Should it be skipped? This is also true of events that may already have occurred before the time is set back. Should it be repeated? Think about these things before they happen—they're a lot harder to change later. The same consideration must be given to events in the presence of a general time set command.

A number of time services are available over communications media such as networks (including the Internet), dial-up services via modem, shortwave radio, and satellites. Some are even able to adapt to transmission delays to minimize errors. The U.S. National Institute for Standards and Technology (NIST) and the U.S. Naval Observatory (USNO) both operate time standards with incredibly high accuracy, and offer publicly accessible sources of time data and synchronization. NIST offers time sync capabilities via the Internet, dial-up modem connection, telephone voice announcements, and shortwave radio broadcasts on radio station WWV. USNO offers time sync and reference capabilities via Internet, dial-up modem, telephone voice announcements, and can provide information on using the Global Positioning System (GPS) for highly accurate time. The U.S. Coast Guard also operates the LORAN-C system, which can be used for timing as well as navigation. Before you develop a system to rely on one of these services, check with the organization that operates it. Some services may be scheduled to be discontinued in the future.

Synchronizing the time by setting the clock works well enough for most applications, but there are some stumbling blocks you should be aware of. Some real-time clock chips don't reset their internal divider chain when a time set occurs. Most clocks use a 32,768Hz crystal oscillator, divided down to 1Hz, to update their counters. It's possible to set the time into such a chip, but still have it increment within a few microseconds. If you're using a network time service that guarantees an accuracy of milliseconds, it is wasted on a clock that may be off by almost a second. If you must begin the second when the set occurs, either use a chip that resets the divider or keep time in software.

A similar problem occurs when using an external reference signal, such as the power line, to drive a software timekeeping system with a hardware clock for backup timing when power is off. When power is restored, the system can read the present time (accurate to within one second) but has no accurate way to know on which cycle to increment to the next second. The best that can be hoped for is to watch for the hardware clock to increment, and begin counting out cycles from that point. At some point, a new synchronization signal will arrive, starting the cycle counter over at the correct point. Depending on the application, it may be appropriate to flag the user that time synchronization is questionable until the new sync signal is received.

Considering requirements

Keeping time in any sort of system, embedded or otherwise, can be a simple task or a nightmare. Consider the requirements carefully. Too many people think it is simply a matter of adding a clock chip, only to find that the system still won't do what is needed.

Depending on what is required of a timekeeping system, one may choose to keep time either as a single counter of some unit of time, commonly seconds, or as a series of counters of increasingly larger units. A single value representation is convenient for doing time arithmetic and comparisons, as in the case of scheduling events. However, humans, as well as external systems, are accustomed to a multiple-value representation of time given in seconds, minutes, hours, days, months, and years.

Timekeeping that requires a resolution of microseconds to milliseconds requires some sort of hardware solution. Software on most platforms can usually resolve from milliseconds to years or even centuries but is likely to lose the correct time if power fails. Real-time clock hardware is available that will resolve from seconds to years, and maintain time even when external power is off. However, these chips may be difficult to synchronize to an external reference with better than one second resolution.

Whatever the requirements, help is available if the developer needs it. Clock chip vendors have data sheets and application notes that cover most situations, and a good deal of software has been written in existing systems that may be used outright, or modified, to give the necessary functions. **esp**

David Hinerman is a software engineer at Rochester Instrument Systems in Rochester, NY. He has over 15 years experience as an embedded software developer. Contact him at hinermad@worldnet.att.net.

The author would like to thank Norbert Balko for his review and suggestions.

Sources

National Institute for Standards and Technology

Network time service: *www.bldrdoc.gov/timefreq/service/nts.htm*

Dial-up (modem) time service: *www.bldrdoc.gov/timefreq/service/acts.htm*

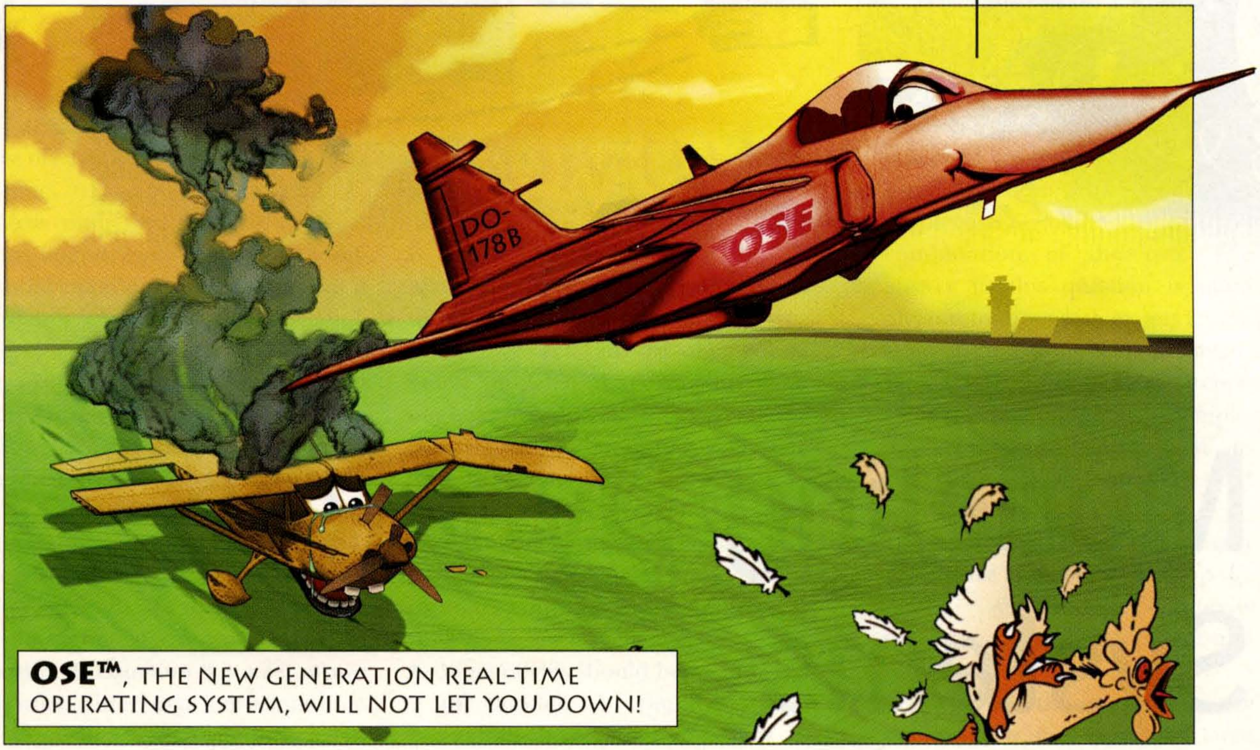
U.S. Naval Observatory

Network, modem, and Global Positioning System time services: *tycho.usno.navy.mil/ctime.html*

U.S. Coast Guard

LORAN-C radionavigation and time service: *www.navcen.uscg.mil/loran/*

– Conventional RTOS? ...sorry!



OSE™, THE NEW GENERATION REAL-TIME OPERATING SYSTEM, WILL NOT LET YOU DOWN!

OSE THE NEW GENERATION RTOS!

SAFER

OSE is the world's only RTOS that is safety certified to the demanding specifications of IEC 61508 . OSE is also being certified according to the stringent DO 178-B.

FASTER

Streamlined for extreme reliability and speed, OSE's kernel and TCP/IP stacks blow away the competition. From the RTOS to the tools, OSE sets tomorrow's standards for small size and high performance.

TOUGHER

As the only fault-tolerant RTOS, OSE supports mission-critical real-time systems, allowing complete non-stop recovery from hardware and software failures AND hot swaps – critical for high-availability functionality.

LONGER-LASTING

OSE is heterogeneous, scalable, and distributable, protecting your investment by allowing your application to grow from one CPU to hundreds.

MORE POWERFUL

OSE's kernel offers automatic supervision, dynamic reconfiguration and integrated error handling, letting you focus on your core competency: designing applications.

SIMPLER

OSE's powerful ultra-efficient message-based architecture lets you write nearly every bit of application code using only eight system calls.

PROVEN

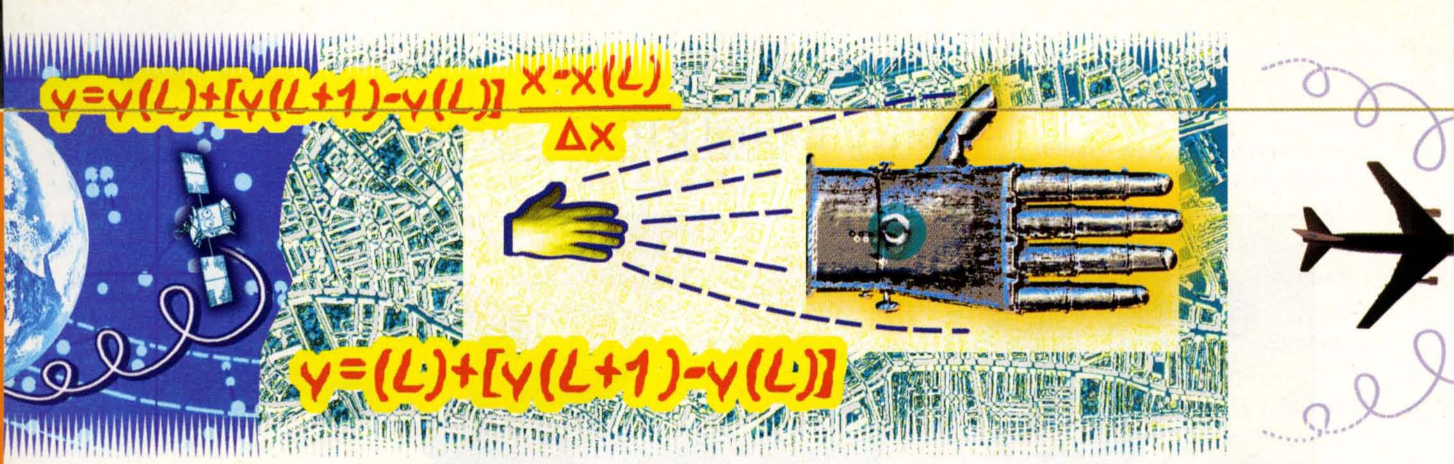
Millions of products worldwide are already taking advantage of OSE , including the top brands in telecommunications and process control. OSE is the RTOS of the future.

www.enea.com

ENEA OSE SYSTEMS

5949 SHERRY LANE , SUITE 625, DALLAS TX 75225. PHONE: 214-346-9339. FAX: 214-346-9344. EMAIL: info@enea.com

JIM LEDIN



Modeling Dynamic Systems

To develop a simulation of a complex dynamic system, you must first develop mathematical models of major system components, as well as of any significant interactions between the system and its operational environment. Here's an introduction to the development of mathematical models of dynamic systems.

A mathematical model is an algorithm or set of equations that is combined with a set of data values to represent the significant behavior of a system, process, or phenomenon.

The development of a mathematical model for a given real-world system can be a difficult task. In cases where the system's dynamics are not well understood, a series of experiments must be performed to collect data that can then be processed using various techniques to yield a model of system behavior. This article introduces some of the methods used in the development of mathematical models of real world systems and phenomena.

Continuous-time dynamic systems

The behavior of a dynamic system evolves over time. This behavior is usually represented by differential equations when modeling continuous-time systems. A system is called continuous-time if its descriptive equations are defined for all values of time. Our focus in this article will be on the modeling of continuous-time systems.

Some examples of continuous-time dynamic systems are the translational and rotational motion of an aircraft, the orbital motion of a satellite, the response of a robotic arm to the motion of its actuators, and the response of an op-amp bandpass filter to an input signal. The differential equations describing the behavior of these systems are called dynamic equations.

High-fidelity dynamic equations representing complex, real-world systems tend to be nonlinear and time-varying, which makes their analytical solution difficult or impossible. When simulating these systems, numerical integration algorithms are used to estimate the solution of the dynamic equations. Because numerical integration is used, a mathematical model of a dynamic system only requires the dynamic equations that describe the system's behavior.

Experts with thorough knowledge of the system and components of interest typically perform the model development.

The actual solution of these equations occurs while the simulation is running.

Most numerical integration algorithms operate only on first-order differential equations. This means that if a dynamic equation contains second-order (or higher) derivatives, it must be transformed into an equivalent set of first-order differential equations. This is a simple procedure, which is described in the sidebar "Modeling High-Order Differential Equations."

Equation 1 below shows a second-order (nonlinear) differential equation. The result of solving Equation 1 for the highest derivative is shown in Equation 2. Equation 3 is a set of two first-order differential equations that are equivalent to Equation 1. In Equation 3, x_1 is equal to the solution function x, and x_2 is equal to the derivative x'. The variables x_1 and x_2 are referred to as state variables:

$$x'' + 3x'^2 + 5x = 1 \qquad (1)$$

$$x'' = -3x'^2 - 5x + 1 \qquad (2)$$

$$\begin{aligned} x_2' &= -3x_2^2 - 5x_1 + 1 \\ x_1' &= x_2 \end{aligned} \qquad (3)$$

In general, a differential equation such as Equation 1 will have many solution functions. Additional information must be provided to specify the solution of interest. In dynamic system simulation, this additional information is given in the form of initial conditions on the state variables. These initial conditions are the values of the state variables at the beginning of the integration interval.

Equation 4 shows an example set of initial conditions at time zero that, combined with Equation 3, uniquely specify the solution to the dynamic equation of Equation 1:

$$\begin{aligned} x_2(0) &= 0 \\ x_1(0) &= 0 \end{aligned} \qquad (4)$$

Mathematical modeling

Mathematical models are developed using the techniques of engineering disciplines relevant to the system being modeled. Experts with thorough knowledge of the system and components of interest typically perform the model development.

The process of model development begins by specifying the requirements the model must meet. Some of the issues that must be addressed are:

- What effects should be included in the model? A complex system exhibits many different kinds of behavior (for example, the motion of motors, vibration, wear of moving parts, and so on), but not all of these behaviors may need to be modeled to produce an effective simulation. Limiting the effects modeled to only those that are necessary will make the model less complex and easier to build, test, and maintain, as well as requiring less computational resources to execute
- How detailed must the model be? In many cases, a simple model is all that is needed, but if precise determination of system behavior is required, the model may need to be very complex
- What interactions between the system and the outside environment must be modeled? For example, a model of communication satellite motion must typically operate in conjunction with a model of the earth's gravitational field, as well as models of other relevant phenomena such as solar pressure
- What techniques will be used to develop the model? A fundamental choice is whether to use physics-based equations or measured data as the basis for the model, or some combination of the two. The answer to this question is often obvious to those with expert knowledge of the system being modeled
- What data must be gathered to perform the modeling? For example, an aerodynamic model of an aircraft may require extensive wind tunnel testing
- How much time and how many people are available to develop the model? As model complexity increases, so do development and test hours
- What computing resources are available for the model? A large model may consume significant amounts of memory, disk space and CPU time. However, given the capabilities of current computers, this may not be a critical issue
- Will the model eventually be used in a hardware-in-the-loop (HIL) simulation? This may place severe constraints on the execution time allowed for the model. Alternatively, a complex model may require high performance computing hardware for use in a HIL simulation, perhaps involving the use of multiple processors
- How will the model implementation be verified and validated? Reasonable ways must be found to confirm that the model is correctly implemented and that its behavior matches the real-world system being modeled to an acceptable degree

These issues should be addressed as part of planning for the simulation effort. The questions above can be applied initially at the highest level of

BEN FISHMAN

A model is called "physics-based" if it is based on the equations of generally accepted physical laws.

Modeling High-Order Differential Equations

The steps below show how to transform an ordinary differential equation of arbitrary order into an equivalent set of first-order differential equations suitable for implementation in a simulation. If the differential equation is of order *n*, the result of this procedure will be *n* first-order differential equations.

1. Solve the equation for the highest order derivative. This places the equation into the form $x^{(n)} = f[t, x, x', \dots x^{(n-1)}]$ where $x^{(n)}$ is the nth order derivative.

2. Make the following substitutions for the function and its derivatives: $x_1 = x$, $x_2 = x'$, $x_3 = x''$, and so on. This changes the equation into the form $x_n' = f[t, x_1, x_2, \dots, x_n]$, a first-order differential equation.

3. Write first-order differential equations for each of the variables x_1 through x_{n-1} as follows: $x'_1 = x_2$, $x'_2 = x_3$, ... $x'_{n-1} = x_n$.

the entire system being simulated and then again and again as the system is broken down into subsystems and individual components to be modeled. The same questions should also be used in the development of additional models that are required for a complete simulation, such as the gravitational field and solar pressure models in the communication satellite previous example.

Level of model complexity

The required complexity of a mathematical model is primarily determined by the answers to the first two questions listed in the previous section: the effects to be modeled and the level of modeling detail required. For most systems complex enough to make a simulation worthwhile, a large number of effects can be identified that potentially have some bearing on performance. The model developer must determine which effects are significant and which can be ignored. This is partially an economic decision because the more effects that are added to a model, the more it will cost to develop and validate, and the longer it will take to complete development.

One useful approach to dealing with these issues is to start with a relatively simple model containing a limited set of effects and a coarse level of model detail. Then, as experience is gained with the simulation, more effects and model details can be added as needed. Often, in the early stages of simulation development, it is not clear which effects and model details are truly significant. If a large number of effects and model details are included in the initial design of the model, it may turn out that much effort has been wasted modeling things that turn out to be insignificant in determining system performance.

If the software interfaces to each model are well defined, it should be possible to replace individual models with higher fidelity representations without the need to make significant changes to the rest of the simulation. It may also be useful to maintain multiple levels of fidelity simultaneously for particular models in the simulation. This will allow the simulation user to select the desired level of fidelity as part of the simulation input data set. Having multiple model fidelity levels available in a simulation can allow the user to perform detailed modeling of particular effects or system details when needed, and potentially reduce execution time significantly on simulation runs when that level of detail is not required.

Modeling methods

We will now discuss some of the commonly used techniques for developing the equations and data sets for a mathematical model. A model is called "physics-based" if it is based on the equations of generally accepted physical laws. A spacecraft orbital model based on Newton's laws of motion is an example of a physics-based model.

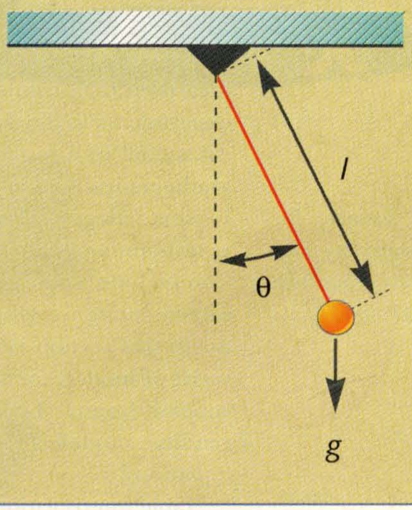
Many systems have behavior that is too complex to represent easily in terms of the laws of physics. The aerodynamics of a supersonic aircraft, for example, tend to be complex and nonlinear. In this case, the only reasonable approach may be to measure the system's behavior with a sub-scale model in a wind tunnel and then create a set of lookup tables to serve as the model. This is called an "empirical" model.

Let's look at an example of a physics-based model. Figure 1 shows a pendulum suspended from a string of length *l* under the influence of gravitational acceleration *g*. The pendulum angular deflection with respect to the vertical is θ, given in radians. The mass of the pendulum bob is defined as *m*. The goal for this model will be to determine the period of oscillation of the pendulum as a function of the initial deflection angle θ_0 assuming that the initial velocity, θ_0, is zero.

Given the goal of determining the oscillation period, the effects that must be modeled will now be considered. We will look at the relevant physical effects and determine which to include in the model and which to ignore:

- Gravity must be modeled, since the pendulum would not move at all without it
- The mass of the pendulum bob must be modeled for the same reason
- If the size of the bob is small in comparison to the length of the string, the bob can be modeled as a

FIGURE 1 Simple pendulum



point mass. This simplifies the model significantly

- If the mass of the string is much less than that of the bob, the string's mass can be ignored
- Friction within the string will be assumed to be a small effect over short time periods and will be ignored
- The pendulum will be assumed to move slowly so that air resistance is not a significant factor over a short time period

We know that a real pendulum will eventually slow down and stop due to friction in the string and air resistance. This isn't the kind of behavior we are interested in, so we will modify our goal to be the determination of the oscillation period at the time its motion is started. This assumes that the pendulum slows gradually and the oscillation period changes slowly.

We have made several simplifying assumptions that will make the model development task easier. Next, we will apply the laws of physics to the system to develop the dynamic equations.

The force of interest due to gravity will act on the pendulum bob in the direction perpendicular to the string at any moment in time. This force is defined as shown in Equation 5:

$$F = -mg\sin\theta \qquad (5)$$

Applying Newton's law $F = ma$ to the problem leads to Equation 6, where a is the acceleration of the bob in the tangential direction:

$$a = -g\sin\theta \qquad (6)$$

The acceleration a is related to the angle θ by the relation $a = l\theta$. This leads to the final dynamic equation given in Equation 7:

$$\ddot{\theta} = -\frac{g}{l}\sin\theta \qquad (7)$$

Note that this equation does not depend on the mass of the bob m; however, it does depend on the assumptions we listed previously. It is also a nonlinear differential equation because the term $\sin\theta$ appears in it. To completely determine a solution for this equation the initial conditions of the system must be specified as shown in Equation 8. The parameter θ_0 in Equation 8 is the angle from which the bob is released at time zero with an initial velocity of zero. For this example, the possible values for θ_0 will be

In situations where the dynamic equations or data values for a mathematical model are not known to the desired degree of precision, alternative methods for model development are necessary.

assumed to lie in the range [–π/2, π/2]:

$$\theta(0) = \theta_0$$
$$\theta(0) = 0 \qquad (8)$$

To make Equation 7 suitable for simulation, it must be restated as a set of first-order differential equations and corresponding initial conditions as shown in Equation 9:

$$\dot{\theta}_2 = -\frac{g}{l}\sin\theta_1$$
$$\dot{\theta}_1 = \theta_2$$
$$\theta_1(0) = \theta_0$$
$$\theta_2(0) = 0 \qquad (9)$$

Using the techniques of numerical integration, these equations can be solved for various values of θ_0 and the oscillation period can be determined from examining the solutions.

This example has demonstrated the basic approach for developing a physics-based mathematical model of a dynamic system. Similar techniques can be used in the development of models from other disciplines such as electronics and chemistry, as long as the dynamic equations describing the system are well defined and the data values appearing in the equations can be determined with sufficient precision. In this example, the only data values required were the gravitational acceleration *g*, the string length *l*, and the initial displacement θ_0. Of course, other data would have to be examined to verify that the assumptions are reasonable, such as the assumption that the mass of the string is small relative to the mass of the bob.

In situations where the dynamic equations or data values for a mathematical model are not known to the desired degree of precision, alternative methods for model development are necessary. These techniques are discussed in the next section.

