january 2000 www.embedded.com volume 13, number 1

Embedded Systems® PROGRAMMING

An MCU/DSP Balancing Act

Choosing the right solution

Assembly Language Programming

How to Increase **Interrupts**

Calculating **CRCs**

Minimization Techniques

The **JetSend** Protocol

Internet Appliance Design:
Embedding TCP/IP

A MILLER FREEMAN PUBLICATION

Cut Development Time

Using In-Circuit Emulators and BDM's

Macro
File Edit View Macro Debug Help
Immediate
Watch
Stack
Loaded
NOHAU
HSP
NEW
seeHau
User Interface
nohau

```
[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"
```

Paused.

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

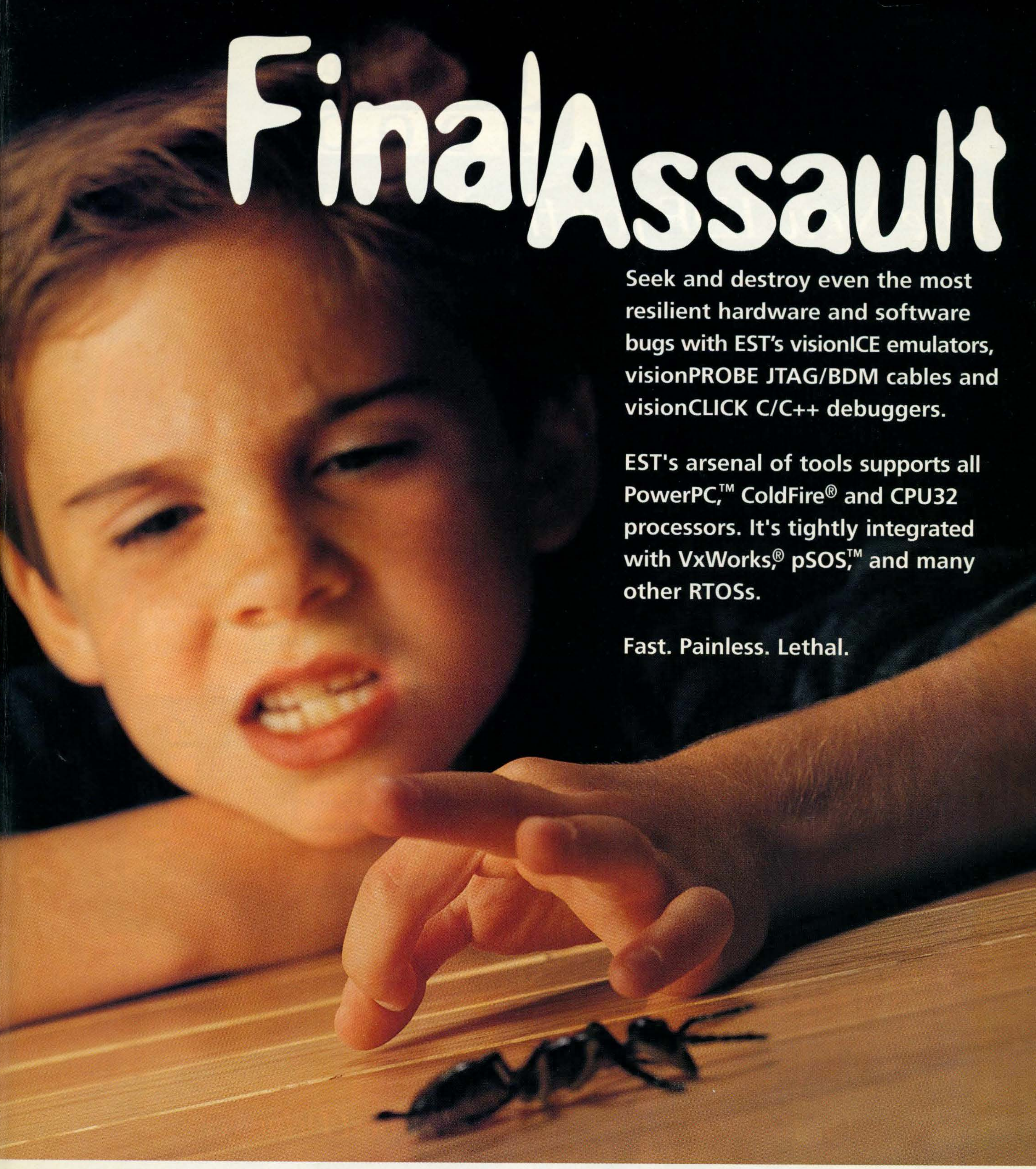
Nohau Supports These Microcontroller Families

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



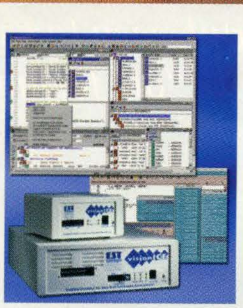
Final Assault

Seek and destroy even the most resilient hardware and software bugs with EST's visionICE emulators, visionPROBE JTAG/BDM cables and visionCLICK C/C++ debuggers.

EST's arsenal of tools supports all PowerPC,™ ColdFire® and CPU32 processors. It's tightly integrated with VxWorks,® pSOS,™ and many other RTOSs.

Fast. Painless. Lethal.

visionICE



visionCLICK

visionPROBE

PowerPC™ 68K ColdFire

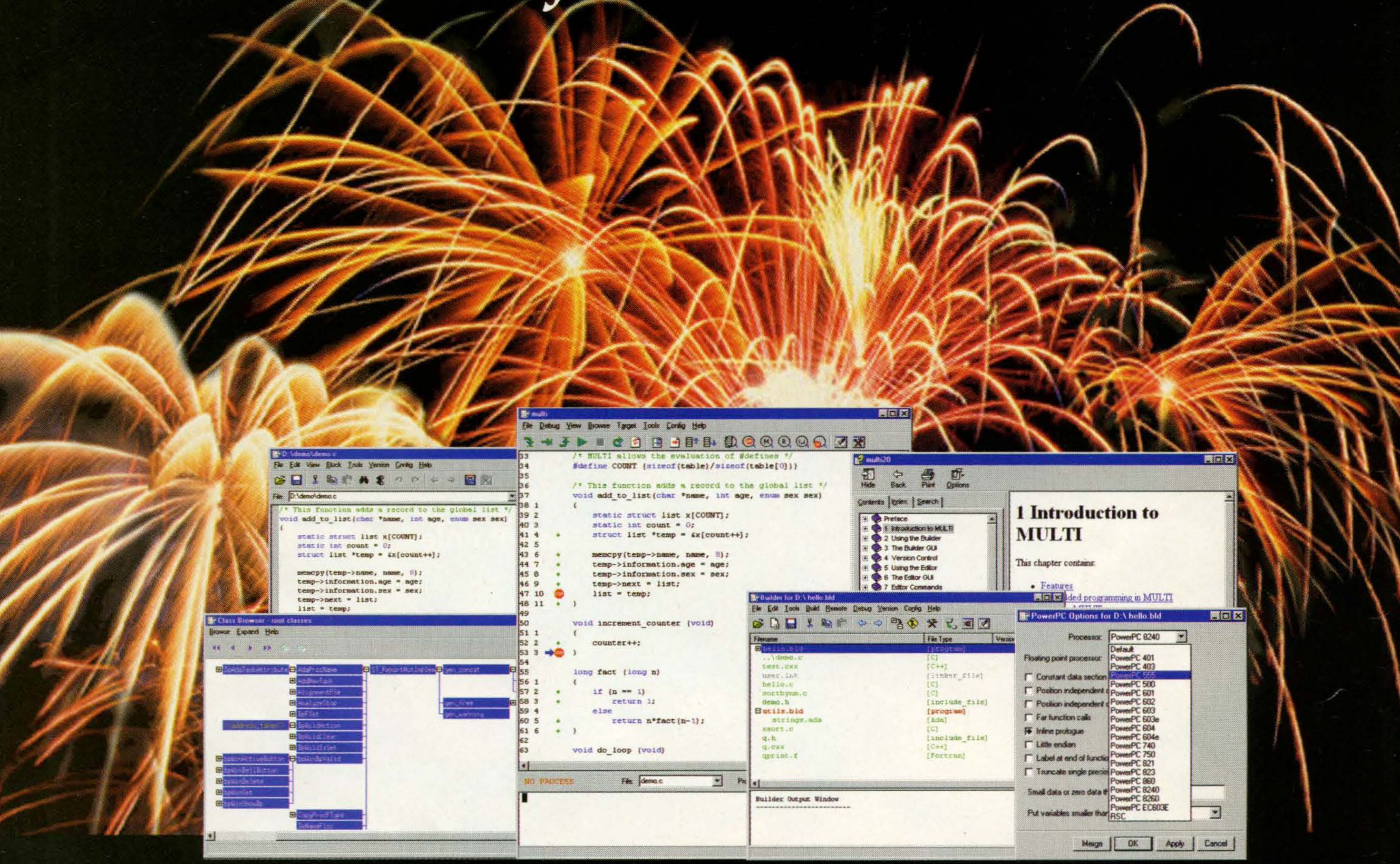
EST

Embedded Support
Tools Corporation

Call 800-957-5588 or Download a Free Demo: www.estc.com

MULTI® 2000

The New IDE for the New Millennium



```
/* MULTI allows the evaluation of #defines */
#define COUNT (sizeof(table)/sizeof(table[0]))

/* This function adds a record to the global list */
void add_to_list(char *name, int age, enum sex sex)
{
    static struct list x[COUNT];
    static int count = 0;
    struct list *temp = &x[count++];

    memcpy(temp->name, name, 8);
    temp->information.age = age;
    temp->information.sex = sex;
    temp->next = list;
    list = temp;
}

void increment_counter (void)
{
    counter++;
}

long fact (long n)
{
    if (n == 1)
        return 1;
    else
        return n*fact(n-1);
}

void do_loop (void)
```

Filename	File Type
hello.bld	[program]
..\demo.c	[C]
test.cxx	[C++]
user.lnk	[linker_file]
hello.c	[C]
sortbynum.c	[C]
demo.h	[include_file]
utils.bld	[program]
strings.ada	[Ada]
xsort.c	[C]
q.h	[include_file]
q.cxx	[C++]
qprint.f	[Fortran]

1 Introduction to MULTI

This chapter contains:

- Features

Processor: PowerPC 8240

☒ Inline prologue

- New and Improved GUI
- Graphical Browser
- On-Line Help
- Syntax Coloring and Auto Indenting
- Simulator
- Version Control
- C++ Debugging
- EventAnalyzer
- Project Builder
- Code Coverage
- Run-Time Error Checking
- Profiling

The Best Compilers. *The Best Debuggers.*

Now, all wrapped up in a brand new, powerful, easy to use IDE. Once you try MULTI 2000, you won't want to struggle with "old fashioned" tools ever again. Contact us now for a Free Evaluation CD and start the millennium with a bang!