Empirical modeling

Empirical modeling techniques use measured data from various types of experiments to develop a mathematical model of a system. In reality, all mathematical models are empirical to some degree. For example, the pendulum model in the previous section includes some empirically determined constants. However, our interest in this section is on the development of models for systems with substantial dynamic behavior that is not readily modeled by well-defined equations.

Table interpolation is a static modeling technique used to evaluate functions of the form shown in Equation 10:

$$y = f\left(x_1, x_2, x_3, \mathrm{K}\right) \qquad (10)$$

This is a static method in the sense that it does not permit the direct implementation of dynamic equations. However, table interpolation functions can be used in the construction of dynamic equations. For example, coefficients appearing in the dynamic equations can be computed using table interpolation.

This approach is normally used when the output of the function must be determined experimentally. It is also applicable as a speed optimization technique if a lengthy computation (perhaps an iterative procedure) is required to evaluate the function. In this case, a table interpolation to estimate the result of the computation may execute many times faster than a direct computation.

The function inputs x_1, x_2, and so on can be any variable in the simulation, such as time, a state variable, or even a constant. The number of function inputs is arbitrary, but in practical applications it is usually five or less. As more inputs are added to the function, its memory requirements and

THREADX

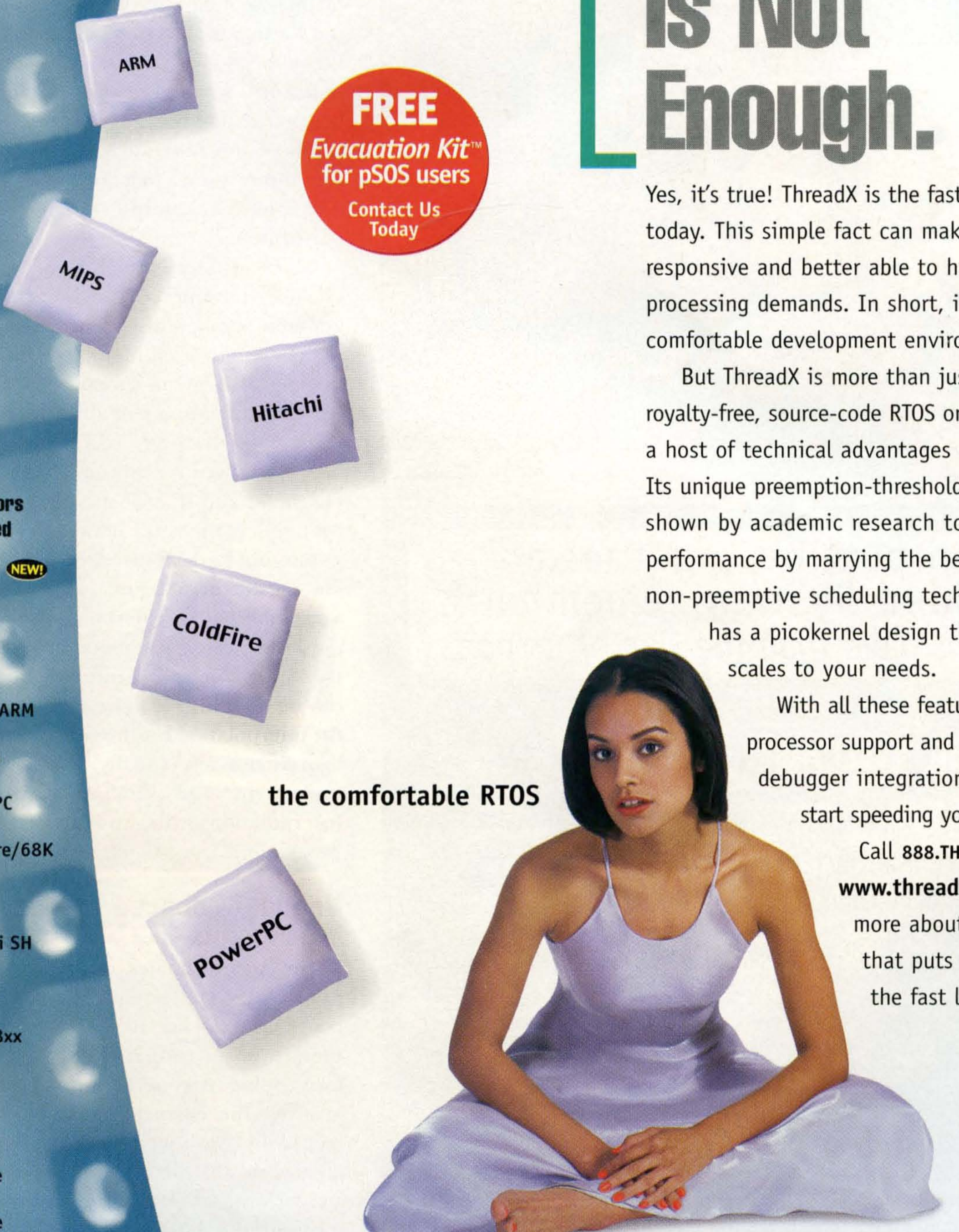
Because Being the Fastest Is Not Enough.

Yes, it's true! ThreadX is the fastest RTOS on the market today. This simple fact can make your application more responsive and better able to handle extreme real-time processing demands. In short, it gives you a more comfortable development environment.

But ThreadX is more than just fast. It is the leading royalty-free, source-code RTOS on the market today, with a host of technical advantages over the competition. Its unique preemption-threshold™ technology has been shown by academic research to improve real-time performance by marrying the best of preemptive and non-preemptive scheduling techniques. ThreadX also has a picokernel design that automatically scales to your needs.

With all these features, plus comprehensive processor support and advanced ThreadX-aware debugger integration, ThreadX can instantly start speeding your application to market. Call **888.THREADX** or visit us at **www.threadx.com** and find out more about ThreadX, the RTOS that puts you comfortably in the fast lane!

FREE
Evacuation Kit™
for pSOS users
Contact Us Today



the comfortable RTOS

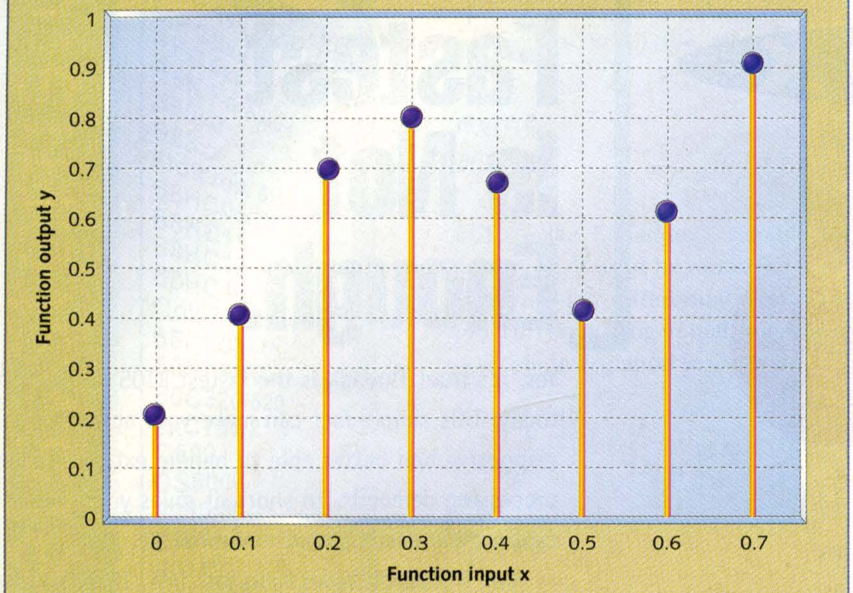
Processors Supported

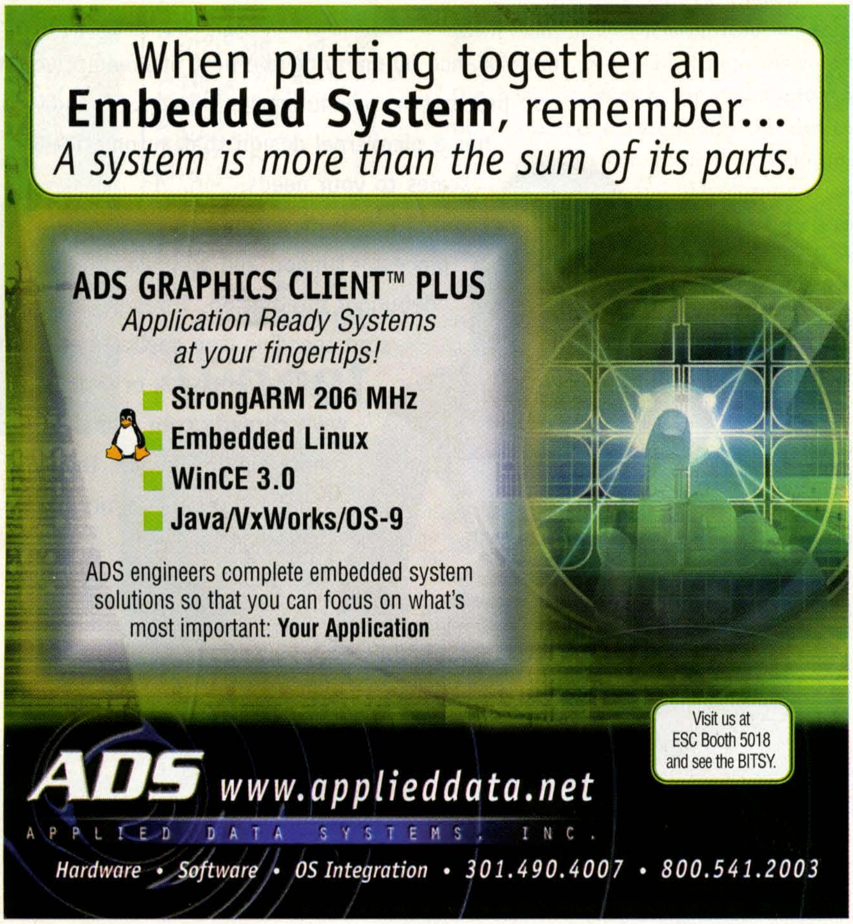
- Win32 NEW!
- ARC
- ARM
- StrongARM
- Thumb
- PowerPC
- ColdFire/68K
- x86
- Hitachi SH
- MIPS
- NEC V8xx
- TinyJ
- SPARC
- M-Core
- TriCore
- SHARC
- TMS320C6x
- TMS320C54x

eL
Express Logic, Inc.

tf **888.THREADX** • *fx* **858.613.6646** • **www.expresslogic.com**

FIGURE 2 Example one-dimensional lookup table





execution time will tend to increase. The output *y* depends only on the values of these inputs when the function is evaluated.

A one-dimensional example of the data set for an interpolation table with eight equally spaced breakpoints is shown in Figure 2. The span of the input variable *x* is [0, 0.7]. If the input variable precisely matches the *x* location of one of the breakpoints it is a simple matter to return the corresponding *y* value as the result of the function evaluation. If the input value falls between the breakpoints an interpolation must be performed.

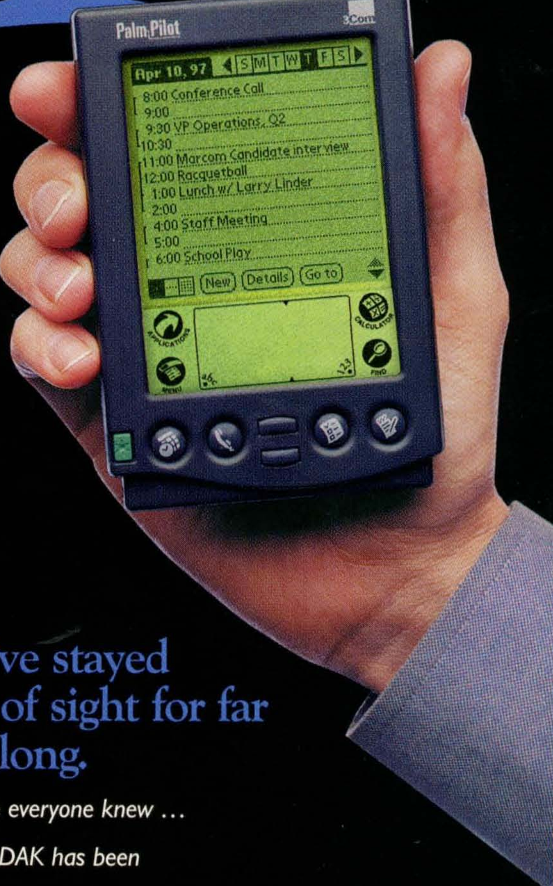
Many different techniques are available for interpolating functions between breakpoints. We will discuss the commonly used method of linear interpolation. One-dimensional linear interpolation is performed graphically by drawing straight lines between adjacent points as shown in Figure 3. The interpolated function changes continuously between breakpoints, but its derivative is discontinuous at the breakpoints. If this not acceptable, other interpolation techniques that produce smoother results (such as cubic spline interpolation[1]) can be used instead. The steps in linear interpolation for the case where *x* coordinates are equally spaced are shown in the sidebar "Linear Interpolation with Equally Spaced Breakpoints."

Unequally spaced breakpoints

If the *x* coordinates of the breakpoints are not equally spaced, it takes more work to determine which point interval contains the function input value. A general approach for locating the correct interval is the technique of bisection. This is an algorithm for performing an efficient search of an ordered list. The steps in the bisection method are shown in the sidebar "Linear Interpolation with Unequally Spaced Breakpoints."

HEY!

We're in here!



We've stayed out of sight for far too long.

It's time everyone knew …

that KADAK has been

quietly setting real-time standards since 1978.

That's right, from Sony to Hewlett Packard, Philips to Hughes Aircraft, over 1,000 of the most demanding companies in the world have chosen our AMX™ kernels to provide the processing power for all manner of smart devices.

From hidden embedded devices to the market-leading 3Com PalmPilot™ connected organizer, they've come to count on KADAK for superior real-time multitasking kernels, crystal-clear documentation, highly-responsive technical support and the ultimate in work and time-saving tools and utilities.

So, next time you need a real-time multitasking kernel remember …You can count on KADAK!

AMX™

Real-Time Multitasking Kernels

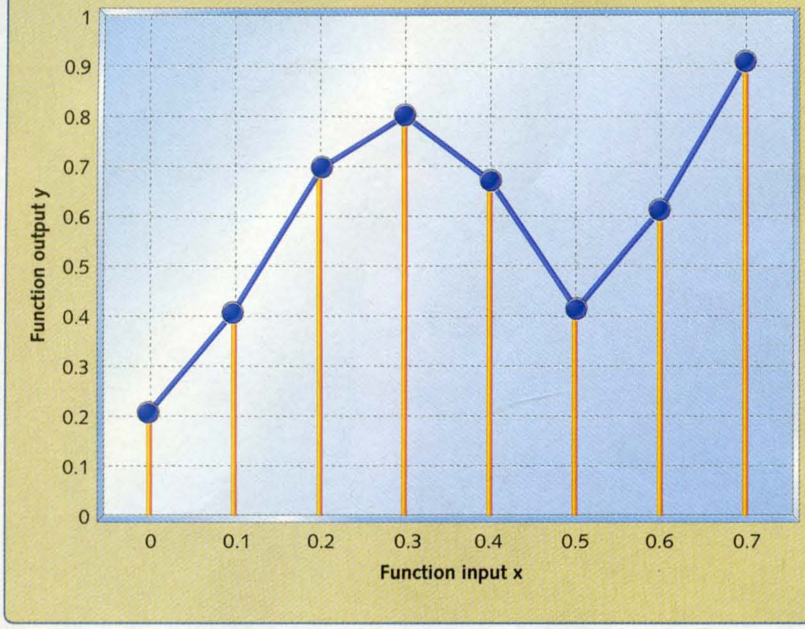
KADAK

Visit us at www.kadak.com

Celebrating our 20th Anniversary

KADAK Products Ltd. 206-1847 W. Broadway, Vancouver, BC, CANADA V6J 1Y5 Tel: (604) 734-2796 Fax: (604) 734-8114 E-mail: amxsales@kadak.com

FIGURE 3 Linear breakpoint interpolation



Linear Interpolation with Equally Spaced Breakpoints

We will assume that the y coordinates of each point is stored in an array of length N with indices that begin at zero. The x coordinates of the points begin at x(0) with a separation of Δx between them.

1. First, ensure that the input variable has a value greater than or equal to the first point in the table and less than the last point. A limit function can be used if that is appropriate. It is also possible to linearly extrapolate outside the table using the first (or last) two data points in the table to define a straight line. However, this approach may produce significant errors if the extrapolation does not accurately model the system's performance outside the table's range and will not be considered here. It may make more sense to issue an error message and abort the simulation run if the input variable is outside the valid input range of the table.

2. Determine the array index of the closest point with an x coordinate that is less than or equal to the function input value, but not equal to the last point in the table. For equally spaced points, the lower point index is computed as shown in Equation 11:

$$L = \text{int}\left(\frac{x - x(0)}{\Delta x}\right) \tag{11}$$

In Equation 11, L is the index of the lower of the two points that surround the input value x, $x(0)$ is the x coordinate of the first point in the array, and Δx is the x interval between points. Based on the range limits placed on x in step 1, L will be in the range $0 \leq L < N\text{-}1$.

3. Perform linear interpolation between the points with indices L and L+1 as shown in Equation 12:

$$y = y(L) + \left[y(L+1) - y(L)\right]\frac{x - x(L)}{\Delta x} \tag{12}$$

Although the bisection method is the most general technique for locating the correct breakpoints interval, it may be possible to eliminate this search much of the time. If the input value x changes slowly between evaluations of the function, the simple step of checking if the input is contained in the same interval as it was for the previous function evaluation will often eliminate the need for bisection. If the input does not lie in the same interval as was used in the previous evaluation, bisection would then be performed.

The bisection algorithm is considerably more time consuming than the direct computation used with equally spaced breakpoints. The advantage of using unequally spaced breakpoints is that it may be possible to adequately model a particular function with a much smaller table than would be required with equally spaced breakpoints. This is because the unequally spaced breakpoints can be closely spaced in regions where the function has rapid fluctuations and more widely spaced in regions where the function is relatively smooth. When using equally spaced breakpoints, all the breakpoints must be spaced closely enough to accommodate the most rapid function fluctuations in the function even if this only occurs over just a small part of the input variable's span.

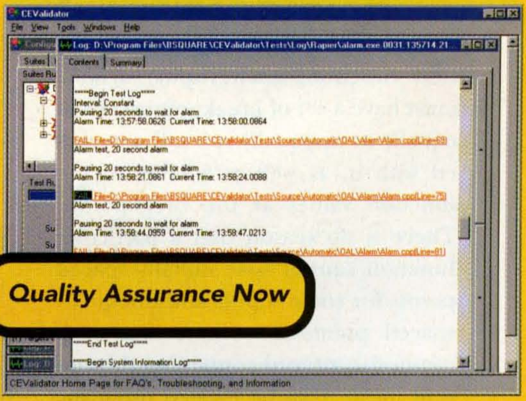
The selection of equally spaced vs. unequally spaced breakpoints depends on the characteristics of the function to be interpolated, the time available for function evaluation, and the memory available for table storage. If unequally spaced breakpoints are used, the locations of the breakpoints should be selected carefully to minimize the number of breakpoints required while simultaneously limiting the magnitude of the interpolation error. When equally spaced breakpoints are used, the point interval must be chosen to limit the maximum interpolation error to an acceptable value.

For every era there's product testing

BSQUARE's QA products and services make your product testing painless.

Put your quality assurance testing on autopilot with BSQUARE's CEValidator™ family of test automation products. Run and record systematic and comprehensive sets of tests for any Windows® CE-based project with the click of a mouse. Complete, thorough testing is at your fingertips. Want us to test it for you? BSQUARE is also your source for custom testing and QA packaged services for all your Windows-based development needs. The QA process has never been faster, easier–or safer.

BSQUARE has helped companies like Motorola®, JVC® and Symbol® Technologies get great products to market fast. **Test us out! Download free evaluation versions at www.bsquare.com/qa**


CEValidator
File View Tools Windows Help
Log: D:\Program Files\BSQUARE\CEValidator\Tests\Log\Rapier\alarm.exe.0031.135714.21...
Suites
Suites Ru
Contents
Summary
Test Ru
Su
Su
Quality Assurance Now
CEValidator Home Page for FAQ's, Troubleshooting, and Information

```
*****Begin Test Log*****
Interval: Constant
Pausing 20 seconds to wait for alarm
Alarm Time: 13:57:58.0626  Current Time: 13:58:00.0864

FAIL: File=D:\Program Files\BSQUARE\CEValidator\Tests\Source\Automatic\OAL\Alarm\Alarm.cpp[Line=69]
Alarm test, 20 second alarm

Pausing 20 seconds to wait for alarm
Alarm Time: 13:58:21.0861  Current Time: 13:58:24.0088

FAIL: File=D:\Program Files\BSQUARE\CEValidator\Tests\Source\Automatic\OAL\Alarm\Alarm.cpp[Line=75]
Alarm test, 20 second alarm

Pausing 20 seconds to wait for alarm
Alarm Time: 13:58:44.0959  Current Time: 13:58:47.0213

FAIL: File=D:\Program Files\BSQUARE\CEValidator\Tests\Source\Automatic\OAL\Alarm\Alarm.cpp[Line=81]

*****End Test Log*****

*****Begin System Information Log*****
```

BSQUARE Solutions

- Platform development and connectivity
- QA tools and services
- OS licensing and tools, training, end-user software
- Engineering services and support
- Solutions for set-top boxes, Web pads, Windows-based terminals, mobile & wireless devices and more

Innovating tomorrow's devices™

bSQUARE

www.bsquare.com

Linear Interpolation with Unequally Spaced Breakpoints

This algorithm uses the bisection method to locate the point pair surrounding a function input value. We will assume that the *x* and *y* coordinate arrays are stored in the same manner as was used for the *y* coordinates for equally spaced breakpoints.

1. Ensure that the input variable *x* has a value greater than or equal to the first point in the table and less than or equal to the last point.

2. Define an index *L* and initialize it to zero. Define an index *U* and initialize it to *N*–1. These lower and upper indices bracket the entire list initially.

3. Repeat these steps until the quantity (*U*–*L*) is equal to one:

a. Set the current index *i* to be (*U*+*L*)/2, truncated to an integer.

b. If the point at index *i* is greater than the input, set *U* = *i*. Otherwise, set *L* = *i*.

4. Upon exiting the loop in the previous step, *L* will contain the index of the lower point of the correct interval.

5. Perform linear interpolation between the points at indices *L* and *L* +1 as shown in Equation 13:

$$y = y(L) + [y(L+1) - y(L)]\frac{x - x(L)}{x(L+1) - x(L)} \qquad (13)$$

Multidimensional table interpolation

We will now examine linear interpolation of multiple-input functions. To evaluate a multiple-input interpolation function, each input variable must have a set of breakpoints (either equally or unequally spaced) associated with it, as well as a table of the function values at the breakpoints. There is no reason that a particular function cannot have equally spaced points for some inputs and unequally spaced points for others. For each input, the interval containing the current input must be located using the interpolation techniques described previously. Then, using these breakpoints, a multidimensional interpolation is performed.