Green Hills
• SOFTWARE, INC. •

Tel: 805.965.6044 • Fax: 805.965.6343 • Email: sales@ghs.com • www.ghs.com

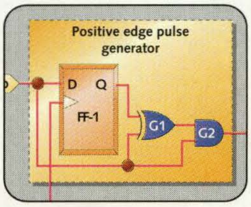
EmbeddedSystems
PROGRAMMING

contents

JANUARY 2000

Cover Story

internet appliance design

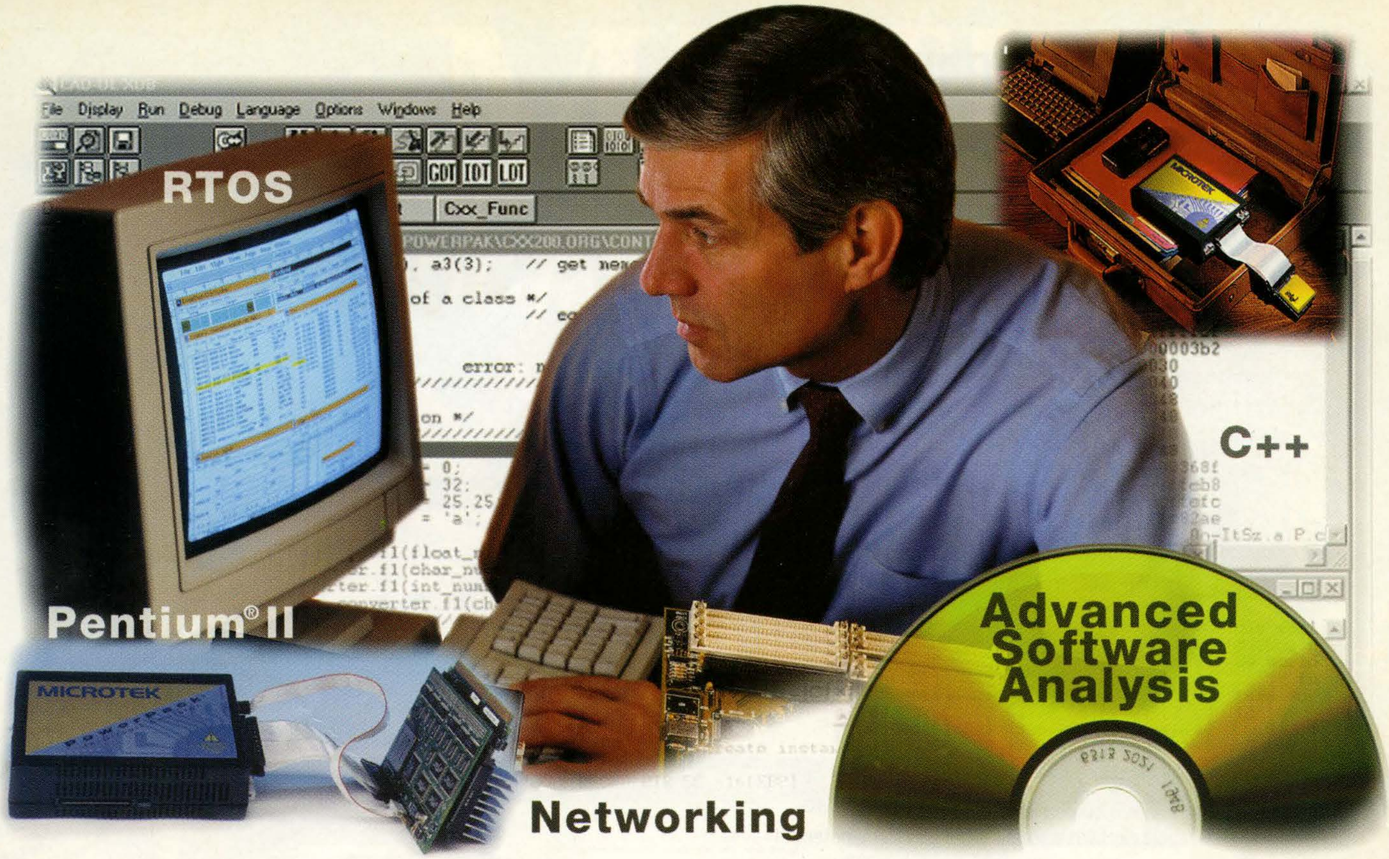


82
A serial output interrupt expander can stretch an MCU's usefulness.

columns

departments

COVER ILLUSTRATION BY RUPERT ADLEY



"Microtek in-circuit emulators gave me more features and the best value."

You want to do it right, and meet project deadlines. That's not easy, especially when your debug environment includes RTOS and C++ support. And more hardware hides deeper inside silicon.

You can't escape these challenges, and neither can we. That's why we pack more features and interoperability into our emulators.

Only Microtek emulators offer integrated state analysis with clock-cycle time stamping – *a traditional logic analyzer feature.* This allows you to track down the causes for system crashes that are impossible for a software debugger to find. At the same time, we have integrated high-level source display... *the kind logic analyzers can only show with a complicated connection to the target, multiple tool interfaces and non-interactive, post-processed source display.*

You can quickly view executed code coverage from your emulator without a separate tool interface, and without instrumenting your code or maintaining separate builds. Plus, now you can run your Microtek emulator from any seat on the network.

No other emulator offers integrated state analysis. No other emulator offers our advanced trace and trigger system. No other emulator offers SWAT™ Software Analysis Tools.

While other emulator companies are de-emphasizing their emulator business, we continue to innovate. This means you can choose the performance options you need – *from entry-level to the most advanced in the industry, all at competitive prices.*

MICROTEK
IN-CIRCUIT EMULATORS

1 (800) 886-7333
www.microtekintl.com

Phone: (503) 533-4463
Fax: (503) 533-0956
Email: info@microtekintl.com

Support
for the following processors:

NEW Low Voltage Pentium®
Pentium® II
Pentium®
Intel486™
Intel386™EX/CX/SX
Intel 186/188
683xx/HC16
8051/52/251

Call for a Free AppNote on these topics:

1. Using Microtek Emulators with WindRiver's Tornado™
2. How to Emulate a Single Target System on a Network
3. Instruction-Level Execution and Branch Code Coverage
4. Emulator Integrates Logic Analyzer State Analysis and High Level Trace

Financing Options Available

EmbeddedSystems
PROGRAMMING

EDITORIAL DIRECTOR
Lindsey Vereen, lvereen@mfi.com

MANAGING EDITOR
Michael Shapiro, mshapiro@mfi.com

TECHNICAL EDITOR
Michael Barr, mbarr@mfi.com
(410) 796-1888

SPECIAL PROJECTS EDITOR
Tarita Whittingham, twhittingham@mfi.com

ASSISTANT EDITOR
Felisa Yang, fyang@mfi.com

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

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

VICE PRESIDENT/ELECTRONICS
Donna J. Esposito

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

PUBLISHER
Eric Berg, (415) 278-5220

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

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

WESTERN ACCOUNT EXECUTIVE
Craig Hammond, (415) 278-5294

SALES ASSOCIATE, RECRUITMENT
Sam Louis, (415) 278-5223

EASTERN REGIONAL SALES MANAGER
Damon Graff, (781) 235-8258

EASTERN SALES
Jared Grimm, (781) 235-8258

PRODUCTION COORDINATOR
James Whitehead

CIRCULATION MANAGER
Jennifer Schuler

CIRCULATION DIRECTOR
John W. Rockwell

SUBSCRIPTION CUSTOMER SERVICE
Toll Free: (888) 847-6177
Non-U.S. subscribers: (847) 647-8602
embedded@halldata.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
Sherry Bloom, (415) 808-3980

EXECUTIVE PRODUCER, EMBEDDED.COM
Leticia Smith, lsmith@mfi.com

ONLINE PRODUCTION COORDINATOR
Billy Biondi, wbiondi@mfi.com

Miller Freeman
A United News & Media publication

CEO/MILLER FREEMAN WORLDWIDE
Tony Tillin

CHAIRMAN/MILLER FREEMAN INC.
Marshall W. Freeman

PRESIDENT
Donald A. Pazour

EXECUTIVE VICE PRESIDENT/CFO
Ed Pinedo

EXECUTIVE VICE PRESIDENTS
Darrell Denny Galen A. Poss
Regina Starr Ridley

SENIOR VICE PRESIDENTS
Wini D. Ragus
John Pearson

SENIOR VICE PRESIDENT/PRODUCTION
Andrew A. Mickus

VICE PRESIDENT/CIRCULATION
Jerry M. Okabe

Consumer Audit
BPA

Visit our web site at
www.embedded.com

Lindsey Vereen

Entertaining Times

"So where are the flying cars?" asks my son Brad, who was raised on bright promises for the new millennium. He expected everyone to be zipping around in an airborne vehicle by this time. Come to think of it, so did I, based on prophecies I was fed a generation earlier. I suppose if the future were easier to predict, we would all have long since made killings in the stock market. What with the vagaries of prognosticating, and since Jeanne Dixon is not in my direct bloodline, I'm not about to make any predictions for the new millennium that would set me up for future ridicule. Instead I'll play it safe and limit myself to noting a few things that may be worth watching in the coming months and years.

For example, how much longer will Moore's Law be in effect? We can assume that semiconductor technologies will continue to reduce transistor size for a while at least, making transistors cheaper and more plentiful. Sooner or later, though, the laws of physics will call a halt to the incredible shrinking transistor. The maneuvers to postpone reaching this limit should prove worth watching.

The machinations in the development tools and RTOS business should provide plenty of entertainment as well. We've seen the first stages of consolidation, most recently with the acquisitions of ISI by Wind River and Cygnus Solutions by Red Hat. How many more mergers will follow?

Will changes in design methodologies brought about by increasing software content in electronic systems and shorter market cycles lead to new tools to support new design methodologies? What will be the impact of executable models in embedded software development? What's the likelihood of actually generating production code with a push of a button? As new tools evolve and gain acceptance, it will be interesting to see if the companies that produce them will be swallowed by the larger tools companies, or whether they will instead grow to compete with these companies.

Microsoft should offer some amusement of its own. Will it be split up, and if so will the disparate parts thrive? What, if anything, will become of Windows CE? Just watching Microsoft jockey for position in the post-PC era ought to give us hours of diversion.

The open source phenomenon should also prove intriguing. Will this trend continue, and if so, what will be its impact for embedded software developers? What will its effect be on software licensing models and on the companies that thrive on license revenues?