We will look at an example using a two dimensional table with equally spaced breakpoints for both input variables. It is straightforward to extend this example to three or more dimensions. Equations 14, 15, and 16 list the *x* and *y* axis breakpoints and the function values *z* at those breakpoints. The table in Equation 16 is defined so that increasing values of *x* appear across the columns from left to right and increasing values of *y* lie along the rows from top to bottom:

$$x = [3 \quad 4 \quad 5 \quad 6] \qquad (14)$$

$$y = [1.2 \quad 1.4 \quad 1.6 \quad 1.8] \qquad (15)$$

$$z = \begin{bmatrix} 0 & 0.1 & 0.3 & 0.3 \\ 0.1 & 0.3 & 0.5 & 0.4 \\ 0.1 & 0.5 & 0.6 & 0.7 \\ 0.2 & 0.5 & 0.6 & 0.9 \end{bmatrix} \qquad (16)$$

This example uses equally spaced breakpoints, but the steps are identical for unequally spaced breakpoints once the correct intervals have been determined. In Figure 4, the function inputs *x* and *y* define the point *p*, at which we wish to evaluate the function output *z*. The four points defined by the intersection of the lines x=x_1, x=x_2, y=y_1, and y=y_2 represent the surrounding breakpoints defined in Equations 14, 15, and 16. Two-dimensional interpolation is performed in two steps.

First, the interpolated function values at the points p_1 and p_2 in Figure 4 are computed using the formulas shown in Equation 17:

$$\begin{aligned} z(x,y_1) &= z(x_1,y_1) + \\ &\quad [z(x_2,y_1) - z(x_1,y_1)]\frac{x - x_1}{x_2 - x_1} \\ z(x,y_2) &= z(x_1,y_2) + \\ &\quad [z(x_2,y_2) - z(x_1,y_2)]\frac{x - x_1}{x_2 - x_1} \end{aligned} \qquad (17)$$

Then, linear interpolation is performed between p_1 and p_2 along the *y* dimension as shown in Equation 18:

$$\begin{aligned} z(x,y) &= z(x,y_1) + \\ &\quad [z(x,y_2) - z(x,y_1)]\frac{y - y_1}{y_2 - y_1} \end{aligned} \qquad (18)$$

A plot of the function defined by Equations 14, 15, and 16 using linear interpolation is shown in Figure 5. Note that the between-point surfaces resulting from the linear interpolation will not generally be flat. The only time an interpolation surface will be flat is when the four points at the corners of the interpolation intervals for the surface all happen to lie in the same plane.

Multidimensional interpolation can be extended to any number of input variables. For example, with three inputs, linear interpolation is performed across a three dimensional box using the eight function values located at the box corners.

Linear function interpolation is a common tool in the simulation of complex dynamic systems. The selection of equally spaced or unequally spaced points is a trade-off between data table size and speed of function evaluation, which depends on the characteristics of the data that represent the function. Any arbitrary number of inputs can be used in an interpolation function, but as more inputs are added, execution time and memory usage will increase.

System identification

Another technique for developing models of dynamic systems is system

Microware customers have more than 250,000 deployed JAVA™ units running on OS-9® today.

We can help you do the same.

MICROWARE®

SOFTWARE COMPONENTS • CONSULTING SERVICES • DEVELOPMENT TOOLS •
Embedded Solutions for Successful Products™

Get to market first™

If you're about to jump into the market with a new product, make sure you've got a Java-branded embedded solution that goes the distance. Our customer success rate is more than double the industry average – because our customers use our powerful development tools, proven software components like the OS-9 RTOS, and experienced consultants who help them succeed. Microware's Java implementation is tightly integrated with graphics, networking, plug-in support, non-intrusive garbage collection, memory protection, and development tools for Internet solutions.

Microware's record of success in helping OEMs get their designs to market quickly is legendary. Microware solutions power successful designs for world leaders like Hitachi, Intel, ARM Ltd., MIPS Technologies, Motorola, STMicroelectronics, and Toshiba. That's why embedded systems developers on six continents have turned to Microware for more than 20 years.

Call 888-642-7609 today or visit our web site at www.microware.com.

JAVA™ COMPATIBLE

identification.[2] To perform system identification, one or more input data sequences and the corresponding output data sequences are required for the system being modeled. Typically, tests of a real world system are designed and executed to generate this data. Through the application of a variety of algorithms, it is possible to derive an estimate of the system transfer function from input to output. The resulting model is typically linear and time-invariant, so care must be taken to confirm that it is an adequate representation of the system being modeled.

FIGURE 4 Two-dimensional linear interpolation

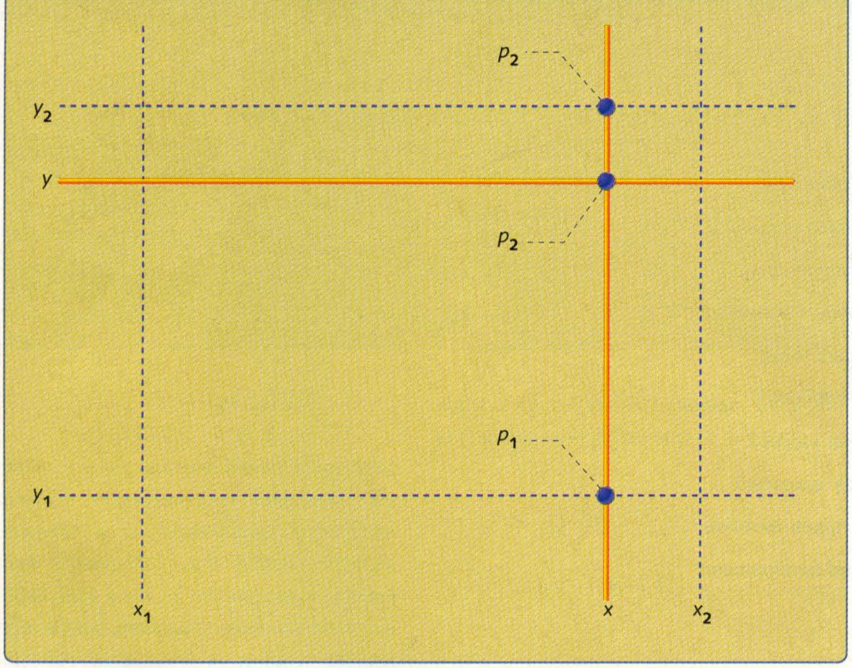
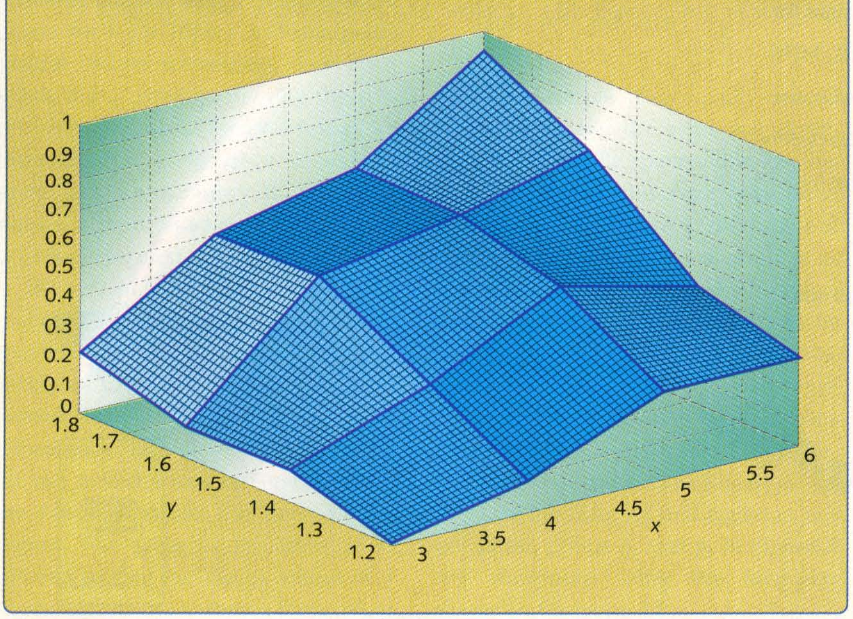


FIGURE 5 Linear breakpoint interpolation



The model developed using system identification techniques is a dynamic model, while the table interpolation methods presented in the previous sections are static function evaluation techniques. System identification and table interpolation methods are similar in that they are based on the use of measured data rather than an assumed set of mathematical relations as in the case of physics-based modeling.

Effort levels

This article has presented some commonly used techniques for the development of mathematical models of complex dynamic systems for simulation. We have also looked at the relevant questions that should be asked prior to beginning the development of a model of a dynamic system. The answers to these questions will help the model developer determine the approach and level of effort that are needed for developing a particular model. Proper application of these methods will lead to the development of mathematical models that realistically represent real world systems without wasting resources in the development of an unnecessary level of model fidelity. **esp**

Jim Ledin is a consulting engineer in Camarillo, CA. He can be reached by e-mail at jim@ledin.com.

This material is excerpted from the book Simulation Engineering, *which will be published in 2001 by CMP Books.*

References

1. Press, William H., Saul A. Teukolsky, William T. Vetterling, Brian P Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge, England: Cambridge University Press, 1992.
2. Juang, Jer-Nan. *Applied System Identification.* Upper Saddle River, NJ: Prentice-Hall, 1993.

SEAN BEATTY

Sensible Software Testing

To find and kill bugs, you must know where they live. You can use knowledge about the sorts of errors found in a program—and the risks they pose—to select the most effective testing strategies.

Many demands are placed on a software engineer's time. High quality, robust feature count, and low cost are often considered critical goals even though they generally compete against each other. In many development environments, time to market is critical, and the specification is in flux throughout the development process. Too often, testing gets whatever time is left between the point at which the code is finished and the date it must be shipped.

While it may be impossible to fix all the problems with the environment in which software engineers must work, it is helpful to quantify the challenges they face. This article aims to help the software engineer develop a practical approach to testing firmware. By understanding the goals of a given testing strategy and the associated costs, better decisions can be made as to what to test, and when.

Approach

To meet the goal of identifying a practical software testing strategy for a given project, I propose analyzing the problem along three lines:

- Identify the types of bugs common in embedded systems software
- Discuss the various methods used to find bugs
- Apply the "best" methods as part of a sound software development process

Bugs introduced into the software during the "coding" or implementation phase are quite common.

This approach sounds simple, but like many plans, the devil is in the details. The engineer needs to know the frequency of a certain type of bug's occurrence and the relative effect it has on the function of the system. Most of this article is devoted to elaborating on the various types of bugs so these issues can be better understood. It's only after developing a good scope of the problem that an engineer can develop an appropriate solution.

Once the problem is understood, various ways of uncovering software bugs are considered. This includes discussing the effectiveness of given methods in finding the different types of bugs. When all these topics are properly understood, the engineer is well on the way to implementing a sensible embedded systems software test plan.

Before proceeding further, I must clarify some of the terms used frequently in the following discussion:

- The words *bug, failure,* and *error* are used interchangeably. They all indicate some problem with the software
- The terms *subroutine, function,* and *method* are used synonymously to indicate some code that can be called
- Bug frequency is categorized in four groups: rare, less common, more common, and common
- Bug severity is indicated by one of four labels: non-functional, low, high, and critical

A taxonomy

When devising a plan to remove bugs from software, it helps to know what you're trying to find. Software can fail in many ways, and mistakes are introduced into the code from many different sources. Some bugs have greater repercussions than others, and almost all of them have consequences determined by the type of application and the domain in which it operates.

What follows is a catalog of errors found in embedded systems software. The list is long, yet important. Without understanding how software can err, it's difficult to find the potential errors. The relative frequency and severity is listed for each type of bug. Definitions for some of the terms used in the discussion appear in the sidebar.

Non-implementation error sources

Errors can be introduced into the code from an erroneous (or ambiguous) specification or an inadequate design. They can also result from hardware that doesn't operate correctly, or operates differently than specified or otherwise understood.
Frequency: All too common.
Severity: Ranges from non-functional to critical.

Implementation error sources

Bugs introduced into the software during the "coding" or implementation phase are quite common. These types of errors will receive emphasis in this article. There are many types of implementation bugs of varying severity. It helps to group them into general classifications based on some common elements. Arguably, some bugs could appear in more than one classification.

Algorithm/logic/processing bugs

Off by "1"

It's common to be "off by one" in a calculation. For example, a loop needs to execute 10 times, and a construction such as `for (x = 0; x <= 10; x++)` is used. This will execute 11 times, not 10 times. Another example: `for (x = array_min; x < array_max; x++)`. If the intention is to set `x` to `array_max` on the last pass through the loop, the software is in error.
Frequency: Common.
Severity: Varies, but is typically high, since the program doesn't operate as intended. However, in other instances this type of error may never be detected. For example: filtering a variable. If an average of the last 10 samples is intended, and instead nine samples or 11 samples are averaged, it's possible that a difference in the program's function will not be detected.

Parameter passing

Incorrect arguments or parameters may be passed to a subroutine. Examples include passing a financial number in dollars when a yen amount was expected or returning a temperature in degrees Celsius when the calling program is assuming Fahrenheit.
Frequency: Common only when many complicated function invocations are used.
Severity: Varies.

Return codes

Improper handling of return codes is another potential error source. Assuming the called function executes correctly and not checking for unexpected return codes can cause problems. Avoid using the same return code for different conditions. A programmer could misinterpret a return code, especially when using a routine developed by someone else.
Frequency: Common when using unfamiliar libraries or complicated functions with many return codes.
Severity: Varies.

Math overflow/underflow

Unless you're using a floating-point

RUPERT ADLEY

library, arithmetic overflow or underflow must almost always be checked. Fixed point and integer types can only hold numbers of a certain range. The result of an operation should be checked to ensure an overflow/underflow did not occur, before using the result in any meaningful way. Failure to check for overflow/underflow can result in data-sensitive problems that can be difficult to track down. If an overflow condition is detected, it must be handled in some appropriate way (often by limiting the data to the largest number that can be represented in the data type). These checks are unnecessary only when the input data is well known and it's impossible for the operation to ever overflow or underflow.

Frequency: Common where arithmetic operations are performed using integer or fixed-point math.

Severity: Generally high.

Logic/math/processing error

Of course, it's easy to make mistakes when implementing the logic of a program. Incorrect decision logic (`IF`, `THEN`, `ELSE`, `SWITCH`, `WHILE`, `GOTO`, and so on) grows common in complicated functions and deeply nested decisions. For example: `IF ((this AND that) OR (that AND other) AND NOT (this AND other) AND NOT (other OR NOT another))`. Boolean operations and mathematical calculations can also be easily misunderstood in complicated algorithms.

Frequency: Common.

Severity: Generally high.

Reentrance problem

If a section of code can be interrupted before it completes its execution, and can be called again before the first execution has completed, the code must be designed to be reentrant. This typically requires that all variables referenced by the reentrant routine exist on a stack, not in static memory. In addition, any hardware resources used must be manipulated carefully. If not, data corruption or unexpected hardware operation can result when the interrupted (first) execution of the routine finally completes.

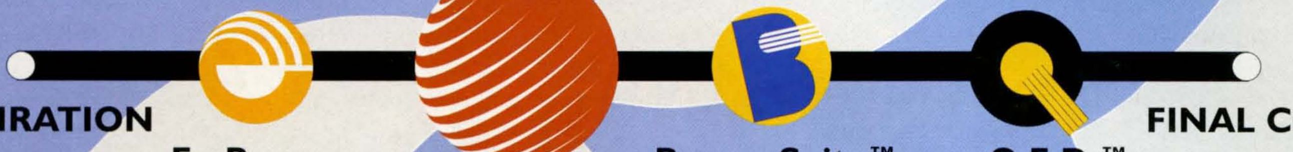
Frequency: Rare in embedded systems code, since most code is not reentrant.

Severity: Generally critical.

Incorrect control flow

The intended sequence of operations can be corrupted by incorrectly designed `for` and `while` loops, `if then else` structures, `switch case` statements, `goto` jumps, and so on. This causes problems such as missing exe-

The RTOS for Internet Appliances, Systems-On-A-Chip, and all other Embedded Systems



INSPIRATION

EmProve
Virtual Application Development and Prototyping (VAD-P™) environment

RTXC™
Internet Connected Multitasking Real-Time OS

BeaconSuite™
- Compilers
- Linkers & Locators
- Debuggers
- Simulators

Q.E.D.™
- JTAG Debugger
- 186 /386 ICE

FINAL CODE

The shortest distance between two points includes the ground we've already covered for you.

We have covered that ground with the most useful, flexible, scalable RTOS you can imagine for DSP, 32-, 16-, and 8-bit embedded applications.

It's what we do for a living - building it right so that you don't have to. You *could* write your own d.o.s. (dumb old scheduler), and *then* try your hand at integrating some file, graphics, and network components.

But why?

As a proven OS, **RTXC** has a pre-emptive scheduled multitasking real-time kernel with an extensive set of services and subsystems built in to assure top performance.

For example, **RTXCnet** includes a full array of networking components - from TCP/IP or PPP to web browser/server. All RTXC components are integrated to kernel resources and architected to fully utilize pre-emptive multitasking. We blow the competition's polling-based architectures away - and so will you! Just pick the networking components you need and RTXCnet arbitrates synchronization and interactions between your application, network stack and kernel.

RTXCfile32, our MS-DOS compatible file management subsystem can be used on any embedded system that uses RTXC - without regard to the target processor. We have drivers for CD ROM, floppy disks, hard disks, PCMCIA memory cards, and FLASH or RAM disks. Our drivers can be tailored to meet your most demanding file-oriented storage device needs.

Our **RTXCgraphics** subsystem endows your development effort with an existing portable graphic user interface library. Video device support is superlative, and fully supports the Rapid Application Development Standard.

Supported Processors

AMD: x86xx Family, ELAN 3/4/520C, K6
ARM: 6, 7/7T, 710, 9T, StrongArm
Hitachi: H8/300 Family
IBM: PPC 4/6
Infineon: 80C16x Family
Intel: 80x86/386/486 Real Mode,
80x86/386/486/Pentium Family Protected Mode
Mitsubishi: M16C
Motorola: DSP, StarCore, M.CORE, MPC500,
ColdFire 52xx/53xx, 68K Family, CPU 32 Family,
68HC16 Family, 68HC12 Family,
and 68HC11 Family
Philips: 80C51XA
ST Microelectronic: ST10
Texas Instruments: 320C3x/5x/20x/54x

www.embeddedpower.com

Embedded Power
CORPORATION

U.S. 281-561-9990
email: info@embeddedpower.com
Europe: +44 (0) 1256 474448
email: eurosales@embeddedpower.com
© 2000 Embedded Power Corporation
Trademarks are the property of their respective holders.

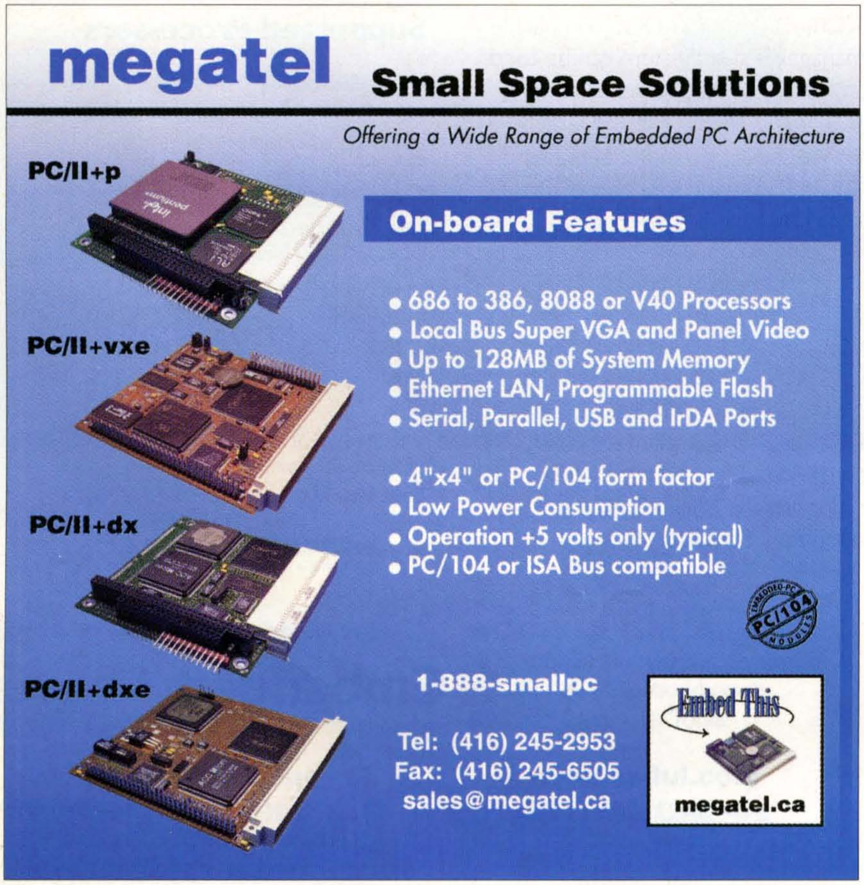
Initializing a variable properly is only the first step in using it correctly. It's generally a bad idea to use a variable for more than one purpose.

cution paths, unreachable code, incorrect control logic, erroneous terminal conditions, unintended default conditions, and so on.
Frequency: Common.
Severity: Varies from non-functional to critical.

Data bugs

Pointer error
Pointer errors border on the famous (infamous?). Every programmer who has used pointers with any frequency is familiar with the mysterious symptoms with which a bad pointer manifests itself. Pointer problems can be notoriously difficult to track down. Some coding guidelines I've seen even recommend avoiding all pointer usage, if possible, to avoid these nasty bugs.

Pointer errors are often more common when certain types of structures are used in the code. Doubly linked lists make heavy use of pointers, so it's easy to point to the wrong node or link to a NULL pointer. When using look-up tables or lists, take care to properly increment any pointer used to step through the table or list. General pointer problems of de-referencing a NULL pointer or pointing to the wrong thing grow more common as the number of nested references increase, for example `**array_of_ptrs_to_ptrs[*index_ptr]`. A bad function pointer can cause the wrong subroutine to be called.
Frequency: Common in languages that support pointers, such as C.
Severity: Almost always high or critical.



Indexing problem
Where "C" programmers use pointers, assembly language programmers use index registers. Index registers (or similar types of registers in other architectures) provide the same type of indirection useful for table look-up, walking through lists, trees, and other data structures, and calling a routine determined at run-time. They also have the same potentials for error.

High-level language programs often make heavy use of arrays. Many times, strings are stored as arrays of characters. Individual elements within an array are identified with an array index. Accessing the wrong element within an array is another example of an indexing problem.
Frequency: Common.
Severity: Almost always high or critical.

Improper variable initialization
Sometimes improper initialization can be obvious, as when reading a variable that has never been written. Other times it's more obscure, such as reading a filtered value before the proper number of samples have been processed.
Frequency: Less common.
Severity: Often low, but it varies.

Variable scope error
To get the expected results, the correct data must be processed. The same name can be applied to different data items that exist at different scopes. For example, an automatic variable can coexist with a static variable of the same name in that file. Different objects instantiated from the same class refer to their members with the same name. When pointers are used to reference these objects, it becomes even easier to make a mistake.
Frequency: Less common.
Severity: Generally low to high.

Improper data usage
Initializing a variable properly is only the first step in using it correctly. It's generally a bad idea to use a

Accelerated Technology INCORPORATED

There is only one conclusion.

Only one RTOS provides all the benefits you need. Accelerated Technology combines a level of service that is unmatched, an affordable pricing model and open source benefits with a vast, tightly integrated embedded product line offering.

Nucleus MNT - Windows-based rapid prototyping environment
Nucleus EDE - intuitive embedded development environment based on Microsoft Developer Studio™
Nucleus PLUS - robust, scalable, multitasking real-time kernel
Nucleus NET - complete TCP/IP networking protocol stack
Nucleus UDB – portable source level debugger
SurroundView and Nucleus ProView - revolutionary profiling tools that introduce a new level of application and OS monitoring
Nucleus WebServ - a tightly integrated, internet-enabling embedded web server
Nucleus WebBrowse - a compact and tightly integrated embedded web browser
Nucleus GRAFIX - portable embedded graphical user interface

These Nucleus embedded products support a wide range of processors:

- MCF5206, 5307 • M•CORE
- PPC40x, 5xx, 60x, 7xx, 8xx, 82xx
- 680x0, 683xx
- ARM6/7/9, AEB, Atmel 40400, CL7110, 7111, 7209, Samsung SNDS100
- SA100-285, SA1100, SA1110 • ARC
- SH1, SH2, SH3, SH4, SH3-DSP, H8S/2000, H8/300H
- IDT RC3081, RC4640/50, R5000, RC32364
- LSI LR 33000, 40xx, 410x, 64008, Lexra 4180, NEC 41xx, 4300, 5000, NKK 4650, Toshiba TX3904, 3927
- x86 RM/PM • TriCore • C167

For additional information on the complete Nucleus product line and how it can greatly diminish your time-to-market needs, royalty-free, please contact us at:

Phone: 1.800.468.6853
Address: 720 Oak Circle Dr. E. Mobile, AL 36609
Email: info@atinucleus.com
Internet: www.atinucleus.com

NUCLEUS

All You Need in an RTOS. Royalty Free.

variable for more than one purpose. It's too easy to modify it in one place for one reason and then alter it again in another place for a different reason—undoing the first change. This is generally only a problem in smaller systems that make heavy use of global data and are short on RAM. Another improper data usage involves modifying data but never storing it or testing it. This is unlikely to perform as intended. Storing a data value in the wrong units is also a serious problem, for example calculating a result in degrees Fahrenheit and storing it in a temperature variable that expects degrees Celsius.

Frequency: Common.

Severity: Varies.

Incorrect flag usage

This is a specific type of incorrect data usage, but it occurs commonly enough to merit its own category. Flags are typically used to communicate between various parts of a program, are generally global in scope, and are almost always static. When an RTOS is used, this communication function may be handled with a semaphore. Every flag should be set, cleared, and tested at some point in the program. Missing one of these three generally indicates an error. A flag may inadvertently be used for more than one purpose, or used to indicate more than one condition. This is also typically an error.

Frequency: Common where hard-coded constants are used to represent the bit-position of a flag within a flag-word, instead of using symbolic constants. Less common when using bit-fields as part of a structure.

Severity: Varies.

Incorrect address

Most of the time, a bad address is the result of an incorrect pointer. Nevertheless, it is possible to hard-code a bad address into the code. This generally happens only when the memory subsystem or some peripheral changes.

Inadvertent overflow of a data type can produce some very strange symptoms.

Some terms used

Software: Code that is intended to run on a microprocessor or microcontroller in an embedded system. This code typically resides in ROM or EPROM of some sort. The code may be written in assembly or a high-level language.

Non-functional: A software change that does not affect the object code in any way. The change can be made at the customer's discretion. The change only corrects comments, other documentation, or a deviation from software guidelines.

Low: A software change that should be made whenever convenient. The customer "can live" without the change in the short-term. A release will not be rescheduled due to this change alone. This change may be a "better" way to implement a function.

High: A software change that should be made as soon as possible. The change should be in the next scheduled release. This change is required to meet a customer specification.

Critical: A software change that must be made as soon as possible. A new release of the code must be scheduled. This change may be required to satisfy safety or legal issues.

Logic: As used here, logic refers to the logical implementation of a function, not simply Boolean operations (although they are included). Put another way, the logic involves the sequence of operations necessary to process the function's inputs to develop correct function outputs.

Filter: A software filter performs a weighted average on data, smoothing out the variations in the data's sequential values. This is analogous to the way an analog filter reduces the fluctuations in a varying voltage, producing a more average voltage. The "heavier" the filtering, the less effect a variation will have on the filtered result. A software filter uses the most recent data value, as well as a certain number of previous values (for the same data variable) in various weights, to produce a filtered data result. Examples:

```
filtered_data = (old_data/0.9) + (new_data/0.1);
filtered_data = (oldest_data*12 + old_data*3 + new_data)/16.0;
```

Atomic: An operation that cannot be divided. This implies the operation cannot be interrupted, since the currently executing instruction is always completed before a pending interrupt is recognized. Some processors provide a special read-modify-write instruction which is atomic.

FMEA: Failure Mode and Effects Analysis. (Sometimes called FMECA: Failure Mode Effects and Criticality Analysis.) A systems safety engineering technique that attempts to discover all the problems that could occur in the use of a device. The probability of an occurrence and the seriousness of the risk involved are assigned to each possible failure. Then a solution such as improved design, testing, labeling, or training is proposed to mitigate that problem.

Frequency: Rare.
Severity: Generally high to critical.

Data/range overflow/underflow

These problems can be hard to detect. Inadvertent overflow of a data type can produce some very strange symptoms. Most of the time, the program executes as expected. Then occasionally it goes haywire, seemingly unexplainably. Errors of this type include passing a parameter that is out of bounds, and storing the result of a calculation in a data type not large enough to hold the data. Of course, the more strongly typed the language, the less of a problem this becomes.

Frequency: Common in assembly language programs, and high-level language programs that target small (8-bit) processors. In the latter, the types may be smaller than expected. For example, is it safe to assume an integer is 16-bits wide?

Severity: Low to critical. Sometimes the effects can go unnoticed.

Signed/unsigned data error

A mix of signed and unsigned data types can easily lead to calculations that produce wrong results. Assembly languages have different branch instructions used after comparing signed and unsigned data. Using the wrong branch instruction is often a critical error. When mixing signed and unsigned types, care must be taken to understand the sign of the result and store it in the proper data type. Mixed sign arithmetic can easily overflow the data types used in the calculation if not handled properly.

Frequency: Common in assembly language programs, or when using fixed-point (integer) math. Not a problem where floating-point math is used exclusively.

Severity: Varies, generally high to critical.

Incorrect conversion/type-casting/scaling

Converting a data value from one representation to another is a common operation, and often a source of bugs. Data sometimes needs to be converted from a high-resolution type used in calculations to a low-resolution type used in display and/or storage. Conversion between unsigned and signed types, and string and numeric types is common. When using fixed-point math, conversion between data types of different scales is frequent. Typecasts are useful to get data into whatever representation is needed, but they also circumvent compiler type-checking, increasing the risk of making a mistake.

Frequency: Common in programs that are more complicated.

And Came Up With This Great ARM Solution.

ThreadX

- No Royalties
- Fastest RTOS for ARM
- Smallest memory requirements
- Complete ThreadX source code
- Preemption-threshold™
- Picokernel™ architecture
- Optimized ARM interrupt handling
- IRQ and FIQ interrupt support

MULTI 2000

- Best IDE for ARM Development
- Best ARM Debugger
- Best ARM Compiler
- Seamless ARM/Thumb interworking
- CodeBalance™ tuning
- ThreadX-aware Debugging
- Run-time Event Analyzer for ThreadX
- ThreadX C Library Integration
- Complete ARM simulation

What if the best RTOS in the market combined with the best Compiler, Debugger, and IDE? You'd get seamless integration, no hitches, no snags, and no surprises.

Well now it's happened. Express Logic's ThreadX® kernel and Green Hills Software's MULTI® 2000 IDE have come together to create the industry's most powerful development and run-time solution for deeply embedded applications.

With ThreadX you get a royalty-free, source-code RTOS, and state-of-the-art features like preemption-threshold and picokernel design. The advanced MULTI 2000 IDE gives you an Optimizing C/C++ compiler, ThreadX-aware debugging, complete ARM simulation, and the run-time EventAnalyzer for ThreadX. Together, ThreadX and MULTI 2000 are easily the best ARM solution available today.

For a free demo CD of the MULTI 2000/ThreadX solution, call Green Hills Software at 805.965.6044 or visit us on the web at www.ghs.com/armsolutions.

www.ghs.com/armsolutions

Best support for the ARM Processor

ARM6
ARM7
ARM7M
ARM7TDMI
ARM7500fe
ARM8
ARM9
ARM9E
StrongARM

Other Processors Supported:

- Win32 • PowerPC • X86
- ColdFire/68K • Hitachi SH
- MIPS • NEC V8xx • SPARC
- M-Core • TriCore

From the brains of

Express Logic, Inc. and Green Hills Software, Inc.

Green Hills
SOFTWARE, INC.

ph. 805.965.6044 • sales@ghs.com • www.ghs.com/armsolutions

Severity: Varies, low to critical.

Data synchronization error

Many real-time embedded systems need to share data among separate threads of execution. For example, suppose an operation that uses a number of different data inputs is performed. This operation assumes these data are synchronous in order to perform its processing. If the data values are updated asynchronously, the processing may be using some "new" data items with some "old" data items, and compute a wrong result. This is especially true if a control flag is used to interpret the data in some way. Some embedded systems use a serial port to send a "system snap-shot" of the critical data items in response to an asynchronous request. If the data items in the snapshot are not updated synchronously, the snapshot may contain a mix of some current information and some old information.

Frequency: Less common.

Severity: Low to high.

Real-time bugs

Interrupt handling

It's critical to be able to handle all interrupts that the system will ever receive. Receiving an unexpected interrupt without being able to handle it is usually disastrous. For this reason, even interrupts that are not expected to occur should still be handled with an "unexpected interrupt" handler, just in case. Vectors for every interrupt that your processor could receive, either intentionally or inadvertently, must be present and must point to the correct handler.

An equally disastrous mistake is an incorrect return from an interrupt handler. Most processors have separate instructions to "return from subroutine" and "return from interrupt." If you use the wrong instruction to return from the interrupt, you will corrupt the stack (by not unstacking registers that were pushed onto the stack automatically when the interrupt was acknowledged). High-level languages often use a special keyword to indicate to the compiler that the "return from interrupt" instruction should be used with a particular function.

Frequency: Rare.

Severity: Critical.

Interrupt suppression

Most programs that use interrupts also suppress them around critical sections of the code. Receiving an interrupt during a critical section may cause the program to miss some time-related specification, corrupt data, mishandle external hardware, and so on. Therefore, it's critical to ensure that all sections of code that need interrupt suppression have it. Often, this is system-specific knowledge that should be well documented.

One situation that doesn't require detailed system knowledge involves data corruption. Whenever data is written by both an interrupt service routine (ISR) and another place in the program, special care must be taken. Any read-modify-write on the data must be atomic, or interrupts must be suppressed around that section. When using a high-level language, be aware that writes to a multi-byte type may not be atomic. This may not be obvious by looking only at the source code. If an interrupt occurs between the read and modify cycles, the modification may be made inappropriately, since the data may have changed. If an interrupt occurs between the modify and write cycles, the new data updated by the ISR will be overwritten. Even if the data is not modified in this latter situation, the ISR-updated data could be overwritten. This type of read-don't-modify-write is sometimes done to refresh or test memory, or in certain peripheral interfaces.

Personal Best

There's a feeling about achieving a goal, or simply reaching a new level of performance that's hard to put into words. It's not about winning or beating the next guy necessarily, just about knowing that you've done more than you've ever done before.

HI-TECH C users get that feeling often - the feeling of being "in the groove", the realization that not only did you meet your deadline, but you did it easily, without stress.

If you program embedded microcontrollers, and you want to raise your performance level, get HI-TECH C. World-beating performance from a tool that gets the job done without getting in your way, and backed by support our competitors can't believe - that's what *you* need to achieve your personal best. Call us today, or visit our web site for demos and information.

www.htsoft.com

HI-TECH Software
7830 Ellis Rd.
Melbourne FL 32904
Ph 800 735 5715
Fax 407 722 2902

HI-TECH
SOFTWARE

FIGURE 1 Effectiveness of various bug-finding techniques

	Individual code walkthrough	Group code walkthrough	Step-by-step execution of code	Structural (white box) testing	Functional (black box) testing	Verification	Stress/performance testing	Writing test procedure	User report
Non-implementation errors	⬜	⬜	🟨	🟨	🟩	⬜	🟩	🟨	🟨
Off by "1"	🟩	🟩	🟦	🟩	🟨	⬜	⬜	🟩	🟨
Parameter passing	🟩	🟩	🟩	🟦	🟩	⬜	⬜	🟩	🟩
Return codes	🟨	🟩	🟨	🟩	🟩	⬜	⬜	🟩	🟨
Math overflow/underflow	🟦	🟦	🟨	🟩	⬜	⬜	🟨	🟩	🟩
Logic/math/processing error	🟨	🟩	🟨	🟦	🟨	⬜	⬜	🟩	🟨
Reentrance problem	🟨	🟨	⬜	🟨	⬜	🟩	🟩	🟩	🟨
Incorrect control flow	🟨	🟨	🟩	🟦	🟨	⬜	🟨	🟦	🟨
Pointer error	🟨	🟩	🟨	🟩	🟨	🟨	🟨	🟩	🟨
Indexing problem	🟩	🟩	🟨	🟨	🟨	🟩	⬜	🟩	🟨
Improper variable initialization	🟨	🟩	🟩	🟩	⬜	🟩	⬜	🟩	⬜
Variable scope error	🟨	🟩	🟨	🟨	🟨	🟨	⬜	🟩	⬜
Improper data usage	🟨	🟩	🟩	🟨	🟨	🟦	⬜	🟩	🟨
Incorrect flag usage	🟨	🟩	🟨	🟨	🟨	🟦	⬜	🟨	🟩
Incorrect address	🟨	🟩	🟩	🟨	🟩	⬜	⬜	⬜	🟩
Data/range overflow/underflow	🟨	🟩	🟨	🟩	🟨	🟨	🟨	🟩	🟨
Signed/unsigned data error	🟨	🟩	🟨	🟩	🟨	🟦	⬜	🟩	⬜
Conversion/type-casting/scaling	🟨	🟩	🟨	🟩	⬜	🟨	⬜	🟩	🟨
Data synchronization error	🟨	🟩	⬜	⬜	🟨	🟩	🟨	🟨	⬜
Interrupt handling	🟦	🟦	⬜	⬜	⬜	🟦	🟩	⬜	🟨
Interrupt suppression	🟨	🟩	⬜	⬜	⬜	🟦	🟩	⬜	⬜
Task synchronization	🟨	🟨	⬜	⬜	🟨	🟩	🟩	🟨	⬜
Stack overflow/underflow	🟨	🟨	🟨	⬜	🟨	🟦	🟩	⬜	⬜
Other stack errors	🟩	🟩	🟩	🟩	🟨	🟨	⬜	🟩	⬜
Version control errors	⬜	🟨	⬜	⬜	🟨	⬜	⬜	⬜	🟦
Resource sharing problem	⬜	🟨	⬜	🟨	🟨	🟩	🟩	🟨	⬜
Resource mapping	🟨	🟩	⬜	⬜	🟦	🟦	🟦	⬜	🟨
Instrumentation problem	🟨	🟨	⬜	⬜	🟦	🟨	🟩	⬜	⬜
Syntax/typing	🟨	🟩	🟩	🟩	🟨	🟨	⬜	🟩	🟨
Interface	⬜	🟨	🟨	🟨	🟦	🟩	🟨	🟩	🟨
Memory allocation/deallocation	🟩	🟦	🟨	🟨	⬜	🟦	🟩	🟨	⬜
Peripheral register initialization	🟨	🟨	🟨	🟨	🟩	🟩	🟨	🟨	🟨
Watchdog servicing	🟨	🟨	⬜	🟨	🟨	🟦	🟩	⬜	⬜

Effectiveness: Low ⬜ Medium 🟨 High 🟩 Very High 🟦

Too much of a good thing can be bad. If interrupts are suppressed too long, timing deadlines may not be met. Or a system clock may not keep time "correctly." Processors typically inhibit all interrupts of priority equal to or lower than the current interrupt priority. Therefore, when calculating the maximum interrupt suppression that your system could ever encounter, it's not enough to simply look for "interrupt disable" and "interrupt enable" instructions. You must also account for the time spent within any ISR.

Frequency: Less common.
Severity: Critical.

Task synchronization

Tasks must be synchronized correctly. Some operations must wait for others to occur first or other tasks to complete. One task may acquire raw data; another may process this data as a set; still another may make control decisions on the processed data values. Proper synchronization is sometimes implemented by relying on flags or semaphores to control task execution. Other tasks are synchronized by scheduling them to execute at regular intervals. If one task doesn't finish in time, a second task that depends on its completion may fail. Other task-related problems include race conditions and priority inversion problems.

Frequency: Less common.
Severity: Varies.

System bugs

Stack overflow/underflow

"Don't blow your stack!" Although this expression usually refers to something quite unrelated to embedded systems, it's quite applicable here. Pushing more data onto the stack than it is capable of holding is called overflow; pulling more data from the stack than was put on it is called underflow. Both result in using bad data, and can cause an unintended jump to an arbitrary address—very bad. The stack pointer can also be directly manipulated on many processors, and is sometimes so used to quickly generate temporary variable space on the stack.

Many high-level languages offer no direct way to manipulate the stack. However, deeply nested subroutines with many parameters can still cause an overflow. Therefore, it's important to ensure that the worst case stack depth generated by a program can never exceed the stack allocated. Multi-tasking systems complicate this analysis, since each task needs its own program stack. In addition, interrupts require stack space in order to save the value of the processor's registers. Moreover, the deepest stack needed by any interrupt must be added to each

task's stack space (assuming any task can be preempted by any interrupt). It's easy to see how quickly the program's stack space can be consumed in these types of designs.
Frequency: More common in assembly language programs and complicated designs.
Severity: Critical.

Other stack errors
Other stack errors can corrupt data. For example: pushing the X-register then Y-register onto the stack to save their values, but pulling them off the stack in the wrong order.

A stack imbalance occurs when not all the registers pushed onto the stack at the beginning of a routine are pulled off the stack before the routine returns (or vice-versa). This causes execution to jump to an arbitrary address.
Frequency: Less common. Generally only occurs in assembly language routines.
Severity: Stack imbalances are always critical, and generally produce immediate and dramatic failures. Data problems vary in severity.

Version control error
It doesn't matter how good your last bit of code was if it didn't get included in the build. Including the version of the file that still has the bug produces another bug report. Including a version that is now incompatible with the latest hardware may produce many bug reports! Version control grows in importance as the complexity of the software project (read: the number of people involved in the software) grows.
Frequency: Common only in complicated systems with many files and many developers. This problem can become more difficult in distributed development environments.
Severity: High to critical.

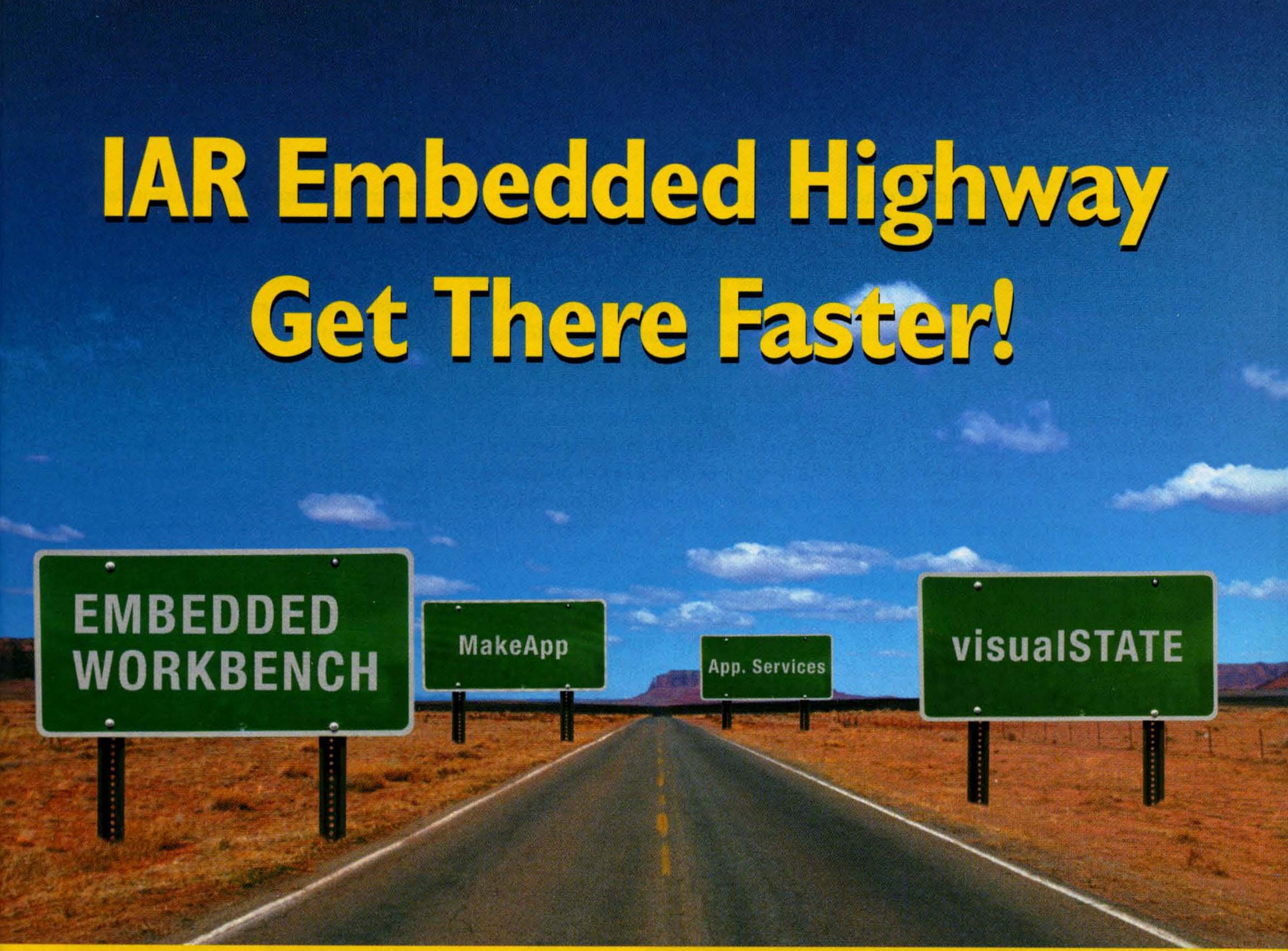
Resource sharing problem
Resource sharing is common in most embedded systems at some level. Wherever sharing occurs, strict rules for using the resource cooperatively must be defined and followed to avoid conflicts. Ignoring a mutual exclusion semaphore can corrupt data. Two different tasks that both use the same peripheral must cooperate. For example, an analog multiplexer may be used to direct one of a number of different inputs to a single A/D converter. If one task alters the mux setting to measure a given signal and another preempts it and sets the mux to pass a different signal, when control returns to the first task it will be measuring the wrong signal.
Frequency: Less common.
Severity: High to critical.