Consolidation among the single-board computer companies is also something to watch for. Several mergers have already occurred during the past year or so, and we may see more. Will any companies be left standing besides Motorola and Solectron when the dust settles?

The rate at which bandwidth is increasing is often compared to Moore's Law. Will bandwidth continue to double at the same rate transistors do? What changes will this increase bring about?

Although it may be a curse to say "may you live in interesting times," I suspect that we are at least in for some entertaining times—almost as amazing as flying cars actually appearing in our local automobile showrooms. But no predictions.

lvereen@mfi.com

Jack W. Crenshaw

Maximizing Rigor

A funny thing happened on my way to last month's column: I discovered that I wasn't being rigorous enough. The only saving grace is that it seems that neither has anyone else.

If you read the column ("More on Minimization," December 1999, p. 7), you'll recall that we were talking about ways to seek a minimum of a function, by dividing a given segment into smaller segments. You might also recall that I got seriously bogged down when I reached the question of what to do if the old minimum and all the new probes all yield the same value.

At the time, I offered a solution, which does seem to get us out of trouble most of the time. However, it doesn't satisfy me because it doesn't improve the situation. That is, we haven't gotten any closer to the new minimum, and in fact, might be moving away from it. The only hope is that the new geometry will somehow lead to a better result on the following iterations.

Looking back into the distant past, I now remember that this is why I've never looked very deeply into methods for minimizing functions. I kept running into this potential problem, and didn't see any obvious way out of it. What threw me on this one is that nobody else who has published algorithms seemed to be particularly bothered by the possibility that I've described. I've read almost all the books ever written on computer algorithms, and I've never seen anyone express much concern that the question would ever come up. Certainly, I've seen more than one implementation of one of the most popular and ubiquitous methods for minimizing functions of a single scalar variable, and not one of them deals with the problem as a special case.

One reason I keep mentioning the different, but similar, problem of finding the root of a function is that I admire the robustness which almost all solutions offer. If a function is continuous, once a root has been bracketed by any of these methods, losing it is impossible. That root *will* be found, in time.

I want a minimization technique that is equally robust, and works even when the function being used is pathological. So far, we don't have that. It's this lack of robustness, I now realize, that's been bothering me. Last month, I offered a suggestion as to how to get out of the box, but as I said, it doesn't satisfy. In retrospect, I've decided not to leave it at that. I've decided that we will not continue until I've found a method that *does* meet my standards for robustness. So that's what we'll do this month.

Some of you, especially those readers who are already knowledgeable about methods for minimizing functions, may be wondering when I'm going to stop tinkering around with simple interval-halving schemes, and get to the better known and faster methods that you and I both know exist. To those, I would say, fear not. This series is probably going to be the most extensive and intensive series I've ever done in this column. Rest assured that before we're done, we will have looked at virtually every reasonable algorithm for both scalar and vector (multivariate) arguments. But we must learn to walk before we learn to run, and sometimes it's a good thing to try to squeeze as much information out of simpler methods before moving on to the big time. Please be patient; we'll get there.

Through the use of a more rigorous approach, Dr. Crenshaw digs himself out of a minimization quagmire.

The problem

For the benefit of newcomers, we've been looking at the problem of finding the minimum of some function $f(x)$. Last month, we were looking at a method where we begin with three equally spaced points, the middle of which has a lower value of $f(x)$ than the outer points. The search method involves probing the function at two new points, which split both intervals. If either of these points yields a lower value of $f(x)$, we take this as the new minimum, and reduce the size of the search region. If neither of them is smaller than the original midpoint, we move the boundaries in. Either way, we have the minimum captured and inexorably reduce the boundaries of our search. Eventually, we draw a noose around the minimum we seek.

The point where I got stuck was: what do we do if the value of $f(x)$ is

Countless individual internet appliances.

Imaging	Internet Appliances	Internet Infrastructure
Telecommunications	Intelligent I/O	Wireless
Aerospace/Defense	Automotive	Industrial Control

Whatever your market, we have your embedded software.

One common element.

WindRiver®
SYSTEMS

In the booming market of Internet appliances, success is all about time-to market, cost and product differentiation. If you design to win in that environment, there's one company with whom you'll definitely see eye to eye — Wind River Systems.

With our Tornado™ embedded development tools and VxWorks® real-time operating system, you'll have a huge head start in your product development cycle. Our open, reliable architecture will allow you to differentiate your product for any sized design, all the way from the end-user application code to the look and feel of the graphical user interface.

By utilizing Wind River technology, Liberate Technologies (formerly known as NCI) rolled out its interactive TV software across Europe and the US, and TeraLogic, Inc. successfully launched its Cougar Reference Platform for HDTV. With over 200 Java design wins, we've also made Java work in the embedded market. And our Universal Graphics Layer (UGL) provides a solid foundation for graphics development in the embedded space.

Find out more about Tornado and VxWorks. Visit www.wrs.com/html/espgene.html or call 1-800-545-WIND. Then give your designs the advantage of Wind River Systems. Because your competition is so fierce, you can't afford to blink first.

FIGURE 1 A troublesome function

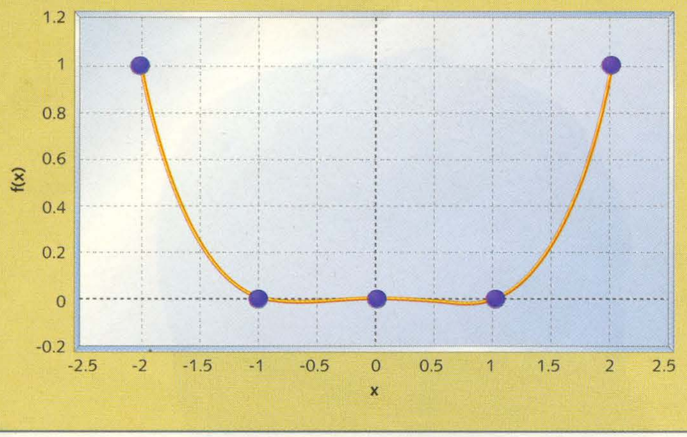
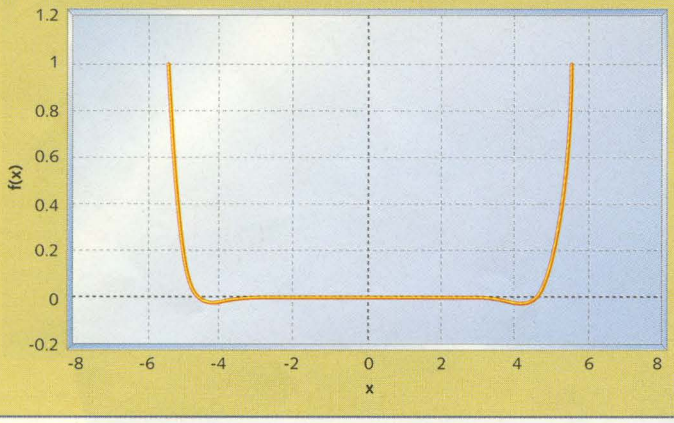


FIGURE 2 More trouble



equal at all three inner points? How do we decide which one of these points to take as the new middle point?

You might be thinking, "But is this really a realistic case? Are there really functions that behave this way?" You bet there are. Here's one that does so, over the range –2..2:

$$f(x) = \frac{1}{12}x^2\left(x^2 - 1\right) \tag{1}$$

The function, which is shown in Figure 1, certainly seems simple enough. It's hardly what one would call pathological. But as you can see, either from the graph or the equation itself, it absolutely must be equal to zero at $x = -1$, 0, and 1.

Imagine that we began the search with the three points $x = -2$, 0, and 2. These are satisfactory points, since the middle point is clearly lower than the other two. So, proceeding with our algorithm of halving the intervals, we probe at $x = -1$ and $x = 1$. Rats! All three inner points are equal, and we are stuck.

Some years ago, I thought maybe I could get out of this box by continuing to subdivide the intervals. I suppose that might work, but it might also take a while. If you think the function of Figure 1 is tough, consider this one:

$$g(x) = \frac{1}{3!\cdot5!\cdot7!}\left[x^2 - \left(\frac{1}{2}\right)^2\right]\left[x^2 - \left(\frac{3}{2}\right)^2\right]\left[x^2 - \left(\frac{5}{2}\right)^2\right]\left[x^2 - \left(\frac{7}{2}\right)^2\right]\left[x^2 - \left(\frac{9}{2}\right)^2\right] \tag{2}$$

I've shown this function in Figure 2, and again, with the region around the x-axis blown up, in Figure 3. This function has no less than *ten* inner points that yield the same value for $f(x)$. Though it's true that I had to construct these two functions to make my point, it's by no means certain that we won't encounter just these same functions, or ones like them, in the real world. The last function makes it clear that we won't be able to escape the problem by simply dividing the interval into more parts.

Which way out?

Let's take another look at Figure 1. We can see that the desired minimum is somewhere around $x = -0.7$. A little calculus, in fact, will show us exactly where it is:

$$\frac{df(x)}{dx} = \frac{1}{6}x\left(2x^2 - 1\right) \tag{3}$$

The function is stationary when its derivative is zero. In this case, we have three possibilities. One is when x itself is zero. But that one turns out to be a local maximum, as the figure plainly shows. The two minima are given by solving:

$$2x^2 - 1 = 0 \tag{4}$$

which gives:

$$x = \pm\frac{\sqrt{2}}{2} = \pm 0.70711 \tag{5}$$

Remember, we've already decided that in the presence of the possibility of multiple roots, we'd prefer to home in on the left-hand one. Therefore $x = -0.70711$ is the solution we seek. But how to find it?