Resource mapping
Some microcontrollers allow the peripheral registers and memory to be mapped to many different locations. Some applications use different mappings for various purposes. Get this wrong, and it's likely the code won't even run.

Less obvious is mapping the code or initialized data to a RAM area during development, where it's easy to modify. This is common when downloading the code into instrumentation of some sort. If the code isn't remapped before burning it into EPROM (or worse yet, releasing the ROM mask), your data or code becomes whatever happens to be in the RAM after power-up!
Frequency: Rare.
Severity: Critical.

Instrumentation problem
Sometimes a software bug is not actually a problem with the software at all. Instrumentation generally alters the behavior of the system, albeit in very small, subtle ways. Sometimes problems disappear when the emula-

IAR Embedded Highway Get There Faster!



IAR Embedded Workbench™

A superior C cross compiler package offering advanced embedded C/C++ optimization techniques

- Supports the following MCUs:

Atmel	AVR, AT89
Hitachi	SH, H8S, H8/300H, H8/300
Intel	8051, 80x96, 80251
Microchip	PICmicro 16/17 and 18
Mitsubishi	M16C/60/20, MC80, 740
Motorola	6811, 6812, 6816
National	COP8, CR16
NEC	V850, 78K0, 78K4
Sharp	SM6000, SM8500
TI	MSP430
Zilog	Z80, Z180

and more... for complete list, check www.iar.com

- Different Architectures, One Solution: The same user friendly environment applies for all targets.

IAR visualSTATE®

A UML-compliant graphical state machine tool generating micro-tight code for any embedded MCU.

- Very small code footprint (A breakthrough patented technology)
- Table driven code (compact & fast)
- Uniquely suited for low-end 8/16-bit MCUs
- Generates MCU & Compiler independent code (works for 8/16/32/64-bit MCUs)
- 100% formal verification of state model (a second patent)
- Allows state level debugging, breakpoint setting, variable/action watch and graphical animation of model
- Static & dynamic analysis with regression testing
- Automatically generates complete design documentation in 'rtf' format.

IAR MakeApp™

The Device Driver Wizard. Automatically generates fully tested C initialization code for MCUs

- Visual configuration of device drivers.
- Supports the following targets

Atmel	AVR
Hitachi	SH, H8S/300H, H8/300
Mitsubishi	M16C
Toshiba	TX RISC

for up-to-date target support check www.iar.com

- Checks for illegal settings & resource conflicts

IAR App. Development Services

- Contact IAR to speed your next embedded development project.

For a free demo & more information, please contact IAR Systems.
Email: info@iar.com, Tel: 1-800-427-8868 Or visit **www.iar.com**

IAR SYSTEMS
From idea to target.
www.iar.com

IAR Systems (US HQ & West Coast), One Maritime Plaza, San Francisco, CA 94111 Tel: 415-765-5500, Fax 415-765-5503
IAR Systems (East Coast) 2 Mount Royal, Marlborough, MA, 01752, Tel: 508-485-2692, Fax 508-485-9126

tor is connected, and other times they only appear when the emulator is used. Reported bugs could also be a result of improper use of the instrumentation.
Frequency: Less common.
Severity: Low.

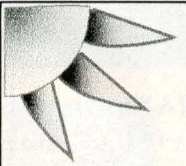
Other bugs

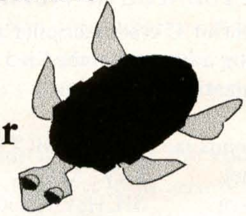
Syntax/typing
Compilers can be a considerable help in checking the accuracy of our typing. For example, some will issue a warning when assignments are made within a conditional expression: writing `if (a=1)` when `if (a==1)` was intended. But other errors defy detection. No compiler will warn you about misspelling a variable name when the misspelling is also a valid symbol (for example, `hiRes_speed`, instead of `hiRev_speed`).
Frequency: Less common.
Severity: Varies.

Interface
Complex interfaces are a common source of errors. Interfaces can be external to the processor or internal. The modules interfaced to could be hardware or software. Documentation that is missing, incomplete, ambiguous, or incorrect is often to blame. Hardware or software changes that aren't properly communicated to all the appropriate people also produce interface problems. These types of bugs include protocol errors and timing or sequence problems. Examples: incorrect EEPROM erase/write sequence, improper use of LCD controller chip commands, wrong sequence in reading/writing serial communication interface registers.
Frequency: Common.
Severity: High to critical.

Memory allocation/deallocation
Using memory management routines can greatly simplify the efficient use of available memory. It can also be an added source of errors. Examples: not checking for successful allocation before using the memory, not freeing memory when it's no longer needed (memory leak).
Frequency: Common only with high level languages, and only when using routines such as `malloc()` and `free()`. Less common with languages that do more memory management automatically (use constructors, destructors, and references).
Severity: Varies. Sometimes a small memory leak may go unnoticed. Not checking an allocation before using the memory can crash the system.

Peripheral register initialization
Most embedded systems have peripheral hardware devices that they use to perform some necessary work. These peripherals often have many different





modes of operation, increasing the number of applications for which they're useful. This can complicate the initialization and use of these devices, producing another source of errors.
Frequency: Less common.
Severity: Critical.

Watchdog servicing
Watchdog timers help ensure that if something in the system goes exceptionally wrong, it will fail in a safe, or at least predictable, manner. Most software FMEAs make use of watchdog timers to mitigate risks. However, with every added complexity comes yet another potential source of problems. Servicing the watchdog timer must be done properly and at the right time. The watchdog must be enabled, and set to timeout at the correct interval. The watchdog servicing must be guaranteed to occur frequently enough, under every correctly operating scenario, to prevent a timeout. Otherwise, the device intended to mitigate serious problems becomes a source of them. This implies that a thorough understanding of the timing characteristics of the system is necessary to ensure proper use of the watchdog timer. One last note: some programmers have used a periodic interrupt to ensure that the watchdog servicing is done on time. As long as the periodic interrupt is at a higher priority than whatever is going wrong, this effectively prevents the watchdog from "watching" the code, rendering its service useless.
Frequency: Less common.
Severity: Critical.

Finding hidden problems

This long list of errors begs the question "How do I find these potential bugs?" (Perhaps a better question is "How do I *prevent* them?" That's a topic for another article.) Many techniques and tools can be used to find bugs. Some of these are more expensive than others are, both in time and material cost.

Costs can only be described relative to each other. An emulator typically costs more than a simulator. However, there's a big difference in cost between an emulator for an 8-bit microcontroller and an emulator for a 32-bit DSP. For the purposes of this article, costs will be grouped in four categories: none, low, moderate, and high. General effectiveness of a testing technique will be categorized as either low, medium, high, or very high. The effectiveness of a particular technique for a specific software error type is given in Figure 1.

Individual code walkthrough
I realize it's probably not accurate to call this a test, but it is so effective I would be remiss not to mention it here. No other technique is as effective at identifying bugs as a good walkthrough. This starts with the individual programmer carefully examining his code for potential mistakes, omissions, misunderstandings, and adherence to the project coding standards. This is best done many hours (if not a day or more) after the code is originally written. That provides the opportunity for a fresh perspective. This walkthrough is best done before the engineer tests his code on the target.
Cost: Time—very low, money—none.
Effectiveness: Very high

Group code walkthrough
A good walkthrough can find more bugs than any other single activity. Conducting group walkthroughs is as much art as it is science. Group and team social skills (or lack thereof) become apparent. The goals of a group code walkthrough include finding potential problems, ensuring adherence to the software design, looking for subtle effects on the rest of the system, and identifying improvements. Ego has no place in this activi-

ty. It must be okay to have others identify your mistakes (and vice versa). The best walkthroughs are short, and done frequently.
Cost: Time—low, money—none.
Effectiveness: Very high. Affected by the effort, experience, and attitudes of the review team.

Step-by-step execution of code
This type of testing attempts to find problems not just by walking through the code, but by executing every line of the code. This code execution can be done on a simulator or on the target. Code execution can be controlled with an in-circuit emulator or observed with a logic analyzer. Sometimes every branch and condition is executed, not just every line of code.

The purpose of this type of testing is to observe the correct operation of the code. This implies that the correct operation is well understood, if not documented. The goal is not to find particular types of errors, or to observe particular behaviors. This type of testing is sometimes done on a single module or file of source code. Other times it's performed on the entire software application.
Cost: Time—can be high, especially when the entire system is being examined. Always tedious, since most of the code operates perfectly. Money—low to moderate. Simulators can be inexpensive, but ICEs can be quite expensive.
Effectiveness: Moderate, depending on the effort of those performing the test, and how well the intended behavior is understood by them.

Structural (white box) testing
Sometimes called "glass-box testing," this activity uses an unobstructed view of how the code does its work. This type of test is usually performed on a single unit of the software at a time. Test procedures are written to exercise all the important elements of the code under test. This may include exercising all the paths in the module. It often involves many executions of the same code, with different values of data. Boundary conditions are typically exercised. This test is also used to determine the consistency of a component's implementation with its design.
Cost: Time—high. It takes time to write the procedures, and examine what is critical to test. Of course, once the test procedures are written, they can be reused to retest the same module later when minor modifications are performed. Money—depends on how the testing is done. Simulators are less expensive than ICEs.
Effectiveness: high. If a module passes a

CMP MEDIA PRODUCT CATALOGS

Complete Embedded Product Information

Available Now!

ARM Development Guide
AMD Fusion E86 Catalog
CAN Solutions Directory
DSP Source Guide
Embedded Internet Source Guide
8051 Product Directory
Flash Memory and Components Source Guide
Hitachi Development Tools Catalog
IP Catalog for System-on-a-Chip Design
MIPS RISC Resource Catalog
Philips 80C51 + XA Development Tools
PowerPC Resource Guide
Siemens Microcontroller and DSP Development Tools Guide
USB, PCI and CompactPCI Connectivity Solutions and PC/104—PC/104-*Plus* Catalog
VantageISI Partners Catalog
X86 Catalog: including 586 & 686 Derivatives
Windows® CE Product Catalog
Windows® Embedded Catalog

ARM DEVELOPMENT GUIDE 1999-2000
1999 CAN SOLUTIONS
1999 DSP SOURCE GUIDE
Embedded Internet Source Guide 1999
8051 1999 EDITION PRODUCT DIRECTORY
1999 FLASH MEMORY AND COMPONENTS SOURCE GUIDE
1999 IP Catalog for System-on-a-Chip Design
MIPS RISC RESOURCE CATALOG
80C51+ & XA DEVELOPMENT TOOLS CATALOG
PHILIPS
HITACHI Microcontroller Development Tools Catalog
PowerPC RESOURCE GUIDE
USB, PCI and Compact PCI Connectivity Solutions Catalog
Vantage ISI Partners Catalog
Integrated Systems, Inc.
WindowsNT Embedded Product Catalog
WINDOWS CE PRODUCT CATALOG
1999 X86 CATALOG

3 easy ways to get your FREE product catalogs:

1. pick them up the Embedded Systems Conferences
2. order online through www.embedded.com
3. call 1-800-500-6815 in the U.S., 1-785-841-1631 outside the U.S.

Check out all the product catalogs at www.embedded.com

Additional charge applied to multiple orders

CMP
OEM Group

This long list of errors begs the question "How do I find these potential bugs?"

thorough white box test, the level of confidence is high that it won't cause problems later.

Functional (black box) testing

In functional test, the program or system is treated as a black box. This implies the tester has no knowledge of what's in the box, only its inputs and outputs. The system is exercised by varying the inputs and observing the outputs. This type of test is usually performed on the entire software system. In complex applications, the software is broken down into components or subsystems, which are tested individually. All the individual parts are then brought together (usually one at a time) and the integrated system is tested.

Test procedures are usually written to describe the expected behavior in response to a given environment and input stimulation. In the absence of formal test procedures, some engineers simply go through the system or software requirements document and make sure the system behaves as specified. The tests compare the software's actual behavior to its specified behavior without regard to the software implementation.

Cost: Time—medium. This type of testing is almost always done to one degree or another. Ultimately, the customer will exercise all the features of the program; it's better to find any obvious bugs before he does. Money—varies depending on how easy it is to manipulate the inputs and observe the correct outputs. Sometimes the entire test can be done stand-alone; other times, custom equipment must be constructed in order to verify correct behavior.

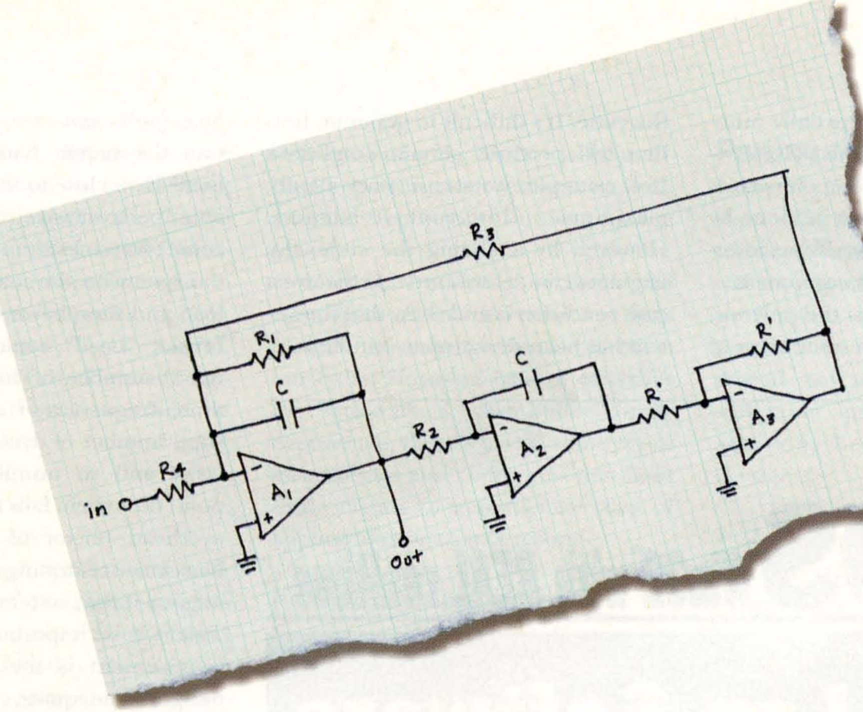
Effectiveness: Medium. Many parts of the system are difficult to exercise with a black box test. Error conditions (especially due to hardware failure) can be difficult, if not impossible, to generate. Some combinations of inputs are difficult to produce, especially those with unique timing characteristics.

Verification

Different organizations mean different things when they use the term verification. Some use "verification" and "validation" interchangeably; others make a clear distinction between the terms. The IEEE defines validation as "the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." This type of activity is described in the previous section. Verification, on the other hand, is defined as "(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness."

In this article, I use the term verification to refer to the detailed analysis of software, independent of its function, to determine if common errors are present. It is generally performed on the code as a whole, not on individual units. It's very white box-like, in the sense that verification examines the structural integrity of the code, and not functionality. Examples of typical verification checks include: stack depth analysis, proper watchdog timer usage, power-up/power-down behavior (in the sense of proper initialization and clean up), singular use of each variable, and proper interrupt suppression.

Cost: Time—medium. Some checks



Analog spoken here.

In this digital world, analog engineers often tell us they feel misunderstood. When the talk around the table is in ones and zeros, they're thinking in terms of currents and voltages. They say it's almost as though if you're doing analog, you're living in a different place. And speaking a different language.

If this is *your* world, we've got some good news: now you have a place you can feel right at home. We call it Planet Analog. And it's built just for you.

Planet Analog is a place on the Web devoted entirely to analog technology. A place where you can get the latest news, in-depth technical coverage and product data. Plus articles contributed from some of the finest technologists in the world. And in coming months, we'll add seminars, chats, polls, conference registration and a lot more. All under the leadership of Steve Ohr, regarded by many as the leading analog advocate in the world today.

So if you're an analog engineer, come to www.planetanalog.com for a place to call your own. You'll find we speak your language.

SPONSORED BY:

Texas Instruments

Analog Devices

National Semiconductor

EDTN NETWORK PARTNERS

EBN, EET, SBN, Embedded.com, ISD Magazine, Power Designers, Home Toys, Electronic Design, Microwaves & RF, EE Product News, Wireless Portable, Wireless System Design, PCN Alert, ChipCenter, Communications System Design, Design & Reuse, Circuits Assembly, HDI, PC Fab, Printed Circuit Design

www.edtn.com

EDTN network

It's all right here.

can be automated, or partially automated. Others are tedious. Money—low to medium, depending on what is checked. Most checks can be done by hand. Others, such as verifying interrupt timing, require test equipment.
Effectiveness: High. This is the only way to develop confidence in some areas of the code. It's difficult to generate tests that will produce certain conditions (for example, worst-case stack depth, maximum interrupt latency). However, by analyzing the code, the engineer can determine what the worst case condition is and show that the system has been designed to handle it.

Stress/performance testing
Can the system handle its intended load? How close to the edge of acceptable performance does the system come? Stress tests are designed to load the system to the maximum specified load and then beyond, to see where it breaks. "Load" could be number of users, number of messages per unit time, frequency of a periodic interrupt, number of dynamically allocated tasks, and so on. Knowing at what point the system fails tells us how much overhead (factor of safety) we have. Performance testing is conducted to measure the system's performance. This may be important to verify that a requirement is met, to ensure the design is adequate, or to determine resources available to add more features. (Examples: processor throughput, interrupt latency, worst case time to complete a given task, and so on.)
Cost: Time—low. Money—low to high, depending on what's measured.
Effectiveness: High. There's nothing like knowing how close to the edge you're really running.

Writing test procedure
Many times the greatest benefit in developing formal test procedures is not executing the tests, but simply developing good test procedures. The code and/or the requirements must be examined carefully to determine what to test and how to test it. In the light of this increased scrutiny, many problems in the code are uncovered.
Cost: Time—high. It does take time to develop good test procedures. Money—none.
Effectiveness: High.

User report
From one perspective, your product is continually being tested by those who use it. Occasionally, a user may find a bug that evaded all previous test efforts. Most customers do not enjoy being used as unpaid test engineers, and get a little testy when they find bugs. Others volunteer to test your code (beta-testers) in order to get the benefits of

your product sooner. Regardless, user reports always have to be verified: sometimes alleged errors are due to improper user understanding, hardware conflicts, or other problems unrelated to the software's operation.

Cost: Time—none (of yours). Money—none. Of course, the intangible costs to your business could be much larger.

Effectiveness: Medium.

Tools

Various tools are helpful to test code and find bugs. Software debuggers allow the programmer to start and stop execution, modify the data in memory and registers, and sometimes trace execution and coverage. This can be useful in stepping through the code, and forcing execution down particular paths. These techniques are often used in unit testing. Similarly, a logic analyzer can be used to trace execution. It can show what happened in time, but doesn't provide a way to control the execution. An ICE (often used with a debugger itself) can do everything the software debugger does, and trace real-time execution. Its pod replaces the processor, giving the engineer complete control of the system. A simulator allows the code to execute in a virtual environment on the host. Most simulators provide a means of writing scripts, which can simulate inputs, record outputs, and otherwise control the execution of your code. They can be useful for unit testing, especially where repeatability is desired.

In addition to these basic tools, many other programs are useful in testing code. Tools exist to capture events and play them back, which is useful in automating a procedure. Other tools simply help organize the testing of complex systems. Some help in developing a traceability matrix. Other tools provide scripting capabilities, which can speed the development of procedures. Yet other tools provide a skeleton of code, which can be modified to perform unit testing on a host, another step toward automating the process.

Literally hundreds of tools and companies are out there that can help identify the bugs we assume to be in our code. However, beware of claims that sound like a "silver bullet." In my experience, there's always some set of assumptions and conditions that limit those claims to a particular class of applications or type of testing.

Knowledge applied

Now that we've covered the sorts of bugs that lurk in our code, and some of the techniques useful in finding them, you may say, "I don't have time to exhaustively test every line of code six different ways. What approach should I use to test my code? How much time and money will it cost?"

The answers to those questions are "it depends." What's the purpose of your testing? Is it to complete a mandatory activity as part of your company's software development process? Is it to ensure that the code implements all the advertised features? On the other hand, is the purpose to ensure beyond a reasonable doubt that the system in which your software runs won't kill someone because of a software error? Or destroy a multi-million dollar piece of equipment? In other words, what is the risk posed by the inevitable software bugs? Testing theory states, "all software has errors." Some errors probably will never be noticed, ever. Other errors may be catastrophic. As engineers we have to balance the risk that alleged software errors pose with the cost of finding them.

For different purposes, and different types of systems, some test strategies are more appropriate than others. Figure 1 shows which tests are generally best suited to finding certain types of errors. Knowledge about the sorts of errors that may be in a program—and the risks that they pose—can then be used to select the testing strategies that should be the most effective.

Parting thoughts

None of the techniques here can be used effectively outside of a good software development process. Nor will they be of help if not performed thoroughly, with appropriate management buy-in. All the ideas presented may need to be modified to suit your specific environment. In addition, here are a few parting thoughts:

- Test early and often
- Understand the real goals of the testing effort, and make your plans accordingly
- Plan your testing efforts and stick to your plan. The time to start thinking about testing is in the concept phase, when each requirement's testability should be evaluated. These plans (and later, test procedures) continue to evolve, and should be complete when the code is ready to test
- Obtain required resources (both time and equipment) early in the development cycle
- Share your successes and failures. Learn from others what works and what doesn't
- Testers and developers are part of the same team. Work cooperatively
- Test each other's code. It's difficult to see your own mistakes **esp**

Sean Beatty is an engineering manager with Software Engineering Professionals. He has been involved with designing and programming embedded systems since 1986, and received his BSEE from the University of Wisconsin-Milwaukee in 1989. Sean has worked with many companies in the embedded systems arena, with particular emphasis in medical and automotive software. His e-mail address is smbeatty@sep.com.