It's tempting to say that if the function is high at P_1 ($x = -2$ in our example), and low at P_2 ($x = -1$), it must be heading south when it gets there. Therefore a minimum must exist in the region between P_2 and P_3, and we'd be safe to pull in P_5 to make the range –2..0. That approach would indeed bracket the minimum for our example, and lead to

FIGURE 3 The details

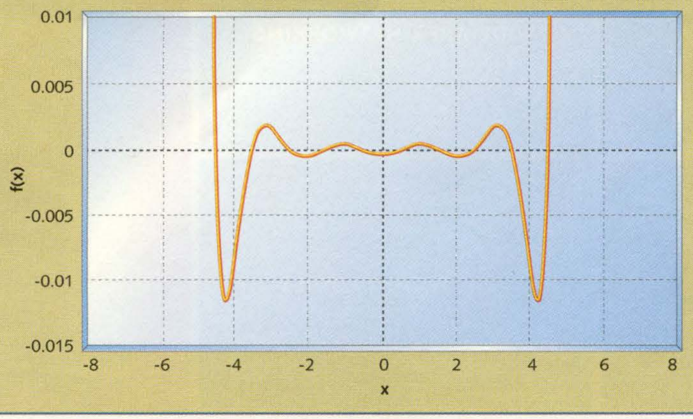
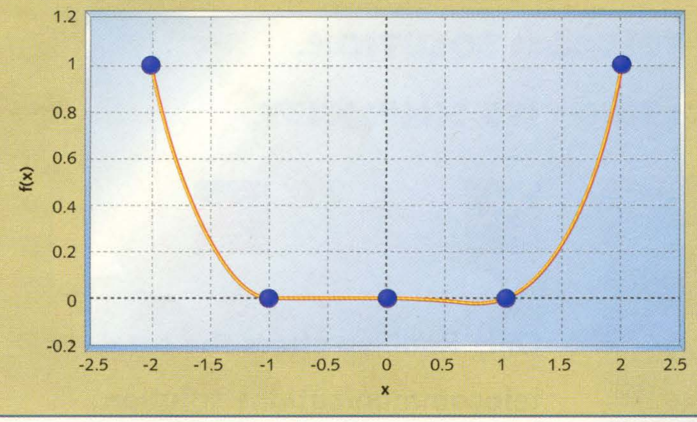


FIGURE 4 Even more trouble



the correct solution. But look at where this would lead us. After the change in range, we are left with the points:

- P_1 = (-2,1)
- P_3 = (-1,0)
- P_5 = (0,0)

P_3 and P_5 now have equal *y*-values, and our requirement that the three points always have distinct values, with P_3 being strictly the lowest, has been violated.

The fact that this approach would indeed have resulted in a good solution *for this example* is immaterial. We can't go by the results of a single example, and I'm not willing to risk the success of the method on the fact that I *think* I know what the shape of the function is. Are you?

If you have any doubt that this approach is a Bad Idea, you need only take one look at Figure 4, where I've modified the function only slightly. In this (admittedly artificial, but still normal looking and perfectly feasible) function, the minimum on the right is the only one present. By moving P_5 in to P_3, we would miss it completely, and the minimum would have neatly slipped from our grasp.

Last month, I suggesting taking the leftmost of the equal points, P_2, as the new middle point, and leaving the two end points alone. That was done with the attitude that we have to do *something* to make future iterations more successful than our current one. By throwing off the symmetry of our first step, I was hoping to rattle the cage of the minimum enough to frighten it out of hiding. In this case, that move would leave us with the three points:

- P_1 = (-2,1)
- P_3 = (-1,0)
- P_5 = (2,1)

At least we have maintained our requirement that the value of $f(x)$ at P_3 remain strictly lower than at the two end points. The approach does, however, have three weaknesses:

- It does not reduce the width of the search area, which we'd like to do at *every* step
- It leaves us with an asymmetric set of points; the intervals are no longer equal
- It will eventually find the *wrong solution*

While it's true that no method short of an exhaustive search can guarantee finding the left-most of multiple solutions, it would certainly be nice if our approach didn't fail to do so on only our second test function.

What's more, we have seen that we can have many more than three points with equal *y*-value. Given any such interval-halving method, what would you like to wager that I can't come up with a function that will hit equal values long enough to drive the method crazy?

Two conclusions seem to be coming clear. First of all, we're probably going to have to abandon the desire to keep our intervals balanced, so that the middle point is halfway between the other two. That desire was already violated by the suggestion just discussed, and it is likely to be unsupportable in general. That conclusion, in and of itself, is important because it leads to a pretty robust and popular method you'll be seeing next month.

The second conclusion is that we need to do something besides halving intervals to help us break out of the equal-value box. We need to do something radical. That something comes from dropping the idea of halving altogether, or at least temporarily.

ASICUS Programmabus.

True to its namesake, Spartan™ FPGAs remain synonymous with "a commitment to winning", regardless of the challenge. And armed with that spirit, Xilinx delivers the Spartan-XL family.

The Spartan-XL architecture offers an easy-to-use alternative to traditional ASICs plus the added flexibility and time-to-market advantages inherent in a programmable logic solution.

Ranging in density from 5K to 40K system gates, the Spartan-XL family features on-chip dual port synchronous RAM, while it's 100+MHz performance and flexible I/O structure make it ideally suited for 32 bit 33MHz PCI applications. All this and more for under $2.50*.

From price to performance, feature set to software, the Spartan-XL device family was born and bred to meet the challenge.

www.xilinx.com

XILINX®

The Programmable Logic Company℠

* 250K units:XCS05XL-4VQ100C

LISTING 2 Minimizing via bisection

```
/* Perform one step of a minimization by double
 * bisection. Input required is three points, of
 * which the middle one is strictly smaller.
 * The points need not be evenly spaced.
 * Function includes protection from equal values
 */
 void bisect(double (*f)(double), double &x0, double &y0,
         double &x2, double &y2,
         double &x4, double &y4){

 // x1 wiggle factor
 double r = 0.95;

 // Bisect the first interval
 double x1 = (x0+x2)/2;
 double y1 = f(x1);

 // Better take care of problem case first
 // Also, better have a way out!
 int i = 20;
 while((y1==y2)&& (i>0)){
   x1 = r*x1 + (1-r)*x0;
   y1 = f(x1);
   —i;
 }
 if(i==0)
   cout << "bisect: cannot find new x1\n";

 // if new point is lower,
 // use as new middle
 if(y1 < y2){
   x4 = x2;
   y4 = y2;
   x2 = x1;
   y2 = y1;
   return;
 }

 // if it's higher, move in left limit
 if(y1 > y2){
   x0 = x1;
   y0 = y1;
 }

 // Bisect the second interval
 double x3 = (x2+x4)/2;
 double y3 = f(x3);

// if new point is lower,
 // use as new middle
 if(y3 < y2){
   x0 = x2;
   y0 = y2;
   x2 = x3;
   y2 = y3;
   return;
 }
```

Breaking out

The way out of the box is to think of continuous functions, rather than sampling at specific points. Taking another look at Figure 1, we can see that, while the function has multiple crossings of the *x*-axis and multiple extrema, the *slope* at P_2 is nonzero. If we had available to us the slope of the function, as well as the function itself, we would know immediately that a minimum sits between $x = -1$ and $x = 0$. We would also know exactly how to arrange further probing values. As a matter of fact, in future columns we'll look at methods that use the analytical slope of the function, as well as its value.

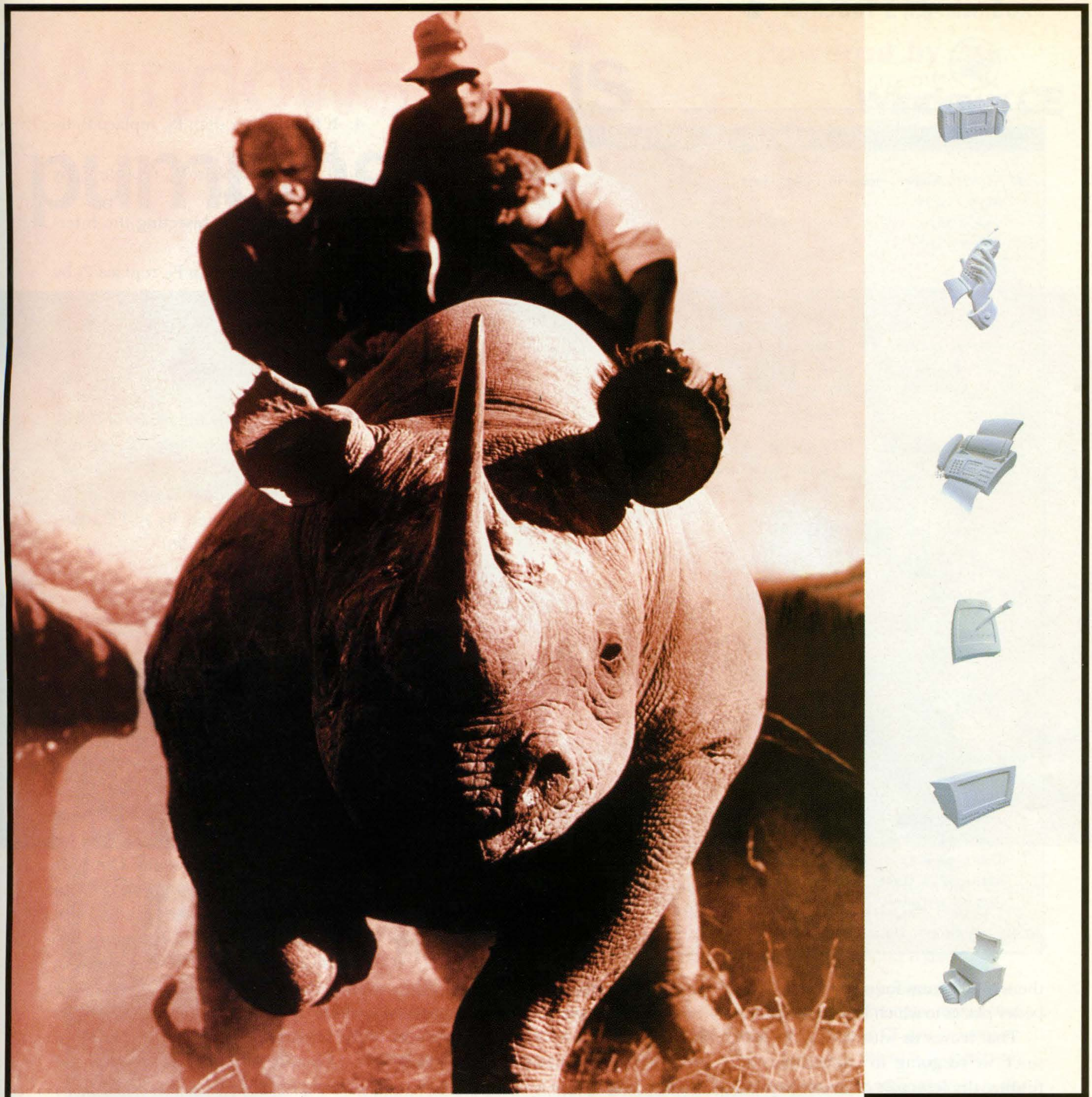
Lacking the analytical slope, is there any way we can tell what to do to get a better set of bracketing points? As a matter of fact, there is: we can sense the slope by "wiggling" one of the points slightly. When we do this, three things can happen:

- The value of $f(x)$ doesn't change
- The value goes down
- The value goes up

In either of the last two cases, we now have a value that's different from the original midpoint. We choose whichever set of points gives us a bracketing condition, and we're out of the box. In the event that the function doesn't change, we must try again, by moving the point (or a different one) some more.