The author would like to acknowledge the following people for their help with this project: Lu Cheng, Dave Mott, Mark Runyon, and the employees of Software Engineering Professionals.

Tools for Embedded Developers

Software

Protocol suite and RTOSes

Several software tools—the USNET TCP/IP protocol, the SuperTask! RTOS, and the TronTask! RTOS—are available in versions that support Motorola's MBX860 board. The board, which is designed for communications and networking OEMs, supports MPC860 processors operating at up to 40MHz, as well as 128MB of EDO DRAM or standard Fast Page DRAM in 168-pin DIMM sockets. Supertask! and TronTask! for MBX860 are available now. A royalty-free license (source code included) costs $8,000. USNET for MBX860 is also available. It costs $7,500.

US Software
Hillsboro, OR
(800) 356-7097
www.ussw.com

Development tools, the processors they support

CodeWarrior Integrated Development Environment 2.0 for 68K/ColdFire is the latest release of a set of development tools whose enhancements include support for Motorola's ColdFire processors, as well as an updated compiler and debugger. The processors and development tools, are targeted for consumer electronics, office automation, robotics, and motor control systems. RTOSes supported by this development environment include the Nucleus PLUS Kernel, EUROS from Dr. Kaneff Engineering Consultants, and Motorola's PPSM (v. 3.02). Also being released is CodeWarrior for the Motorola PowerPC v. 5.0. You can purchase CodeWarrior v. 2.0 for 68K/ColdFire now for $2,500. CodeWarrior for Power PC v. 5.0 is available for the same price.

Metrowerks
Austin, TX
(800) 377-5416
www.metrowerks.com

Tools for multi-processor systems

A version of the 8051 Virtual Workshop is available that supports debugging and verification for multiple processor systems, without hardware. Its automatic address recognition allows serial communication between microcontrollers with "a minimum" of processor intervention. By running multiple instances of Virtual Workshop, either on a single PC or several PCs on a network, you can simulate multiple target systems, making the presence of actual target hardware less critical, although the Virtual Workshop is capable of communicating with a real system.

Crossware Products
Herts, UK
44 (0) 176-385-3500
www.crossware.com

Expanded support for design software

Hypersignal RIDE is DSP design software made to support Texas Instruments' DSK6211 and DSK6711 DSP hardware. It can be used in a number of DSP applications, such as instrumentation, systems modeling and simulation, image processing, and other numeric-intensive applications. Block components are selected from a pull-down menu or toolbar and connected together with a mouse to establish a data flow. Run-time parameters are adjusted from pop-up dialog boxes, and the algorithm is executed on the target DSK by pressing a button. The software is also capable of symbolic debugging, code profiling, and application export. RIDE software is available now in several editions costing between $3,995 and $9,995.

Hyperception
Dallas, TX
(214) 343-8525
www.hyperception.com

Development environment, compiler family, and processor

The MULTI 2000 integrated development environment and family of C, C++, and EC++ compilers are now available for the TriCore Unified processor from Infineon Technologies. MULTI 2000 integrates source-level debugging, program building, profiling, and version control for the development of software for the TriCore architecture, which features a 16-/32-bit instruction set. Multi 2000, including a C/C++ compiler costs $5,900. A Unix version costs $8,900.

Green Hills Software
Santa Barbara, CA
(805) 965-6044
www.ghs.com

Open-source OS released for digital avionics industry

MicroC/OS-II is an industrial open-source RTOS for digital avionics that offers semaphores, message mailboxes, message queues, event flags, interrupts, and time delays, among other features. A certification package includes source code, build files and requirements, design, and test and integration documentation to meet the requirements of avionics manufacturers seeking to obtain FAA RTCA/EUROCAE DO-178B certification. The software and certification documentation are available now, starting at under $30,000.

OnCore Systems
Half Moon Bay, CA
(650) 712-0655
www.oncoresystems.com

Multi-user UML development environment

Features of the latest release of the TAU UML Suite v. 4.3 include support for UML v. 1.3 graphical notation, XML/XMI, Java2, and Enterprise Java beans. It's part of the Tau development environment, which comprises tools for the analysis, design, and testing of event-driven, real-time applications in a package that merges the graphical notation of UML, the real-time design and development language SDL, and the ISO standard test scripting language TTCN. The Tau development environment is available now.

Telelogic AB
Malmö, Sweden
(609) 520-1935
www.telelogic.com

Hardware

Evaluation kit

The evaluation kit for the SX-Stack TCP/IP protocol stack contains a demonstration board, an AC power supply, a nine-pin-to-nine-pin serial cable, a CD-ROM containing source code and other information, and a user's guide. The SX-Stack is a configurable combination of standard Internet protocols implemented as Virtual Peripheral modules. The evaluation kit costs $99. It's available now.

Scenix
Mountain View, CA
(650) 210-1500
www.scenix.com

Emulation pod

The PXC76x emulates Philips' 8051 LPC76x controllers, whose family consists of three members with up to 4K OTP ROM: the 87LPC762, 87LPC764, and 87LPC76. An emulation system is offered in conjunction with the PXC76 pod, which includes dual-ported emulation memory, hardware breakpoints, an operating voltage range from 2.7 V to 5.5 V, and support for on-chip peripherals. The pod itself costs $1,245 but would only be useful by itself to designers who already had the full emulation system, which costs from $4,000 to $6,000. Both pod and system are available now.

Hitex Development Tools
Sunnyvale, CA
(800) 454-4839
www.hitex.com

Microprocessor interface

The Rabbit 2000 Core Module, which provides an interface for the Rabbit 2000 microprocessor, includes the following components in a design which measures 1.9-in. by 2.3-in.: 40 general purpose I/Os, SRAM and flash, memory I/O interface, master-slave control pins, five 8-bit timers, two 10-bit timers, and four CMOS compatible serial ports. It's available now as part of a development kit for $169, and by itself starting at $39.

Rabbit Semiconductor
Davis, CA
(530) 757-8400
www.rabbitsemiconductor.com

GUI tool

Graphics Toolkit, a hardware/software package for developing GUIs, provides tools for repetitive tasks. It supports bitmap and vector graphics with Cartesian and polar coordinate systems. Graphics can be used for both static LCD graphics displays and real-time graphics animations for oscilloscopes, animated bar and pie graphs, low-resolution video sequences, animated street maps, on-screen instruments, and representations of active processes. It also features tools for the integration of input peripherals such as keyboards, pointer instruments, rotary position transducers, and touch screens. Device drivers are included to control these peripherals. It's available now for $595.

Wilke Technology
Aachen, Germany
4 (924) 191-8900
www.wilke.de

Emulator

The PowerTAP emulator for Motorola's MPC7400 was designed for developers in telecommunications, Internet client/server, and high bandwidth graphics and video industries. It features a cache display window, MMU support for memory translation and paging, flash programming, and master/slave support for multiprocessor system debugging. The debugging feature provides bit-level detail of the registers. PowerTAP is available now, starting at $4,990.

Applied Microsystems
Redmond, WA
(800) 426-3925
www.amc.com

Chips

Microprocessor core

The EZ4021 MiniRISC is a 64-bit MIPS microprocessor core that operates at 250MHz and is targeted for communications applications such as set-top boxes, image processing engines, and networking switches and routers. It was introduced with the TinyRISC, a 32-bit microprocessor. The EZ4021 measures 12mm^2 and includes 16K instruction and 16K data caches. Power consumption is 2.6mW/MHz. The TinyRISC measures 2mm^2 and it operates at a speed of 120MHz. Evaluation/development kits will be available in the third quarter.

LSI Logic
Milpitas, CA
(408) 433-8000
www.lsilogic.com

DSP system-on-a-chip

Xtensa III is the the third generation of chips featuring the Xtensa processor technology. The chips come with several options for configuration, including a DSP coprocessor called the Vectra, whose worst-case performance is listed at 200MHZ. The Vectra coprocessor can be configured to 8-, 16-, and 24-bit fixed-point applications. Another option is a 32-bit floating-point coprocessor designed with printing, graphics, and audio applications in mind. Major features of this coprocessor include 16 dedicated floating point registers, as well as a set of load/stores, offset, and indexed address update modes, among others. Licensing fees for an individually configured processor and software tool environment start at $350,000. The Vectra DSP option bumps the price up to $500,000. It's available now.

Tensilica Inc.
Santa Clara, CA
(408) 986-8000
www.tensilica.com

Programmable DSP

The TMS320C5510 DSP device is the first in a family of programmable DSPs featuring the TMS320C55x DSP core. The C5510 DSP will be available at a clock rate of 160MHz, or 320 MIPS, with power consumption of 80mW. It uses a dual-MAC architecture featuring a 32-bit program bus, three 16-bit data read buses, two 16-bit data write buses, and five 24-bit address buses. It also offers 160K words of on-chip SRAM. Samples of the C5510 DSP in 15 x 15 mm microStar BGA packages are available now, with production quantities, as well as a 200MHz version, scheduled for the fourth quarter.

Texas Instruments
Dallas TX
(800) 477-8924
www.ti.com

USB chip

The 32-pin FT8U232AM is a single chip designed for transferring serial data over the Universal Serial Bus. It can be used to upgrade old RS-232 designs. Data transfer rates are listed at up to 2MBps. The chip's architecture accomodates use in a variety of applications such as USB modems, USB interface cables, RS-422 data links, and instrumentation. It is sampling now and will be available in the third quarter in production quantities. It costs $3 each in quantities of 1,000.

Future Technology Devices
Victor, NY
(716) 425-3753
www.ftdi.com

Microcontroller lines expanded

The ST9 and ST7 lines of 8-bit microcontrollers have been expanded to include two families, designated ST92163 and ST7263, respectively, that carry an integrated Universal Serial Bus interface. In addition to complying with the USB 1.1 specification, the interface includes a 5V/3.3V software-controlled voltage regulator, USB transceivers, and a serial interface engine with a built-in direct mem-

ory access (DMA) controller. The ST92163 units are based on the ST9 8-/16-bit core running at 24MHz. Its USB interface has up to eight functions and 16-bidirectional endpoints, programmable buffer size, and transfer direction. The ST7263 family of MCUs is based on the ST7 core, and offers a USB interface with three bidirectional endpoints. The ST7263 family is available now starting at $3 each in quantities of 10,000. It comes in ROM and OTP versions, in SO34 and SDIP32 packages, as well as in die form. Units in the ST92163 family start at $6 each in quantities of 10,000. They come in ROM and OTP versions, in TQFP64 and SDIP56 packages, and also in die form.

STMicroelectronics
Lexington, MA
(781) 861-2650
www.st.com

Signal processor

The ADCDS-1403 is a single package, 14-bit signal processor that combines correlated double sampling with a sampling A/D converter. It also features a user-configurable input amplifier, programmable analog input bandwidth, gain and offset adjustments, out-of-range indicator, internal timing circuits, and a precision reference circuit. The ADCDS-1403 requires two externally generated timing signals to digitize the CCD's analog output signal: a Reference Hold—to capture the CCD reference signal—and a Start Convert command. It's housed in a 40-pin TDIP. It's available now, for $192 each in quantities of 100.

Datel, Inc.
Mansfield, MA
(508) 339-3000
www.datel.com

Flash microcontroller family

The MMC2107 is the first member of the M-CORE family of flash microcontrollers, whose 32-bit chips are designed for use in automation applications such as vending machines, building management and heating-ventilation-air conditioning systems, exercise equipment, and lighting control. Besides the M-CORE 32-bit CPU, the MMC2107 features a 16-bit timer, communications interfaces, a queued analog-to-digital converter, and 8K of static RAM. Three evaluation boards are supported by the MMC2107 microcontroller: the MMCEVB2107, the MMCCMB2107, and the KITEVS2107. The MMC2107 is currently available in two packages: a 144-pin low-profile quad flat pack for use in single-chip and expanded modes, and a 100-pin version. It's available now for $15 per unit in quantities less than 25,000.

Motorola Inc.
Austin, TX
(512) 502-2100
www.motorola.com

Motion sensor interface IC

The LS6511 is a low-current, mixed-signal CMOS Silicon Gate IC designed to interface with a PIR sensor to produce an intrusion detection module for security systems. Gain and bandwidth are controlled by an external R-C network to enable application-specific customization. Output from a two-stage amplifier is processed by a window comparator and digital filter. The LS6511's features include mode select for single pulse detection, an output duration timer, a 5V shunt regulator, and under-voltage detection. Two outputs are provided for direct drive of a relay coil and LED. It's available now in either a 14-pin DIP for $0.85 in quanities of 10,000, or a 14-pin SOIC.

LSI Computer Systems Inc.
Melville, NY
(631) 271-0400
www.lsicsi.com

3DO WANTS YOU.
We're looking for a few good recruits.
Engineers •
Producers •
Directors •
Designers •
Artists •

Engineering Opportunities

CURTS PMC is a global electronics company and a leader in the design and manufacture of control devices for the electric vehicle market. Located approximately 35 miles from San Francisco, we are currently seeking the following experienced Engineers to join our team.

Project Leader

Lead an interdisciplinary team responsible for the design of new or modified motor control products from initial concept through production prototypes. Requirements include a BSEE (or equivalent) and 7+ years of demonstrated applicable experience. Must have experience with power electronics or embedded systems. Design engineering and hands-on technical skills necessary. 3+ years of experience in project engineering desired.

Product Specialist

Support the Global Product Manager for Motor Control in the performance of all duties related to a specific product family. Requirements include a BSEE (or equivalent); MBA or MBA candidate desirable coupled with experience in product definition and working with customer requirements.

Software Engineer

Conceptualize, design, develop, analyze and test embedded micro applications for motor control devices. Requirements include a BSEE/CS (or equivalent) and 2+ years of experience in micro software development. Must have experience in Assembly and C languages, as well as hardware/software integration, emulators, and debugging. Design and hands-on technical skills required. Experience in circuit analysis a plus.

We offer company-paid health benefits for employees & dependents, profit sharing, matching 401(k) plans and PTO. For consideration, qualified candidates should send, fax or e-mail their resume, indicating area of expertise, to: Human Resources Department, CURTIS PMC, 6591 Sierra Lane, Dublin, CA 94568. Fax: (925) 833-8777. E-mail: curtispmc@hotmail.com

For complete job descriptions and additional career opportunities, explore our Web site at: www.curtisinst.com
Equal Opportunity Employer M/F/D/V

CURTIS
Innovation & Excellence

SOFTWARE ENGINEERS

Packard Instrument Company, an international leader in life science instrumentation, is seeking a Software Engineers for our Corporate location in Downers Grove, Illinois.

Duties will include: New product development, design and development of robotic software used to control precision liquid handling equipment used in laboratories and software to control analytical instrumentation in Biomedical research, participating in software designs, estimating schedules, conducting design reviews, coding, documenting and testing PC based and embedded controller software and perform other related functions as assigned.

Candidates must possess:

- **Good communication skills.**
- **Ability to work well in a team environment.**
- **Ability to program in C++, Visual Basic and DSP code.**
- **Ability to analytically design software for complex real time machine control systems.**
- **Knowledge of biomedical laboratory methods and equipment is a plus.**

Previous experience:

- **2-5 years experience in the development of software for new electro-mechanical products.**
- **Must have experience in real time machine control software or robotic control systems.**
- **Desired experience with DSP code, Windows NT programming, C++ and COM.**

Send or fax or e-mail your resume to:
Human Resources Dept., Packard Instrument Company, 2200 Warrenville Road, Downers Grove, IL 60515. Fax: 630-969-6511.
E-mail: jjohannpeter@packardinst.com. Visit our Website: http://www.packardinst.com *Equal Opportunity Employer M/F/D/V*

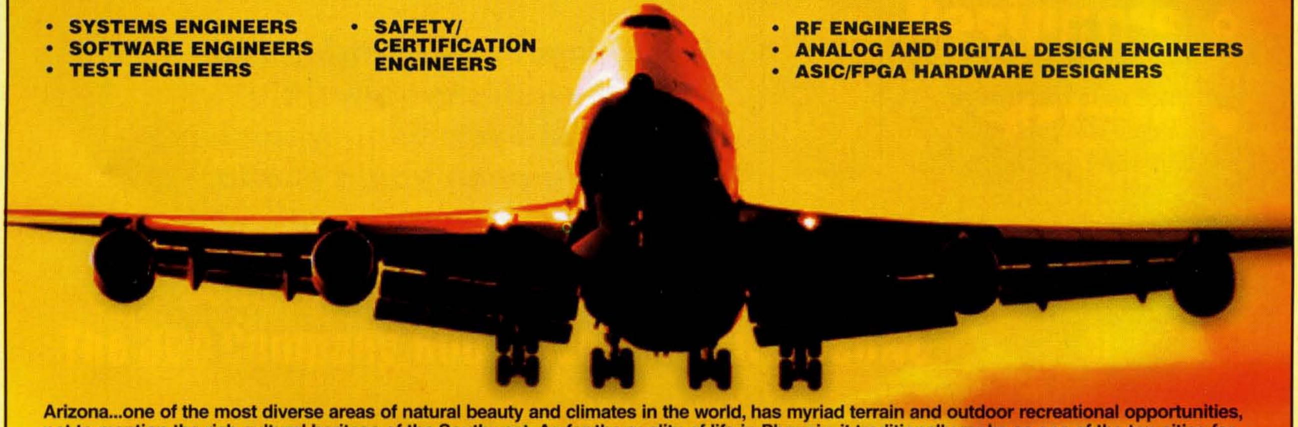
Packard
A Packard BioScience Company

Honeywell

Expansive wings for the imagination.

At Honeywell in Phoenix, AZ the future of Avionics is now. Imagine shaping the cockpit of the 21st century while soaring with a $24 billion global technology player. We are seeking talented individuals to support our Aviation Information Services, Displays, Flight Controls, Fly-By-Wire, Advanced Common Flight Deck, and Platforms products for our Air Transport, Business/Regional Jet, and Helicopter Programs.

- **SYSTEMS ENGINEERS**
- **SOFTWARE ENGINEERS**
- **TEST ENGINEERS**
- **SAFETY/ CERTIFICATION ENGINEERS**
- **RF ENGINEERS**
- **ANALOG AND DIGITAL DESIGN ENGINEERS**
- **ASIC/FPGA HARDWARE DESIGNERS**

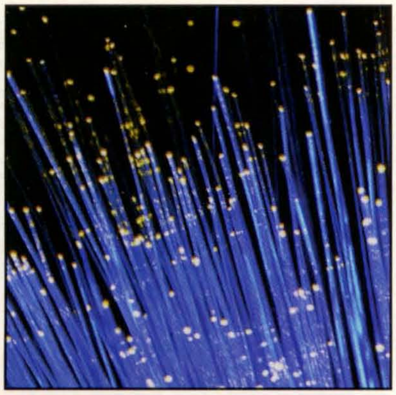


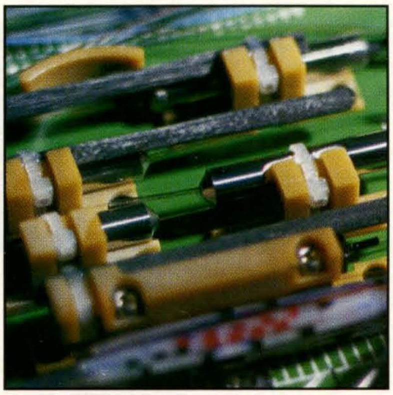
Arizona...one of the most diverse areas of natural beauty and climates in the world, has myriad terrain and outdoor recreational opportunities, not to mention the rich cultural heritage of the Southwest. As for the quality of life in Phoenix, it traditionally ranks as one of the top cities for population growth, economic development, jobs created, as well as a cost of living below the national average. You'll agree that Arizona is a great place to live; especially Phoenix, the 6th largest city in the USA.

As part of our team, you will enjoy a competitive salary and an outstanding benefits package including one of the top 401(k) plans in the nation. Please send your resume, including REF# XHXES08, to Email: janet.ludwig@honeywell.com or Fax: 602.436.5050.

As an equal opportunity employer, we are committed to a diverse workforce.

Stretch Your Imagination.
www.honeywell.com





ANALYSTS INTERNATIONAL™

Contact Analysts International if you have skills in any of the following areas.

Software engineer with one year or more experience doing development in C or C++ for embedded systems.

Real-time programming in C, C++ and assembler.

RTOS: experience configuring an RTOS.

I/O interfacing: Interrupts and traps, buffer management.

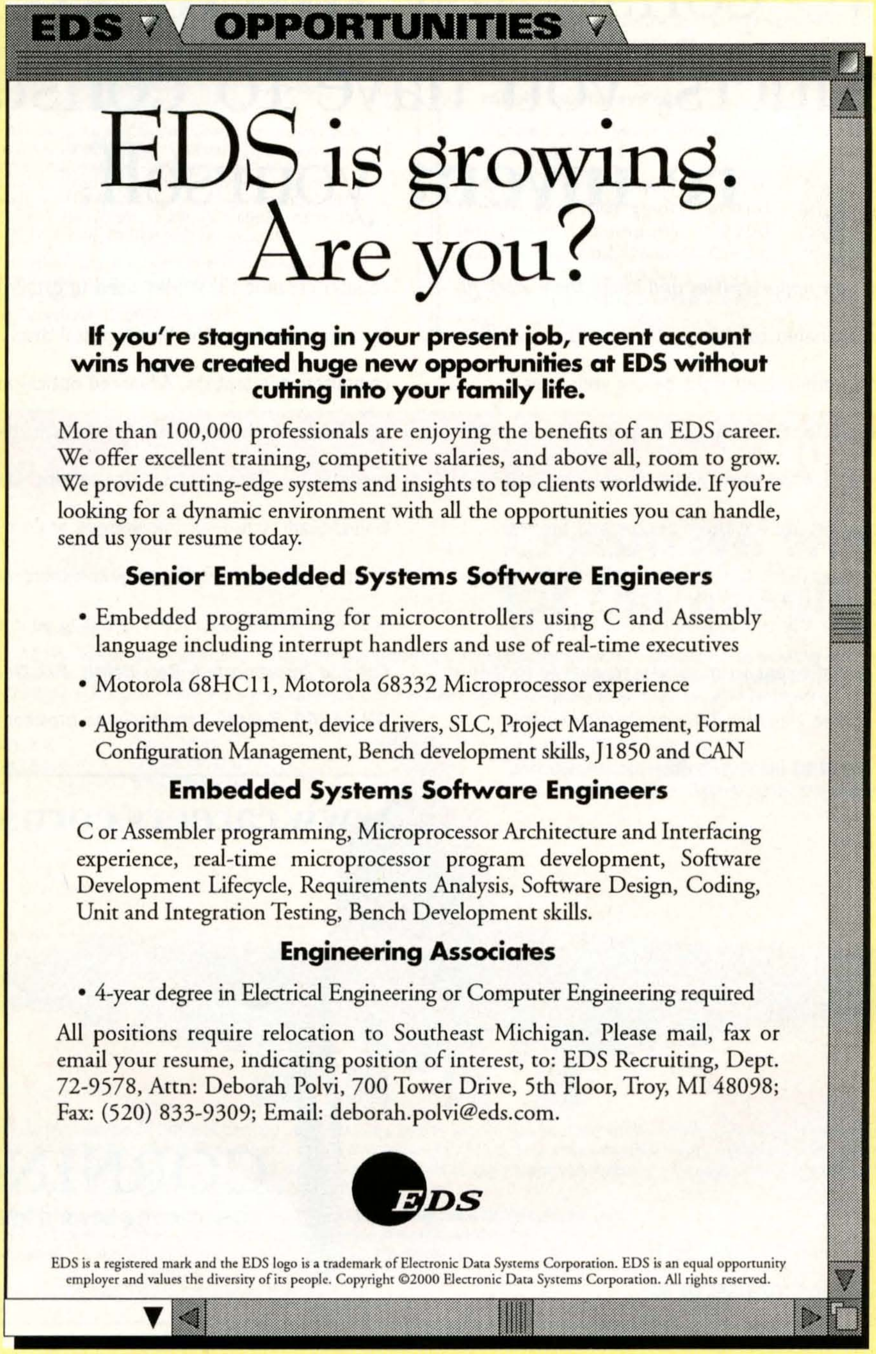
Communications: Serial, HDLC, SONET/SDH, TDMA, ISDN, ISUP, TCP/IP

Industry areas: radio communications, telephony, networking, biomedical.