The next questions we should be asking are: which point do we wiggle, and in which direction? Addressing the first point, I think you can see that wiggling the middle point, P_3, is not such a good idea. In the example of Figure 1, that point is a stationary, local maximum, so its slope is zero, at least locally. In Figure 4, the slope is identically zero on the left, all the way to P_2. The last thing we want to do is to merge the two points; that would take us right back to the "left-most of equal points" scheme, which we have already seen doesn't work all that well.

Moving either of the end points, P_1 and P_5, is a *terrible* idea. They are our fences, our bulwarks holding in the elusive minimum. I am not about to move



LISTING 2, continued Minimizing via bisection

```
// if it's higher, move in right limit
  if(y3 > y2){
    x4 = x3;
    y4 = y3;
  }
}

/* Solve for minimum of function, using n successive bisections.
 * Initial points _MUST_ represent legal configuration, i.e.,
 * y0 > y1, y2 > y1.
 * The points need not be evenly spaced.
 */

double minimize(double (*f)(double), double &x0, double &y0,
                        double &x1, double &y1,
                        double &x2, double &y2, double eps){

  // protection from infinite loop
  int i = 30;
  while((fabs(x2-x0)>eps)&&(i>0)){
    bisect(f, x0, y0, x1, y1, x2, y2);
    —i;
  }
  if(i==0)
    cout << "iteration failed" << endl;
  return x1;
}

void main(void){
  double x0 = 0;
  double y0 = f(x0);
  double x1 = 0.5;
  double y1 = f(x1);
  double x2 = 1;
  double y2 = f(x2);
  cout << minimize(f, x0, y0, x1, y1, x2, y2, 1.0e-6);
}
```

them until I know for certain that I have better places to which I can move them.

That leaves us with P_2 and P_4, and since we're going to concentrate on finding the left-most of multiple minima, we'll choose P_2.

Now we have the question: which way do we move it? It's no good trying to move it to the right; that would get us in the same trouble as moving P_3 to the left. Eventually, the two points could merge. On the other hand, moving P_2 to the *left* is a great idea.

Consider this: we know that the height of P_1 is higher than that of P_2. We also know that the function is (or is supposed to be) continuous. If we move P_2 leftwards, we eventually approach P_1 closer and closer, and the function absolutely has to eventually give up a *y*-value closer to that of P_1. It has no choice.

Of course, the minimum could be between P_1 and P_2, so the point could move down instead of up. That's okay. We don't really care which way it moves. Any vertical motion at all distinguishes P_2 from P_3, and we will have enough information to reorder the points.

So here, in a nutshell, is our algorithm, which is about as robust as I can think of:

1. Given three points P_1, P_3, and P_5, with P_3 strictly lower than the other two,
2. Generate P_2 by bisecting the interval $P_1..P_3$
3. If P_2 is the same height as P_3, force it either higher or lower by moving it towards P_1
4. If P_2 is higher than P_1, replace P_1 by P_2
5. If P_2 is lower than P_3, replace P_5 by P_3, and P_3 by P_2
6. Generate P_4 by bisecting the interval P_3..P5
7. If P_4 is higher than P_5, replace P_5 by P_4
8. If P_4 is lower than P_3, replace P_5 by P_3, and P_3 by P_4
9. Repeat until done

There remain only the issues of deciding what we mean by "move P_2 towards P_1" and "done."

As for the moving, I favor reducing the distance between P_1 and P_2 by a fixed ratio, *r*. That is, let:

$$\begin{aligned} x_2' &= x_1 + r(x_2 - x_1) \\ &= rx_2 + (1-r)x_1 \end{aligned} \tag{6}$$

Each step moves P_2 towards P_1, and it approaches P_1 asymptotically, but it never actually gets there, so we needn't worry about the points merging.

To decide when we're done, the usual criterion is to measure the distance from P_1 to P_5 and stop when we feel that it's small enough. The only other alternative I can think of is to keep a record of past values of x_3, and stop when two successive values are nearly equal. This makes me nervous, though, since I can envision cases where x_3 doesn't move much, and yet the minimum is still not pinned down.

A word of warning: don't expect to be able to pin down the minimum as tightly as you're used to pinning down, say, a root of a function or a polynomial. In the end, as we get closer to the minimum, every continuous function looks like a parabola, and the tighter we pull in the range, the flatter the parabola becomes. If you try to push the technology too far, it's possible to get into the range where the parabola looks like a straight line or, worse yet, a noisy discontinuous function, dominated by floating-point round-off error. A good rule of thumb is to avoid narrowing the range down lower than several times the square root of the resolution you can expect in $f(x)$.

Windows CE is pumping gas.

Powered by Microsoft® Windows CE

Radiant™
SYSTEMS

At twelve gallons of gas per minute, you need a robust operating system to keep track of every drop of inventory. Radiant Systems and Tokheim know that real time is the only time for gas pumps. They know that Microsoft® Windows® CE is the real time embedded operating system. They proved that Microsoft Windows CE can track gas to one thousandth of a gallon.

Your customers demand reliable, customized, network-ready devices yesterday. Get an operating system that delivers. Microsoft Windows CE is powerful, modular, connected. And developers are familiar with its Win32® API. That's why Windows CE was chosen by Radiant Systems to power the electronics for Quantium, Tokheim's line of fuel dispensers.

Windows CE brings the power and flexibility of the familiar Win32 API to the embedded world. Now you can use the world-class desktop API anywhere. That's why Radiant Systems, Tokheim, and other leading embedded developers are selecting the Windows CE operating system. With a community of over five million Windows programming developers, a plethora of core operating system services and sample code, you'll find the resources you need.

Windows CE supplies you with a comprehensive mix of communication and connectivity options. Sophisticated security like CHAP, SSL, and Digital Certificates. Internet protocols and a world-class Web browser with dynamic HTML, cascading style sheets, and JScript® software.

Win the race to market. Power your next generation of intelligent devices with Windows CE.

Visit www.microsoft.com/windowsce/embedded

Where do you want to go today?® **Microsoft**

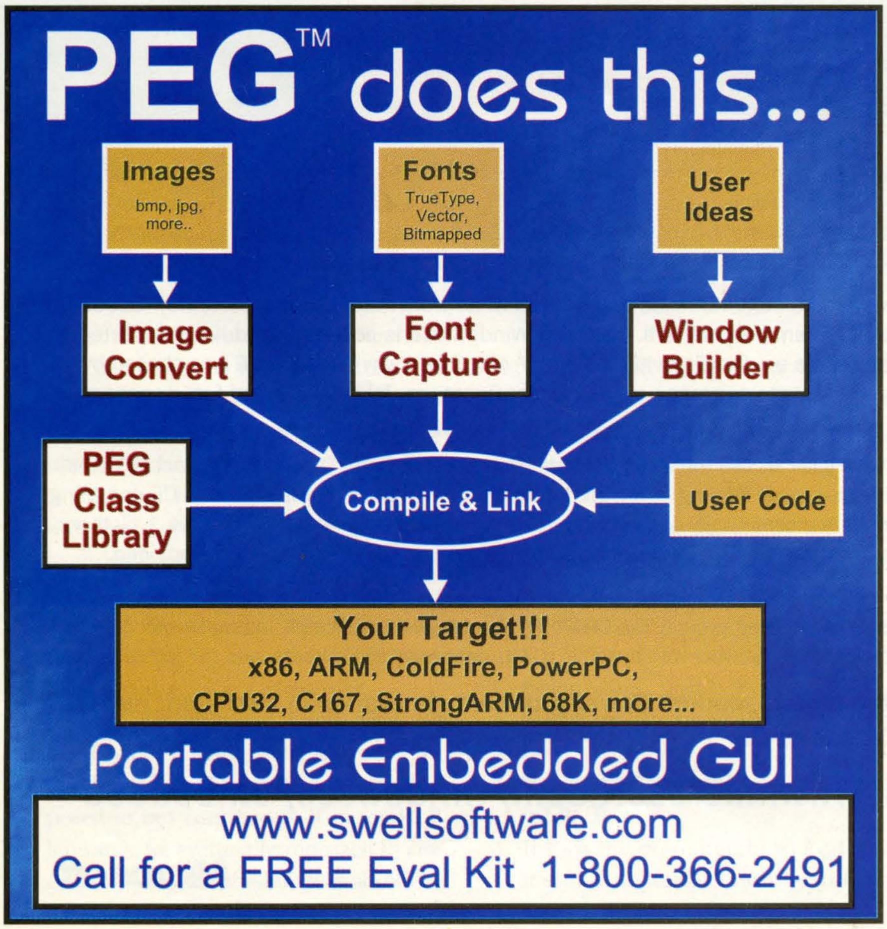
Using these principles, I've written a new version of our function `minimize()`, designed to locate even the most elusive minimum. The code is shown in Listing 1. Except for the wiggling of P_2, and the new halt condition based upon the interval, it's pretty much the same as you saw last month. I've added some protection for infinite loops. Trust me, they *are* necessary, as I found out the hard way. If you make the new error criterion, `eps`, too small, our wiggle logic breaks down because all the function values are equal. Without the loop count, the wiggle loop will run forever.

You'll note that I've resorted to two error messages in case things go wrong. Normally, I try to avoid error messages since we're supposed to be talking about software suitable for embedded systems. In this case, however, we absolutely need to have some way of knowing if the search procedure failed to converge. For use as an example, the error message approach is okay. In a true real-time system, you would surely have some other mechanism, such as a status word, to report error conditions. Such a mechanism is far too implementation-specific for me to use it here, so we'll have to take error messages and live with them.

Where do we go from here?

Let's take a moment to review where we've been, and speculate as to where we're going. To find the minimum of a function $f(x)$, we began with a simple approach that simply divided an initial interval into N points, evaluated all N of them, and took the lowest as the solution. That method was certainly effective—it's the only one guaranteed to find a global minimum, so don't discard it as being too primitive. But it's also slow, and gets slower as we try to improve accuracy.

We made one simple improvement to that approach, which was to skip out of the loop after finding the first minimum (thereby destroying the method's ability to find the global minimum, but also gaining speed).



Our next approach was to seek to speed things up by requiring fewer evaluations of the function $f(x)$. Instead of searching the entire search space with fine resolution, we started out with a rather course search (10 points), and used that to narrow down the region to search for the next step. Calling this function allowed us to iteratively refine the estimate of the minimum.

Last month and this month, we have managed to reduce the number of points in a given search from 10 to five. We built an algorithm that worked last month, and we got one that was robust this month. The obvious next question is: can we reduce the number of points even further?

The answer is yes. We can't reduce it to three; we've already been down that road, and learned that it's bad practice to throw out any point until we're absolutely sure we have improved the situation at every step. We can, however, reduce the number of points to four.

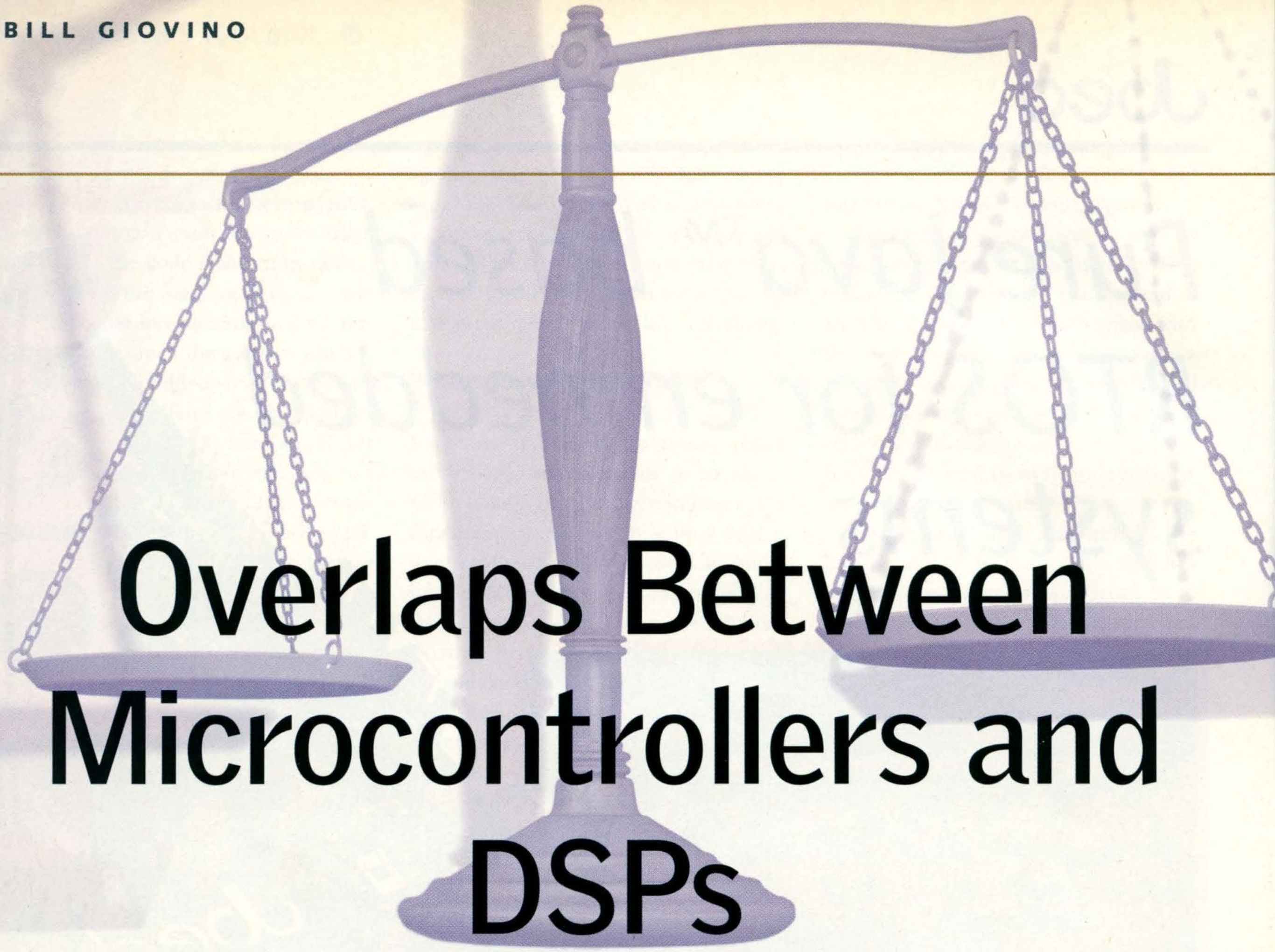
Recall that the current algorithm bisects both intervals—the one from P_1 to P_3, and from P_3 to P_5—in a single pass. This turns out to be unnecessary. With our newfound ability to deal with equal-height cases, we can be sure that each interval is reduced in size on each pass. Given that, we can alternate between P_3 and P_5, bisecting (or otherwise dividing) each interval on alternate passes. That reduces the number of points we must maintain to four: three passed in and out, and one used internally to narrow the search.

We'll look at that technique next month. In the process, we'll (re)discover the famous Golden Ratio search, which will benefit as much from our equal-value fix as the current algorithm does.

See you then. **esp**

Jack W. Crenshaw a senior principal design engineer at Alliant Tech Systems Inc. in Clearwater, FL. He holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at jcrens@earthlink.net.

BILL GIOVINO



Overlaps Between Microcontrollers and DSPs

Microcontollers are primarily used in applications that are interrupt-driven, sensing and controlling external events. You can usually find DSPs in systems that require the precision processing of analog signals. This article describes how traditional DSP and MCU applications are crossing over into each other's territories.

Intel developed the 4004 microprocessor in 1971 for the Busicom desktop calculator (remember when a calculator took up a desktop?) and sold it for $200 a chip. It ran at 92.5kHz internally. Intel's initial strategy was to use this device to sell more memory chips. It was an unexpected success, and was quickly followed by a flurry of similarly enhanced devices from Intel, Motorola, Zilog, and Texas Instruments. Key to the success of the later devices were features like on-chip memory, I/O ports, and hardware peripherals, enabling these chips to economize PC board space in control-oriented applications.

In the past 15 years, digital signal processing has been seen as a specialized segment of an embedded development marketplace dominated by microcontrollers. Digital signal processors (DSPs) were initially used in highly specialized segments where precision processing of analog signals could not be accomplished effectively using conventional analog circuit components. In 1982, the first DSP, the Texas Instruments TMS32010, proved that this segment existed by combining specialized hardware for accelerating multiplication with a Harvard (dual bus) memory architecture, introducing the architectural enhancements that would be found on later digital signal processors.

Years after the introduction of the two architectures, speculation continues on whether a convergence can ever take place. Techniques are debated, architectures compared, and positions promoted. Many articles and papers have been written on this topic—and you're reading one of them now. The difference is that here I will discuss hardcore, real-world selection criteria, including time to market, track records of semiconductor companies, and quality of development tools.

The players

Deeply embedded microcontrollers are primarily used in control-oriented applications that are interrupt driven,

Traditional DSP applications are as subtle as a freight train—they are brute force mathematical applications, pure and simple.

sensing and controlling external events. The external environment is detected either by digital I/O, interrupt pins, or analog (A/D) inputs. The source of the signals to these pins comes from switches, analog and digital sensors, and status signals from other systems. Each input represents a piece of information on the status of some outside event. Outputs are sent to actuators, relays, motors, or other drivers that control events. In between is the trusty microcontroller, analyzing the inputs and the present state of the system, determining when to switch on and what to turn off. The software that does all this, that makes these decisions, does so in a mostly conditional fashion; that is, conditional jumps and bit manipulation and shifts are the staples of embedded control ("interrupts" is counted as a condition here—program flow is altered on the occurrence of an external event).

DSPs, meanwhile, are traditionally found in systems that require the precision processing of digitized analog signals. The performance goal of a DSP architecture is to perform as many arithmetic operations as possible in the smallest number of cycles. Traditional DSP applications are as subtle as a freight train—they are brute force mathematical applications, pure and simple. Traditional DSPs use complex, compound instructions that allow the programmer to perform multiple operations with a single instruction cycle and increase the amount of useful processing done. For example, most are able to compute one tap of an FIR filter in a single cycle. DSP cores are crafted to be number crunchers, and to that end they must do two things well: first, a DSP core must perform multiple math functions, including multiplication, extremely quickly and second, a DSP needs to continuously feed the data path to the number-crunching computational units (so that they can continue to crunch away). It is pursuit of this functionality that makes the programming model of a DSP look so different when compared to a microcontroller. See Table 1 for more details.

Size matters

Note that in Table 1, the benefits of the hardware features of microcontrollers translate into reduced code size and reduced board space—two issues critical to cost efficiency in embedded applications. Reduced code size results in smaller on-chip memory area; the denser the code, the smaller the chip die area. After all, the semiconductor business is, at its base, a real estate business where the cost is about $300 million per acre. Faster execution is also a decision point, but is more important now than before. By contrast, the benefits of DSP hardware features result in faster execution and improved data throughput. Code size has traditionally not been as significant as execution speed, but this too is changing. (I'll describe this in more detail later.)

Control-oriented systems have traditionally utilized only a microcontroller, but some embedded applications add a DSP accessible to the microcontroller's external memory space to speed processing of math-oriented tasks. Examples include digital

TABLE 1 Summary of embedded processor architectural comparison

Microcontroller
Efficiently resolve complex conditional control situations

System Requirement	Feature	Benefit
I/O control	I/O ports with bit-level control	Efficient (quick, small code) control Direct interface to actuators, switches, and digital status signals
Peripheral communications	Serial ports: SPI, I^{2}C, MicroWire, UART, CAN	Hardware support for expansion and external device communications
Precision control of actuators and motors	Sophisticated timers and PWM modules	Low software overhead control
Quickly resolve complex software program control flow	Conditional jumps Bit test instructions Interrupt priority control	Efficient (quick, small code) program flow
Fast response to external events	External interrupts Multiple interrupt levels	Program control immediately redirected on event occurrence; minimal overhead
Conversion of sensor data	Analog-to-digital converters	Hardware support for external sensors

Digital signal processor
Deterministic software behavior

System Requirement	Feature	Benefit
Software filters	Multiply/accumulate unit Zero-overhead loops	Digital filtering in few cycles
Interface to codecs	High-speed serial port(s)	Hardware support for translation of analog signals
High data throughput from serial ports	Peripheral DMA	Fewer wasted cycles fetching data from serial ports
Fast data access	Harvard architectures and variants	Fast execution of signal processing algorithms

RUPERT ADLEY

motor control, robotics, hard disk drives, feature phones, and some electrical meters. These systems are primarily control oriented, with the DSP acting as a math coprocessor to the microcontroller.

In other applications, the MCU and the DSP share the load equally in managing different parts of the system. This has typically been seen in segments such as some communications and telephony applications.

Choosing embedded cores for standard applications

Microcontrollers in deeply embedded applications (that is, with no external memory) have typically been chosen for their cost-effectiveness. Besides the cost of the product, factors that contribute to cost-effectiveness are code efficiency and the on-chip integration of hardware peripherals.

DSP architectures have typically been chosen for their data throughput and their raw computational power. On-chip peripherals would have gone unused. A priority-based interrupt structure and other real-time control features weren't included because real-time control could interfere with the critical processing of the data stream.

Microcontroller applications didn't require a DSP's expensive performance enhancements. Each system was different, and the microcontroller and DSP were seen as two different animals. But the times, they are a-changin'.