Location is Chicago and the surrounding suburbs.

1101 Perimeter Drive, Suite 500
Schaumburg, Illinois 60173
Telephone 847.619.4673 Facsimile 847.605.9489
E-mail: chicago.jobs@analysts.com
www.analysts.com

©2000 Analysts International Corporation EOE. M/F/D/V



EDS is growing. Are you?

If you're stagnating in your present job, recent account wins have created huge new opportunities at EDS without cutting into your family life.

More than 100,000 professionals are enjoying the benefits of an EDS career. We offer excellent training, competitive salaries, and above all, room to grow. We provide cutting-edge systems and insights to top clients worldwide. If you're looking for a dynamic environment with all the opportunities you can handle, send us your resume today.

Senior Embedded Systems Software Engineers

- Embedded programming for microcontrollers using C and Assembly language including interrupt handlers and use of real-time executives
- Motorola 68HC11, Motorola 68332 Microprocessor experience
- Algorithm development, device drivers, SLC, Project Management, Formal Configuration Management, Bench development skills, J1850 and CAN

Embedded Systems Software Engineers

C or Assembler programming, Microprocessor Architecture and Interfacing experience, real-time microprocessor program development, Software Development Lifecycle, Requirements Analysis, Software Design, Coding, Unit and Integration Testing, Bench Development skills.

Engineering Associates

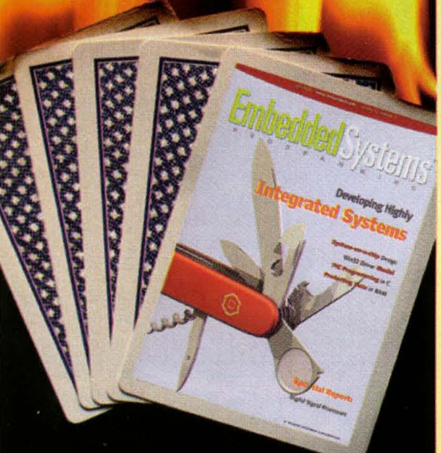
- 4-year degree in Electrical Engineering or Computer Engineering required

All positions require relocation to Southeast Michigan. Please mail, fax or email your resume, indicating position of interest, to: EDS Recruiting, Dept. 72-9578, Attn: Deborah Polvi, 700 Tower Drive, 5th Floor, Troy, MI 48098; Fax: (520) 833-9309; Email: deborah.polvi@eds.com.

EDS

EDS is a registered mark and the EDS logo is a trademark of Electronic Data Systems Corporation. EDS is an equal opportunity employer and values the diversity of its people. Copyright ©2000 Electronic Data Systems Corporation. All rights reserved.

Rent the Hottest List around!



EmbeddedSystems
PROGRAMMING

Over 55,000 Qualified Subscribers

Call Rubin Response
847-619-9800

A Consumer Electronic Leader

Taking Wireless Technology To The Next Level.

INTRODUCING WIRELESS CAREERS FROM PANASONIC.

We're Panasonic Wireless, the North American Design Center (Suwanee, GA & Calgary, Alberta, Canada) of an electronics giant. As an industry leader with a rapidly growing wireless unit, we can offer outstanding opportunities to ambitious, forward-thinking individuals. We're leading the way in digital technology with real-time embedded software and wireless/cellular projects. We have all of the excitement and advantages of a startup venture, yet with all of the rock solid support of a global electronics icon. Our headquarters is also located in Suwanee, GA. Just minutes away from the exciting, cosmopolitan city of Atlanta, we're still just far enough from the hustle and bustle to experience an outstanding quality of life.

SOFTWARE POSITIONS

Engineer I **SR Engineer**
Engineer II **Staff Engineer**

(Positions in Firmware, Protocols and Applications)

SYSTEMS & VERIFICATION

Test Engineer
Systems Engineer

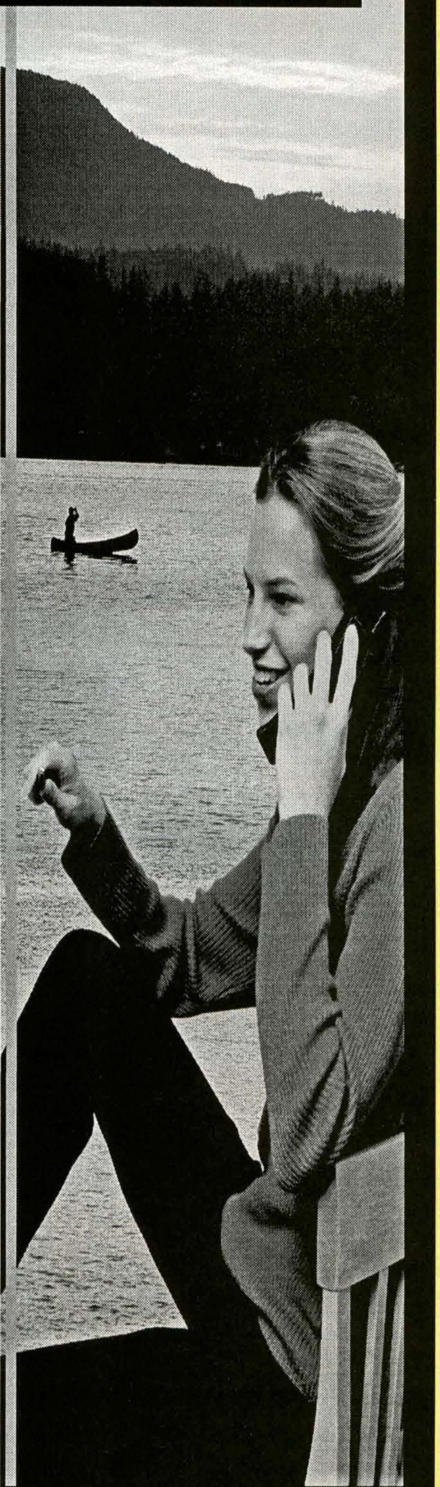
HARDWARE POSITIONS

Engineer I **Staff Engineer**
Engineer II **Technical Group Leader**

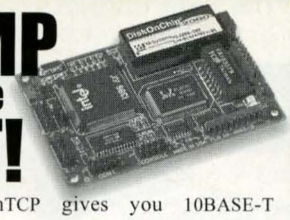
(Positions in RF and Logic)

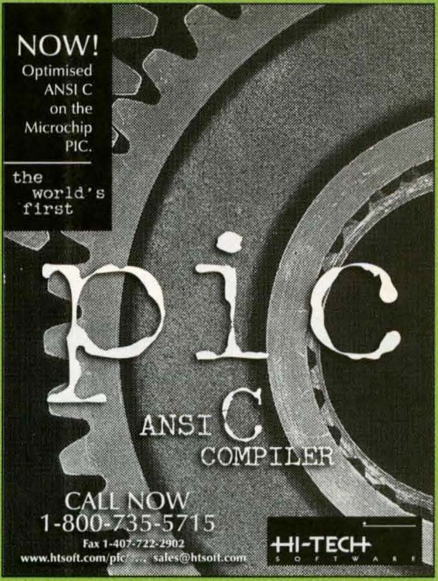
You'll enjoy a highly competitive salary, excellent career growth, and a comprehensive benefits package that includes medical/dental/vision, 401(k), tuition reimbursement, and product discounts. **For consideration, please forward your resume to recruiting@panasonicatlanta.com or fax 770-338-6001. Attn. Jobcode ES8/00.** EOE

Panasonic
Wireless Design Center
Matsushita Mobile Communications
Development Corporation of U.S.A.

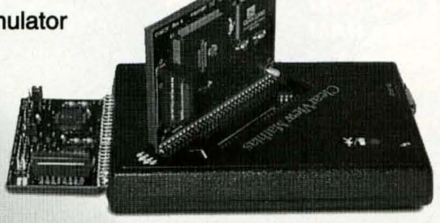


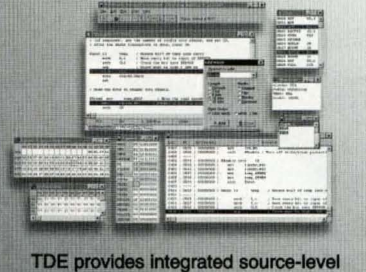
www.panasonic-wireless.com


DiskOnChip 2000
intel



PICmicro® Development System


ClearView Mathias



SB-56K

ADVERTISER INDEX

Advertiser	URL	Page
3DO	www.3do.com	126
Abraxas Software	www.abxsoft.com	143
Accelerated Technology	www.atinucleus.com	103
Advanced Transdata Corp.	www.adv-transdatacorp.com	133
Advantech	www.advantec.com/epc	112
Advin Systems	www.advin.com	132
American Arium	www.arium.com	C3
American Raisonance	www.americanraisonance.com	69
American Raisonance	www.americanraisonance.com	87
Analysts International	N/A	130
Applied Data Systems	www.flatpanels.com	90
ARC Cores Ltd.	www.arccores.com	12
ARTiSAN Software Tools	www.artisansw.com	27
Avocet Systems Inc.	www.avocetsystems.com	88
Avocet Systems Inc.	www.avocetsystems.com	C4
Axiom Technology	www.axiomtek.com	43
Blackhawk	www.blackhawk-dsp.com	21
Blunk Microsystems	www.blunkmicro.com	134
BSQUARE Corp.	www.bsquare.com	93
CAD-UL	WWW.CADUL.COM	100
CMX Systems Inc.	www.cmx.com	141
Cogent Computer Systems	www.cogcomp.com	143
CONITEC Datasystems Inc.	www.conitec.com	135
Corning Inc.	N/A	129
COSMIC Software	www.cosmic-software.com	115
Curtis PMC	N/A	128
Dinkumware Ltd.	www.dinkumware.com	114
Domain Technologies	www.domaintec.com	134
EBS	www.ertsf.com	74
EBS	www.ertsf.com	114
EDS	www.eds.com/careers	130
Electronic Engineering Tools	www.eetools.com	133
EMAC, Inc.	www.emacinc.com	124
Embedded Power Co.	www.embeddedpower.com	101
Embedded Support Tools	www.estc.com	1
emWare	www.emware.com	61
Enea OSE Systems Inc	www.enea.com	83
esmertec	www.esmertec.com	47
Espial Group	www.espial.com	63
Express Logic	www.ghs.com/armsolutions	107
Express Logic	www.expresslogic.com	89
General Software	www.gensw.com	108
Grammar Engine Inc.	www.gei.com	132
Green Hills Software Inc.	www.ghs.com/armsolutions	107
Green Hills Software Inc.	www.ghs.com	6
HI-TECH Software	www.htsoft.com/pic	134
HI-TECH Software	www.htsoft.com	109
Hitex Development Tools	www.hitex.com	133
Hitex Development Tools	www.hitex.com	33
Hiware	www.hiware.com	111
Honeywell	www.honeywell.com	128
IAR Systems	www.iar.com	113
I-Logix	www.ilogix.com	16
I-Logix	www.ilogix.com	41
Infineon Technologies	www.infineon.com	29
Intel	www.intel.com	19
InterNiche Technologies	www.iniche.com	79
Introl Corp.	www.introl.com	133
iSYSTEM USA	www.isystem.com	132
JK Microsystems	www.jkmicro.com	134
JK Microsystems	www.jkmicro.com	134
KADAK Products Ltd.	www.kadak.com	91
Keil Software	www.keil.com	118
Lantronix	www.lantronix.com	14
Lauterbach	www.lauterbach.com	51
Lineo	www.lineo.com	53
Lynuxworks	www.lynuxworks.com	48
Megatel Computer Tech.	www.megatel.com	102
Metalink Corp.	www.metaice.com	133
Micro Digital	www.smxinfo.com	81
Micro/sys	www.embeddedsys.com	39
Microtek International	www.microtekintl.com	4
Microware Systems Corp.	www.microware.com	95
National Semiconductor	www.national.com	44
National Engineering Search	www.nesnet.com	127
Needhams Electronics	www.needhams.com	132
Nohau	www.nohau.com	C2
Nohau	www.nohau.com	132
Noral Micrologics Limited	www.noral.com	135
Nucleus Electronics Corp.	www.nucleus1.com	135
Pacific Softworks	www.pacsoft.com	66
Packard Instruments	N/A	128
Panasonic	N/A	131
Paradigm	www.devtools.com	116
ParaSoft Corp.	www.parasoft.com	35
Phar Lap Software Inc.	www.pharlap.com	120
Phyton Inc.	N/A	13
PointBase Inc.	www.pointbase.com	57
Prosyst USA Inc.	www.prosyst.com	11
Rabbit Semiconductor	www.rabbitssemiconductor.com	52
RadiSys	www.radisys.com/SS7	31
Rational Software	www.rational.com	25
RLC Enterprises	www.rlc.com	135
Schweitzer Engineering	www.selinc.com	127
Scientific Placement	www.scientific.com	127
Signum Systems	www.signum.com	132
Sleepycat Software	www.sleepycat.com	56
Sophia Systems	www.sophia.com	135
TASKING	www.tasking.com	125
Tech Tools	www.tech-tools.com	134
Tern Inc.	www.tern.com	132
TimeSys Corp.	www.timesys.com	64A
Treck Inc.	www.treck.com	73
Trillium Digital Systems	www.trillium.com	59
U.S. Software	www.ussw.com	71
Verticalnet	www.embeddedtechnology.com	97
Vesta Technology	www.sbc2000.com	133
Webprn.com	www.webprn.com	37
Wind River Systems	www.wrs.com	8
Working Engines Inc.	www.workingengines.com	127
Waferscale	www.waferscale.com	106
Xilinx	www.xilinx.com	105
Yokogawa Digital Computer	www.ydcusa.com	132
ZF Linux Devices	www.zflinux.com	55
Z World	www.zworld.com	132

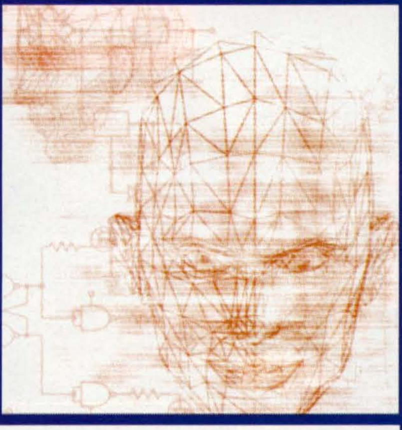
ICSPAT

ICSPAT

International Conference on Signal Processing Applications & Technology

October 16-19, 2000 • Adam's Mark Hotel • Dallas, TX, USA

Meeting of the Minds



4 easy ways to register

Web:
www.dspworld.com
Phone:
(800) 789-2223
Fax:
(415) 278-5390
Mail:
ICSPAT
C/o Corporate Registration
P.O. Box 612768
Dallas, TX 75261-2768

Attention signal processing engineers and programmers!

Don't miss this unique opportunity to enhance your signal processing knowledge and design skills. Take advantage of the many opportunities available this fall to enhance and update your signal processing knowledge at **ICSPAT**, the premier international forum for emerging signal processing research and development.

At **ICSPAT**, you can review cutting-edge research from around the world. Hundreds of white papers will be presented in more than 30 application areas including: Audio, Biomedical, Communications, Industrial Applications, Image Processing, Multimedia, Robotics, Telephony and more!

What you'll get at ICSPAT:

- Access to white papers in more than 30 application areas in the ICSPAT program
- A window on the future of signal processing and the applications in which it is used
- High quality technical knowledge and skill enhancement
- Expanded lectures offering more in-depth, enhanced coverage of topics
- Tutorial and plenary sessions as well as free product training representatives
- The official proceedings on CD-ROM

ESP

For a complete listing of ICSPAT lecture and poster abstracts, plus a comprehensive guide to plenary sessions, special events and industry activities, go to www.dspworld.com.

www.dspworld.com

Media Co-Sponsors:

CMP EETIMES

CMP Communication Systems design

CMP EmbeddedSystems Programming

Gold Sponsors:

MOTOROLA

STAR CORE

Silver Sponsors:

VxTel

CHAMELEON SYSTEMS, INC.

PENTEK

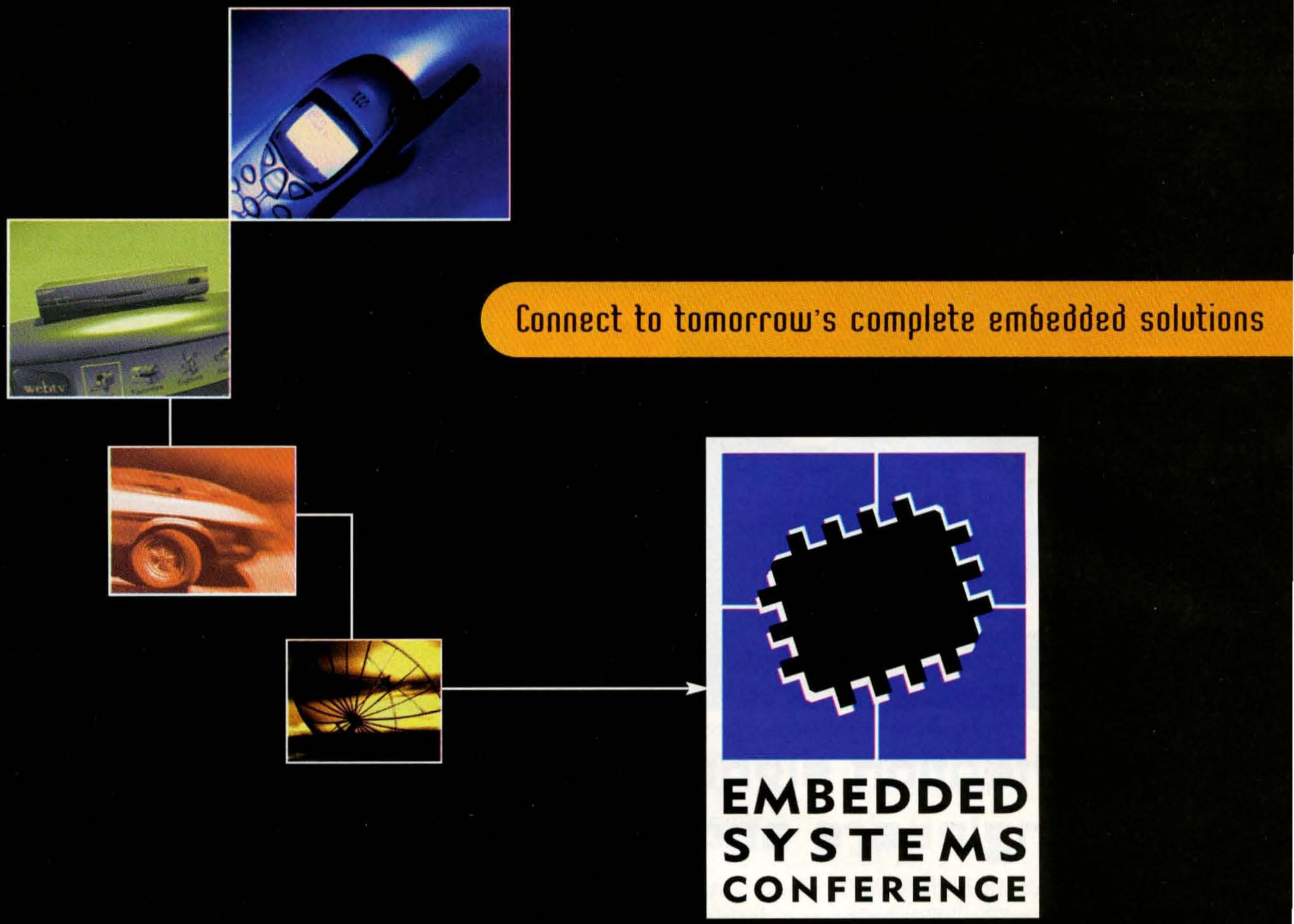
ANALOG DEVICES

BOPS

OSE

PRECISE Software Technologies Inc.

CMP


Connect to tomorrow's complete embedded solutions
webtv
EMBEDDED
SYSTEMS
CONFERENCE

Jack G. Ganssle

Radio Days

"**Hi**, I'm from the government and I'm here to help you!" We cringe when bureaucrats extend their helping hands—hands that usually take more than they give. Yet in the past few months Big Brother has delighted a lot of its citizens.

In May 2000 the Department of Defense turned off *selective availability*, the "feature" in the GPS system that deliberately introduced position errors into what is otherwise an amazingly accurate system. The military once worried that giving the world high-accuracy GPS increased the threat of accurate bombing of American cities. With our traditional enemies gone, and with the knowledge that the DOD can indeed turn off GPS to any particular area at any time, selective availability was clearly an anachronism.

I, for one, have always been infuriated that my tax dollars were used for both reducing the accuracy of GPS, and for cheating the system to recover the lost accuracy. For the Coast Guard has long offered differential GPS, a system that restores the lost accuracy by transmitting correction data from sites whose position has been determined with great care.

So now a little $100 navigation set can show your location to better than 15 meters—sometimes much better. This week my unit was showing an estimated position error of just one meter.

Perhaps this new precision will change many of the embedded systems we're building now. Car navigation units, for instance, often watch for abrupt course changes—like turning a corner—to determine that the vehicle is right at a particular intersection. That technique may no longer be needed.

In December 1999, the FCC also gave some of the country's citizens a gift when they amended the license structure for ham radio operators. The ruling affected many aspects of the hobby, but most profoundly, it eliminated the insanely high Morse code speed requirements. Previously, any ham operator wishing to transmit voice in most of the HF bands (under 30MHz) had to pass a code test at thirteen words per minute. The "Extra" license, ham radio's peak of achievement, required twenty wpm.

Of late, the government has been doing some good things for technologists. But maybe we're doing ourselves a disservice by overspecializing.

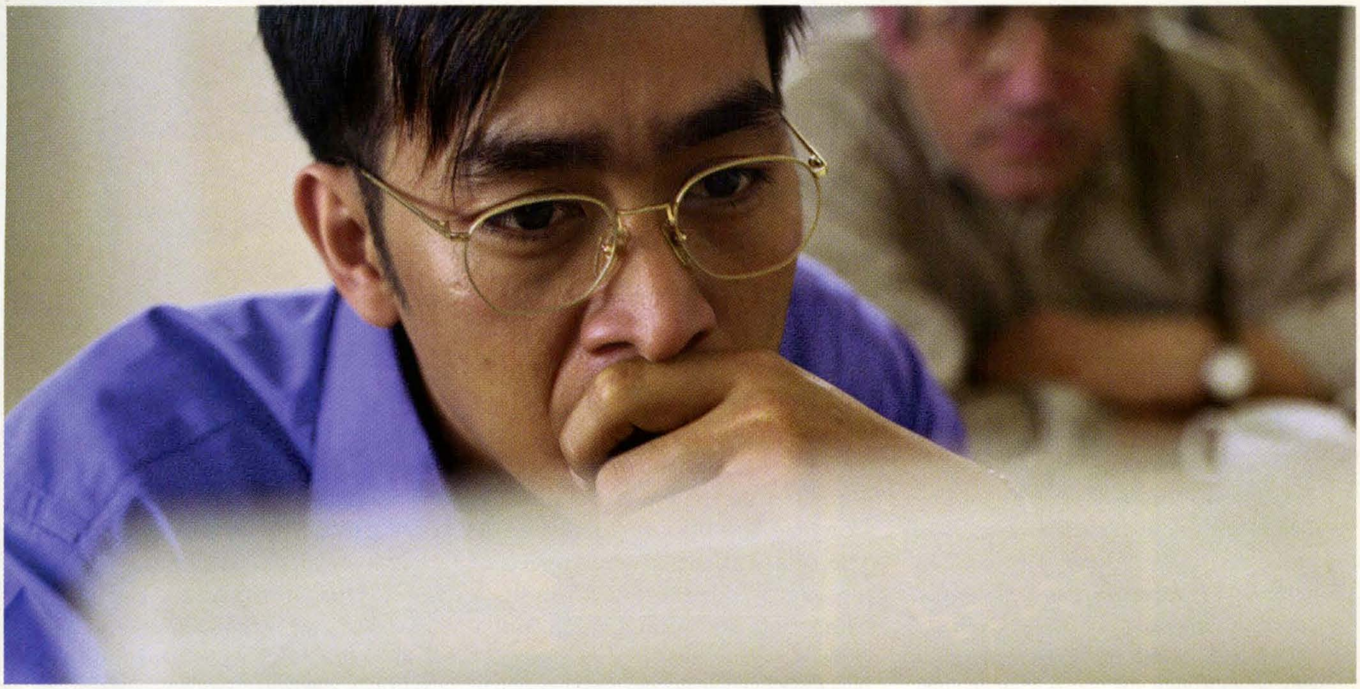
No doubt lots of you have struggled with the code. Most folks quickly reach a plateau around 10wpm. Getting to thirteen wpm requires an awful lot of effort. With the stroke of a pen the FCC maxed the speed at five wpm for all license grades. Five wpm is so slow—two seconds per character—that anyone can quickly pass the test just by memorizing the dits and dahs.

Ham radio's ranks have been thinning for decades, partly due to the difficulty of passing the code test, and partly due to young people's fascination with computers. In the olden days, ham radio was the Internet of the age; technically oriented people played with radios because computers were unobtainable. Now computer obsession and cheap worldwide communication supplants most folks' willingness to struggle with making contacts on noisy, unreliable HF frequencies.

Though I've had a license for decades, my fascination for working with radios died long ago. It's pretty cool to make contact with someone a continent away, but it's so much easier to pick up the phone or pop out an e-mail. Being busy, I've little time or desire to contact a more or less random person just to chat. Too much like a blind date. I'd rather spend time talking with friends and neighbors.

But when sailing I do find the ham bands useful since it's about the only way to talk to pals on other, distant boats, or to get messages to friends and family ashore. At sea, 1,000 miles from land, that radio suddenly becomes awfully compelling.

Today we're surrounded by radio transmissions, from the diminishing ranks of ham signals, to the dozens of FM-stereo stations in every market, to high-powered AM talk shows, TV stations, Bluetooth-enabled systems chatting with each other, GPS signals beamed from space, and, of course, the ubiquitous cell phone. Wireless is the future. We're wrapped in a dense fog of electromagnetic radiation.



Magazines abound with stories of these wireless marvels, yet I suspect that the majority of embedded developers have little to do with radio-based devices. That's a shame, since the technology underlying radio has much to offer even non-wireless developers.

Magazines abound with stories of these wireless marvels, yet I suspect that the majority of embedded developers have little to do with radio-based devices. That's a shame, since the technology underlying radio has much to offer even non-wireless developers.

The guts of radio

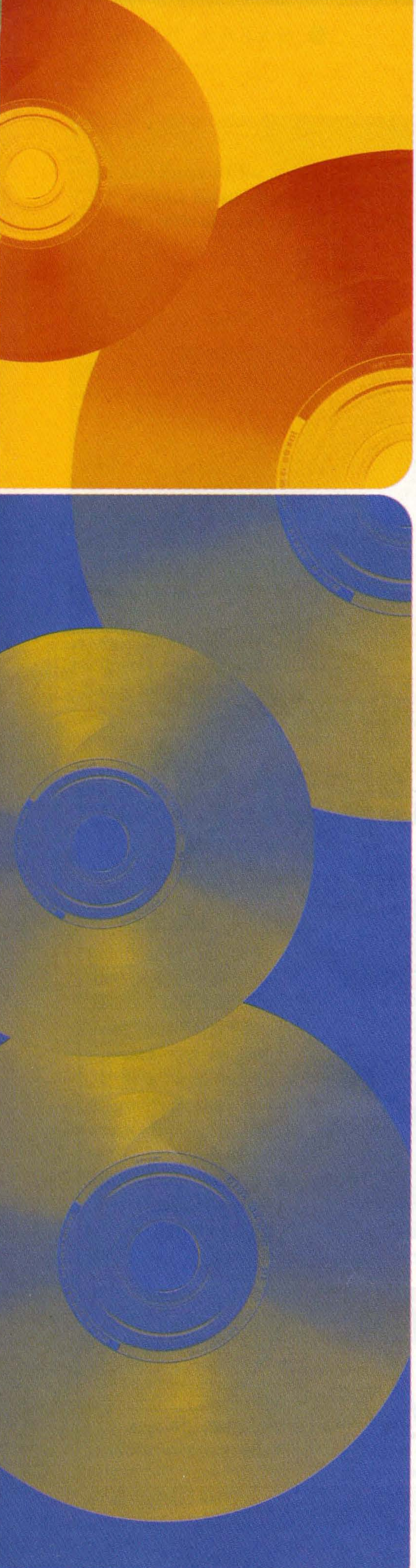
Think of that pea soup fog of electromagnetic waves that surrounds the planet. An antenna funnels all of it into your radio, an incredible mush of frequencies and modulation methods that amounts to an ineffable blather of noise. Where is that one rock 'n' roll station you're looking for in all of the mess? How can a $30 portable FM receiver extract ultra-high fidelity renditions of Weird Al's scatological riffs from the noise?

Today's radios invariably use a design called the Superheterodyne, or Superhet for short. Heterodyning is the process of mixing two AC signals to create a third at a lower frequency, which is easier to amplify and use.

The radio amplifies the antenna's output just a bit and then dumps it into a mixer, where it's added to a simple sine wave (produced by what's called a *local oscillator*) near the frequency of the station you'd like to hear. The mixer's output is the sum and the difference of the raw antenna signal and the local oscillator's sine wave. Turning the dial changes the frequency of the local oscillator.

Suppose you'd like to hear Weird Al on 102MHz. You set the dial to 102, but the local oscillator might actually produce a sine wave about 10MHz lower—92MHz—resulting in a 10MHz difference frequency and a 194MHz sum (which gets rejected). Thus, the mixer outputs a copy of the station's signal at 10MHz, no matter where you've tuned the dial.

A filter then rejects everything other than that 10MHz signal. The air traffic controller on 121MHz, the trucker's CB at 28MHz, and adjacent FM stations all disappear due to the filter's selectivity. All that other stuff—all of that noise—is gone. More amplifiers boost the signal, another mixer drops the frequency even more, and a detector removes



the RF part of the station, putting nothing more than unadulterated Weird Al out the speakers.

But how does the technology behind radio affect embedded systems? I've found it to be one of the most useful ways to eliminate noise coming from analog sensors, particularly from sensors operating near DC frequencies.

Consider a scale, the kind that weighs packages or people. A strain gauge typically interprets the load as a resistance. Feed a bit of current through the gauge and you can calculate weight pretty easily. The problem comes in trying to measure the sample's weight with many digits of resolution—at some point, system noise overwhelms the signal.

Noise has all sorts of sources. That sea of radio signals gets coupled into the strain gauge's wiring. So do distant lightening strikes. The analog-sensing electronics, itself, inherently adds noise to the signal. The challenge is to reduce these erroneous signals to extract as much meaningful data as possible.

In their anti-noise quest, analog designers first round up all of the usual suspects. They shield all sensor wires, twist them together to cancel common-mode signals, and wrap mu-metal (an electromagnetic barrier) around critical parts of the circuit.

When the analog folks can't quite get the desired signal-to-noise ratios, they ask the firmware folks to write code that averages and averages and averages to get quieter responses. Averaging yields diminishing returns (increase the number of sums by an order of magnitude and you only get 50% noise reduction) and eats into system response time. When the system finally gets too slow, we go to much more complex algorithms like convolutions, but consequently lose some of the noise minimization.

None of these approaches are bad. In some cases, though, we can take a lesson from RF engineers and quiet the system by just not looking at the noise. Analog noise is quite broadband; it's scattered all over the frequency domain. We can hang giant capacitors on the strain gauge to create a filter that eliminates all non-DC sources, but at the expense of greatly slowing system response time (change the weight on the scale and the capacitor will take seconds or longer to charge). This sort of DC filter is exactly analogous to averaging.

It's better to excite the gauge with RF, say at 100MHz, instead of the usual DC current source. Then build what is essentially a radio front-end to mix perhaps a 90MHz sine wave with the signals, amplify it as much as is needed, and then mix the result down to a low frequency suitable for processing by the system's A/D converter.

Off-the-shelf chips implement most of the guts of a radio. These offer high "Q" factors, which is a measure of the filter's narrowness. One that passes just 1,000Hz of the spectrum (meanwhile rejecting all other frequencies and thus most of the noise) has a higher Q than one that passes 10,000Hz.

Morse code hung in as a viable communications method for 150 years because it is incredibly bandwidth-efficient and noise-immune. You can filter out all of the RF spectrum except for just a tiny 100Hz slice, and still copy the information intact. Virtually all noise disappears. Voice communication, by comparison, requires at least 3kHz of bandwidth, thus a much lower-Q filter, and makes the system that much more susceptible to noise.

The scale is Morse-like, since the data changes slowly. The high-Q filter yields all of the information with almost none of the noise.

A radio design also greatly simplifies building very high-gain amplifiers—your FM set converts mere micro-volts out of the antenna into volts of speaker drive. It further removes the strain gauge's large DC offset from its comparatively small signal excursions.

Another example application is a color-measuring instrument. Many of these operate at near-DC frequencies since the input sample rests on the sensor for seconds or minutes. High resolution requires massive noise reduction. The radio design is simple, cheap (due to the many chip solutions now available), and quiet.

Many eons ago I worked as a technician on a colorimeter designed in the '60s. The design was quite fascinating as the (then) high cost of electronics resulted in a design that mixed both mechanical and electronic elements. The beam of light was interrupted by a rotating bow-tie shaped piece of plastic painted perfectly white. The effect was to change the DC sensor output to about a 1,000Hz AC signal. A narrow filter rejected all but this one frequency.

The known white color of the bow-tie also acted as a standard, so the instrument could constantly calibrate itself.

The same company later built devices that measured the protein content of wheat by sensing infrared light reflected from the sample. Signal levels were buried deep into the noise. Somehow we all forgot the lessons of the colorimeter—perhaps we really didn't understand them at the time—and slaved on every single instrument to reduce noise using all of the standard shielding techniques, coupled with healthy doses of blood, sweat, and tears. No other technical problem at this company ever approached the level of trouble created by millivolts of noise. Our analog amplifiers were expensive, quirky, and sensitive to just about everything other than the signal we were trying to measure.

Years of struggling with the noise in these beasts killed my love of analog. Now I consider non-digital circuits a nuisance we have to tolerate to deal with this very analog world.

The murky future

It scares me that we could have learned so little from the rotating bow-tie. I'm worried as well that increasing specialization reduces cross-pollination of ideas even from within the same industry. The principle behind radio eludes most embedded folks despite its clear benefits.

Knowledge is growing at staggering rates, with some pundits predicting that the total sum of information will double every decade before long. At this year's Embedded Executive Conference Regis McKenna played down this issue since machines will manage the data.

I'm not so sanguine. Skyrocketing knowledge means increasing specialization. So we see doctors specializing in hand surgery, process engineers whose whole career is focused on designing new Clorox bottles, and embedded developers who are experts at C, but all too often they have little understanding of what's going on under the hood of their creations.

A generalist, or at least an expert in a field who has a broad knowledge of related areas, can bring some well-known techniques—well known in one field—to solve problems in many other areas. Perhaps we need an embedded Renaissance person, who wields C, digital design, and analog op amps with aplomb. The few who exist are on the endangered species list as each area requires so much knowledge. Each can consume a lifetime.

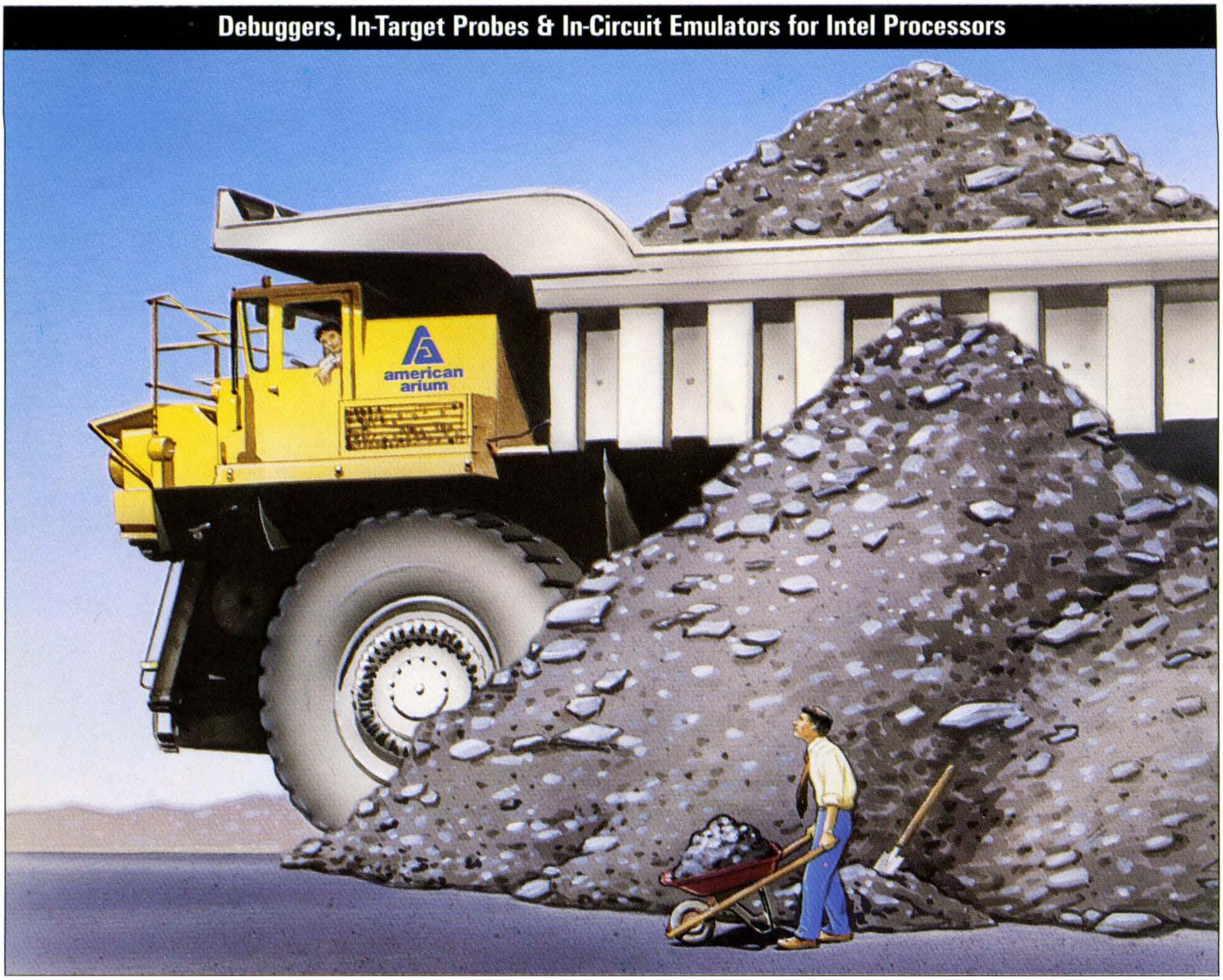
The answer to this dilemma is unclear. Perhaps when machines do become truly intelligent they'll be able to marshal great hoards of data across application domains. Our role then seems redundant, which will create newer and much more difficult challenges for homo sapiens.

I hope to live to see some of this future, but doubt that we'll learn to deal with the implications of our inventions. Historically, we innovate much faster than we adapt. **esp**

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. He founded two companies specializing in embedded systems. Contact him at jack@ganssle.com.

EMBEDDED SYSTEMS PROGRAMMING (ISSN 1040-3272) is published monthly, with an additional issue published in September, by CMP Media, 525 Market St., Ste. 500, San Francisco, CA 94105, (415) 905-2200. Please direct advertising and editorial inquiries to this address. SUBSCRIPTION RATE for the United States is $55 for 13 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 and $40 per year for airmail. POSTMASTER: All subscription orders, inquiries, and address changes should be sent to EMBEDDED SYSTEMS PROGRAMMING, P.O. Box 1278, Skokie, IL 60076-8278. For customer service, telephone toll-free (888) 847-6177. Please allow four to six weeks for change of address to take effect. Periodicals postage is paid at San Francisco, CA and additional mailing offices. EMBEDDED SYSTEMS PROGRAMMING is a registered trademark owned by the parent company, Miller Freeman Inc. All material published in EMBEDDED SYSTEMS PROGRAMMING is copyright © 1999 by Miller Freeman Inc. 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, MI 48106, (313) 761-4700.

Debuggers, In-Target Probes & In-Circuit Emulators for Intel Processors



It's hard to compete without the right tools.

As with most things, there's a hard way and an easy way to develop any new Intel processor based system. And just for the record, we're the guys who offer the easy way.

We've been working closely with Intel for 9 years to make sure that powerful new American Arium development tools are ready and waiting each time early silicon is released. So whether you're developing a Pentium® processor based embedded system or a server designed around the new Itanium™ processor, we've got the affordable and readily upgradeable in-circuit emulator you need. Features include:

- New! GNU support
- Unmatched breakpoints
- Industry leading debugger
- Fast setup and use
- 1 year of software upgrades and tech support

Why not call us today and put your system on the fast track to market.

american arium

14811 Myford Rd., Tustin, CA 92780
Ph: 714-731-1661 Fax: 714-731-6344
E-mail: info@arium.com

For more information on any of our in-circuit emulators, visit...

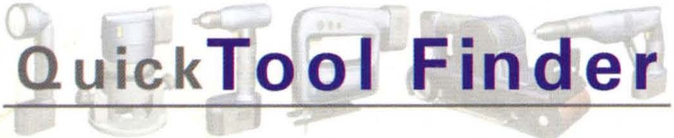
www.arium.com

Pentium and Itanium are registered trademarks of Intel Corporation.

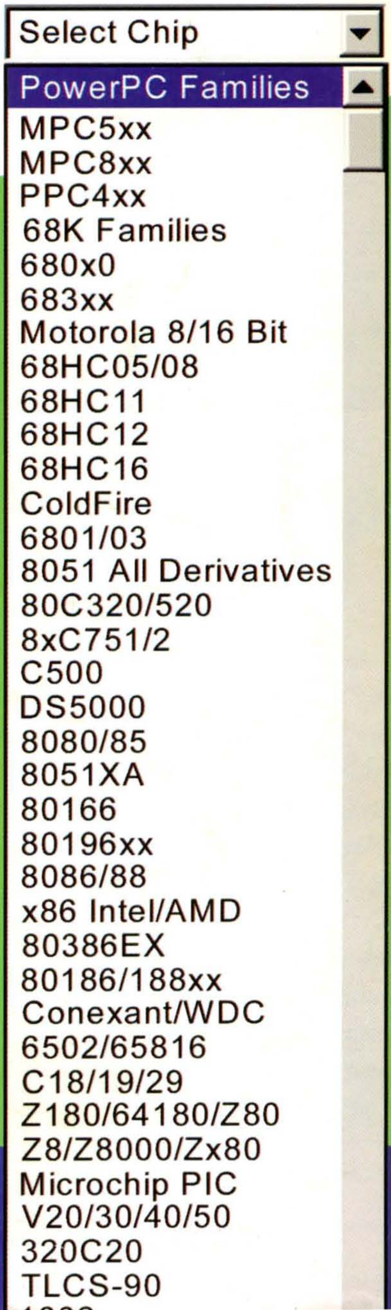
Compilers Assemblers Simulators RTOS In-Circuit Emulators BMDs

www.avocetsystems.com

Every Chip, Every Tool, One Source



QuickTool Finder



Select Chip

PowerPC Families
MPC5xx
MPC8xx
PPC4xx
68K Families
680x0
683xx
Motorola 8/16 Bit
68HC05/08
68HC11
68HC12
68HC16
ColdFire
6801/03
8051 All Derivatives
80C320/520
8xC751/2
C500
DS5000
8080/85
8051XA
80166
80196xx
8086/88
x86 Intel/AMD
80386EX
80186/188xx
Conexant/WDC
6502/65816
C18/19/29
Z180/64180/Z80
Z8/Z8000/Zx80
Microchip PIC
V20/30/40/50
320C20
TLCS-90

AVOCET
SYSTEMS, INC

(800) 448-8500

sales@avocetsystems.com