In the past 10 years, an uncountable number of microcontroller and DSP architectures have been introduced, abandoned, enhanced, expanded, copied, and redesigned. A wide variety of embedded complexities are available for engineers' selection, from low-end four-bitters (not too far removed from the 4004) to 64-bit systems-on-a-chip. In each and every case, the customer's time to market has become a critical factor, as competition for finished goods has become fierce. This has caused the embedded engineers' definition of what constitutes a microcontroller or DSP "product" to expand based on new selection criteria for choosing products that will provide the shortest development cycle. Besides the actual silicon, the embedded product definition has expanded to include data sheets, application notes, availability of technical support, breadth of product roadmap and what has become the most significant factor: availability and quality of development tools.

Development tools have become one of the most important factors in choosing an embedded core. The behavior and electrical properties of the core have become a gating factor; after that, the next criteria are the availability, quality, and interoperability of development tools. A 2,000 MIPS 1mW SuperCore for 50 cents is completely useless if the software engineer is unable to program it using the available compilers and emulators. To the software engineer, the microcontroller or DSP isn't just a square piece of plastic—it exists in engineering reality in the user interface of the computer screen and keyboard that hosts the hardware and software development tools. Focus group studies have shown that the quality and interoperability of development tools are now the second most significant selection criteria in core selection, after price/performance (Beacon Technology Partners, Concord, MA, 1997).

This newer, more complex definition of what constitutes a microcontroller or DSP product is not a surprise for established embedded companies like Motorola and Texas Instruments, but for many other companies it is a new paradigm that is frustrating their push for new business.

The rest of the story

All of these points may seem to veer off the original theme, but what's at issue here is ensuring that the DSP or microcontroller being considered comes from a vendor who is committed to the entire product support of your core. Using the selection criteria of only the technical merits of the silicon, important usability issues can be overlooked, which can have devastating consequences. Market windows have been missed, resulting in reduced sales volumes, cancelled projects, and customers who have actually gone bankrupt because they judged the architecture purely on technical merits. Choosing a core isn't a sterile classroom engineering exercise; that impressive datasheet you hold in your hands might have a C compiler that doesn't work well with its in-circuit emulator, and both might be applicable to a previous version of silicon.

The funny things that happen at semiconductor companies

Semiconductor companies are developing microcontrollers with hardware multipliers, barrel shifters, and Harvard architectures. DSPs are being developed with external interrupts, integrated peripherals, and register-based architectures. In reality, semiconductor companies are developing these devices for one of two reasons. First, there may be a new market focus on systems that require signal processing and real-time control. Second, it may just seem like a good idea ("if we build it, they will come").

To understand today's market situation, we must understand how we got here. Semiconductor companies deal with two types of markets: distribution (large number of customers, high effort to manage, low-to-medium fluctuating volumes) and direct customers (small group of customers, manageable in scope, very high predictable volumes). During the growth of embedded cores in the 1980s, the microcontroller explosion was fueled greatly by the automotive marketplace, as devices developed for automotive applications migrated into distribution. This environment made Motorola SPS the biggest supplier of eight-bit microcontrollers today. Most serious players in the microcontroller market either developed products for

the automotive market, went after niche markets, or gambled with emerging markets. (Microchip's PIC is a notable exception, with its successful broad market appeal.) DSPs were developed for the telecommunications and military markets and, for the most part, have remained there.

The late '90s have seen unprecedented opportunities for growth in the semiconductor industry. The explosive growth in personal computing, telecommunications, Internet technologies, telephony, and portable applications have sent many semiconductor companies scrambling to introduce products that can be used in these new, highly profitable segments. These segments are controlled by a growing group of established (and a few new) direct customers.

With the opportunities presenting themselves in these new markets, almost every semiconductor company has undergone multiple reorganizations while trying to keep up with these changing markets. Today, every semiconductor company—with the exception of those focused on very niche markets—has completely remade itself to go after these existing markets. As a result, two types of semiconductor companies have emerged: semiconductor companies that have traceable, long-term focused strategies with strong customer relationships and established track records, and semiconductor companies that don't.

A notable symptom is present in the second, "unfocused" group: the rapid introduction, with fanfare, of new products, to then be quietly withdrawn within two years when their expected share of the market isn't quickly realized. Obviously, then, it is from the first group of focused semiconductor companies that the majority of the processor innovations are coming from, as they have the track record and the experience to service customers in these emerging markets.

Real-time control and signal processing in one

A commonality found in many of the emerging embedded markets is a need to process some form of analog data, whether a communications stream or multimedia information, while at the same time maintaining real-time control of external events. The mixture of the two vary as widely as the diversity of the systems. On the one extreme, a simple data acquisition system using an eight-bit microcontroller may need to perform DTMF encoding/decoding and simulate a 1200 baud modem. On the other extreme, a DSP in a voice compression application may want to change program flow based on external switches and status signals.

The Real Embedded Developer's Toolkit

TASKING understands the value of your time. You've got to get that product to market yesterday, ahead of the competition, ahead of schedule – reducing development costs. Time is money.

When development teams need quality software devleopment tools, they insist on TASKING as the central element of their real embedded developer's toolkit.

Our solutions, available for numerous 8-, 16-, and 32-bit microprocessors and DSPs, feature:

Easy to use embedded development (EDE)

Improved code efficiency from our compiler's target specific language extensions and intrinsic functions

Immediate problem resolution through our RTOS-aware debugger's real-time Graphical Data Analysis and cycle counting, resulting in reduced development time

Open tool environment supports the use of third party tools and custom capabilities through EDE.

You can be assured that your application will execute reliably with lightning speed. You will benefit from the increased productivity among developers using different microprocessors in future projects because of their experience with the core technology in our tools.

Our tools are backed by the quality software support you demand. Call us today at 1-800-458-8276.
Or visit our web site at: www.tasking.com

Solutions for microprocessors including:

8051, C166/ST10, XA, M16C, 196, DSP56xxx, 68xxx, PowerPC, TriCore and CARMEL.

TASKING, Inc.
333 Elm Street
Dedham, MA 02026-4530
U.S.: 1-800-458-8276
The Netherlands: +31-33-4558584
Germany: +49-7152-97991-0
Italy: +39-02-6698-2207
Japan: +81-3-3457-6831
UK: +44-1280-70-6686

TASKING
Tools to Build your Success

TABLE 2 Examples of eight- and 16-bit embedded processors for dual functionality applications

Supplier	Product	Primary usage	Advantages in secondary usage	HLL support	Development tool quality	Mixed application examples
National Semiconductor	Compact-RISC	16-bit MCU	•Hardware multiply •Signed math •Fast accurate A/D •Synchronous I/O	Very good	Good	•Payphones •Airbag module •Precision motor control •Security systems
Analog Devices	ADSP-21xx	16-bit DSP	•External interrupts •Fast context switching •I/O port	Poor	Needs improvement	•Speakerphone •Precision motor control •Occupant sensing
Infineon	C167	16-bit MCU	•Barrel shifter •DMA •Fast A/D	Excellent	Excellent	•Barcode scanner •Mass storage •Airbag module •Powertrain
STMicroelectronics	ST10262	16-bit MCU	•Same as C167 *plus* single-cycle MAC	Good	Very good	•Mass storage •Powertrain
Texas Instruments	C240	16-bit DSP	•Rich MCU peripheral set •Event manager •On-chip flash	Good	Good	•POS terminals •Electronic steering •Motor control
Zilog	Z893xx	8-bit DSP	•MCU peripheral set •Low cost •Assembly language similar to Z8	No	Needs improvement	•Caller ID •Motor control •Magnetic card reader

Even as these systems are relentlessly striving for lower cost, smaller board space, and lower power dissipation, they must also integrate as much useful silicon as possible on the die. The target for the semiconductor companies is then to increase the amount of useful work that is accomplished in every instruction cycle.

In cases where a mixture of both DSP and microcontroller functionality is needed, four choices are available:

- Microcontroller
- DSP
- Microcontroller signal processor (MSP; Infineon TriCore or Hitachi SH-DSP, for example)
- Both a microcontroller and a DSP (ARM Piccolo, for example)

The first two choices are obviously the traditional approaches. Using a pure DSP such as a TI TMS320C54x to do control-oriented applications, or using a conventional microcontroller such as an 8051 to do signal processing, are obviously unacceptable choices. However, some products lend themselves well to light duty in the other segment, as shown in Table 2. Note that this table only looks at conventional architectures.

A number of potential system advantages exist to using one processor for both real-time control and signal processing tasks:

- Reduced board space
- Lower system power consumption
- Lower system cost
- More functionality to the system
- Simplified system development

Light DSP processing in a microcontroller

To many microcontroller engineers, a DSP is an unfamiliar entity. Microcontrollers have regular instruction sets, either accumulator based or register based, with a friendly program model. "Friendly" doesn't mean that the MCU will buy you a Coke; rather, it translates into either lots of general-purpose registers or a small number of specialized ones. DSPs have lots of specialized registers and multiple buses, a concept that makes it more difficult to program compared to a microcontroller's friendlier architecture.

A software engineer would much rather stick with what is a known entity; that is, an engineer familiar with microcontrollers will use a microcontroller to do light-duty DSP tasks. If a microcontroller is available that has all the features and memory needed to perform control tasks while also providing enough performance to satisfy the signal processing requirements, the engineer will favor this solution. If the two critical issues of price/performance and development tool quality are met, the microcontroller would be a first choice for the application, especially if the engineer is already familiar with the development tools and/or the microcontroller. If the engineer has been battered by previous experiences, then the full processor product definition discussed earlier, as well as the company's reputation in the embedded market, will already have been considered.

Signal processing performance in a microcontroller can be enhanced in a number of different ways:

- A fast multiply instruction, or even more preferable, a hardware multiply, allows certain filters to be implemented more efficiently
- Regular cycle execution, in which all instructions execute in the same number of internal clock cycles, enhances deterministic behavior
- Programmable interrupt controllers are useful in control appli-

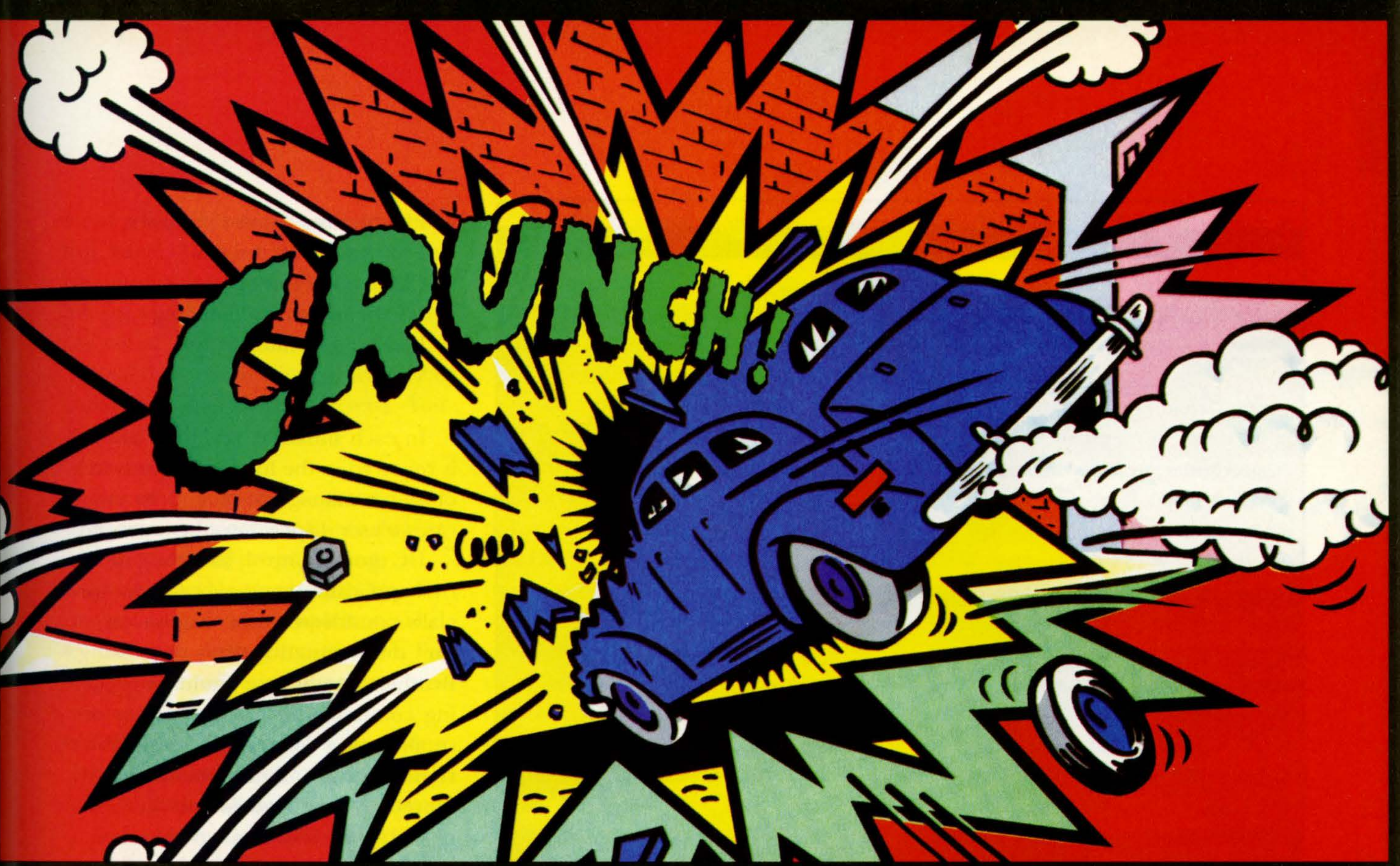

CRUNCH!

TABLE 3 Dual-functionality applications

Application	Real-time control (MCU)	Signal processing (DSP)
Point-of-sale (POS) terminals	•Keyboard capture •Alphanumeric display control	•2400 baud FSK modem •Bar code reader
Payphones	•Keypad decode •Call flow management •Anti-theft control •Vandalism detection •Cash box management	•DTMF decode •DTMF generate •2400 baud FSK modem •Call progress tone detection and generation
Digital feature phone	•Hookswitch detection •Keypad decode •LCD display management •Switch detect	•Vocoder •AGC •Silence detect •DTMF decode •DTMF generate •Echo cancellation
Barcode scanner	•Host communications •Switch detect	•Decode of scan signal •Signal correlation •Transmission of IR signal
Industrial control a.k.a. motion control	•Keypad input •Display management •Switch detect	•Precision motor control •Precision motion control •Torque control
Precision motor control	•Encoder detection •Switch detect •Status signal detect •Limit switch detect	•Digital filtering •Torque control •Vector mathematics •Sensorless control •Acceleration control
Security systems	•Keypad input •LCD display control •Alarm tone •Light sensing	•Noise detection •Magnetic card reader •Vibration detection
Electronic power assist steering (EPAS)	•Vehicle speed sensing •Steering wheel rotation sensing	•Torque algorithms •EPAS algorithm •Feedback loop algorithms
Digital still camera	•Key press and switch detect •LED drive	•Image preprocessing •Image compression
Caller ID boxes	•LCD display management •Keypad management •Switch decode	•Caller ID signal bitstream decode
Cellular phone	•Keypad detect •LCD display management	•Vocoder

cations. If signal processing is also being performed, effective use of a programmable interrupt control unit ensures that signal processing has both interrupt priority and permission levels

- A DMA helps to keep data flowing efficiently through the data paths and can enhance I/O throughput
- A high clock speed facilitates brute-forcing control and signal tasks while still maintaining real-time performance
- A register-based architecture facilitates moving data through the stream

A register-based architecture is certainly preferable to an accumulator-based architecture. Back when the cost of silicon was much more expensive and process technology sizes were specified in whole numbers, an accumulator was an expensive piece of real estate. About the time microcontrollers started being fabricated in 0.8-micron technology, the first general market register-based microcontrollers were introduced. This alleviated the bottleneck of the accumulator and allowed more effective movement of data.

But despite these enhancements, the inability to fetch two pieces of data simultaneously, plus the lack of a multiply-and-accumulate instruction (rarely found in microcontrollers) obviously limits the functionality of most microcontrollers when processing analog data and restricts it to relatively slow to medium speed processing.

Simple signal processing functions that can be performed by some eight- and 16-bit microcontrollers include:

- PID filtering (motor control)
- 2400 baud FSK modem
- Caller ID detection
- DTMF encoding/decoding
- Low tap FIR filter
- Video synch

In each situation, real-time control is required in the form of input switch or status management, and in many cases a user display must be managed.

DC motor control, including industrial applications and robotics, is especially considered to be a viable target for dual functionality processing. Brushless motors are gradually replacing commutation motors as they are smaller, less expensive, and don't require as much maintenance. Moreover, brushless motors generate much lower EMI than commutation motors. The math functions required for the precision control of motors are more applicable to a DSP's hardware than a conventional microcontroller.

Real-time control in a DSP

In the past, no one would have considered a DSP for a microcontroller application because, besides having excessive processing power and insufficient on-chip peripherals, DSPs were simply too expensive for these applications. With shrinkage to submicron and increased on-chip integration on many DSPs, these restrictions are changing.

Systems that require only basic control, used to determine which path of data flow the program will take, are simple and are commonly implemented in signal processing systems. All that is required is efficient implementation of program control routines (conditional jumps and test instructions) and an I/O port for control and detection of external events.

However, a serious gating factor prevents most DSPs from being used in real-time control: lack of necessary peripherals to perform external control and communications. Even most of the light-duty real-time control systems require a minimum peripheral set of a reload timer for software control loops, UART for external serial communications and debug, and an SPI for serial memory expansion and control of external drivers.

Real-time control in a DSP can be enhanced in a number of ways. You can use:

- General purpose registers that streamline control software flow, as well as ease of programming
- Quality development tools for efficient control logic implementation
- In-system debugging features for product verification of control processing
- MCU peripherals such as watchdog timer, multifunction timers, UART, and SPI that allow communications and external control and expansion, which is part of hard-core microcontroller functionality
- True bit manipulations (set, clear, test), not just on core registers but on ports and serial function registers, which significantly enhance data throughput and reduces code size dramatically

The challenge with eight- and 16-bit processors is to perform the entire application, which includes both signal processing and real-time control, in a DSP. A DSP engineer will not hesitate to incorporate control-oriented software into the DSP of a signal processing system, provided, at the outset, that the engineer is convinced that the critical processing and flow of signal data will not be affected. This issue is simple. But microcontroller engineers, despite market surveys to the contrary, will have to be convinced before moving to a DSP for their next projects. As stated before, an engineer totally familiar with DSPs is willing to make a choice mostly based on technical merits. But for a microcontroller engineer, business and development issues far outweigh device technical merits.

Microcontroller engineers tend to prefer a regular memory model that has lots of general purpose registers. A DSP has a few special purpose registers. The friendliest way to program such a memory model is in C. But a traditional DSP programming model doesn't lend itself to C compiler efficiency; quite the contrary, the typical DSP programming model works against implementing significant code size optimizations. Also, the quality of C compilers available for DSPs varies greatly. Many produce stable code with simple optimizations. Some compilers seem okay at first look, but as with many development tools, serious

TABLE 4 Microcontroller signal processors (MSPs)

Product	Processor configuration	Precision		Development tools	Target applications
		Data	Instruction		
Infineon TriCore	Single-core MCU/DSP with configurable memory and I/O	8/16/32	16/32	Excellent	•Disk drives •Powertrain •Digital camera
Hitachi SH-DSP	Single-core MCU/DSP with configurable memory and I/O	16/32	16/32	Very good	•Communications
STMicroelectronics ST100	Single-core MCU/DSP	8/16/32/40	16/32	Excellent	•Disk drives •Powertrain •Telecom •Multimedia
TI 320C27x	Single-core MCU/DSP	16	16	Very good	•Disk drives
Motorola DSP56800	Single-core MCU/DSP	16	16	Good	•Digital feature phone •Two-way messaging •Precision motor control

flaws are discovered six months into development. Some DSP C compilers seem unable to compile even once without generating errors.

Development tools issues aside, most DSPs have built-in features that microcontroller engineers have yearned for for years, such as:

- Zero-overhead loops
- Automatic buffer management
- Bit reversing hardware
- Barrel shifters
- Hardware multiply

From the perspective of a microcontroller engineer, the best way to view a DSP is as an extension of microcontroller migration; that is, move to a DSP from a microcontroller because more processing power is needed and the hardware features make for a more efficient system implementation. The overwhelming reason such efforts are made to convince microcontroller

engineers to program DSPs is simple: microcontroller engineers overwhelmingly outnumber DSP engineers.

A design engineer selecting a microcontroller for a real-time control application can select a micro that fits the system needs and also has a high-quality development environment available. A microcontroller with world-class development tools can be found at every price/performance point. As I stated before, and this cannot be overemphasized, the microcontroller or DSP is not just a square piece of plastic to the software engineer; it exists in engineering reality in the user interface of the computer screen and keyboard that hosts the hardware and software development tools. This point is crucial in reaching a software engineer's heart. The history of processors is filled with cores that had outstanding technical merits but became obsolete because of lower-quality development tools and/or a lack of planning in putting together a suite of tools.

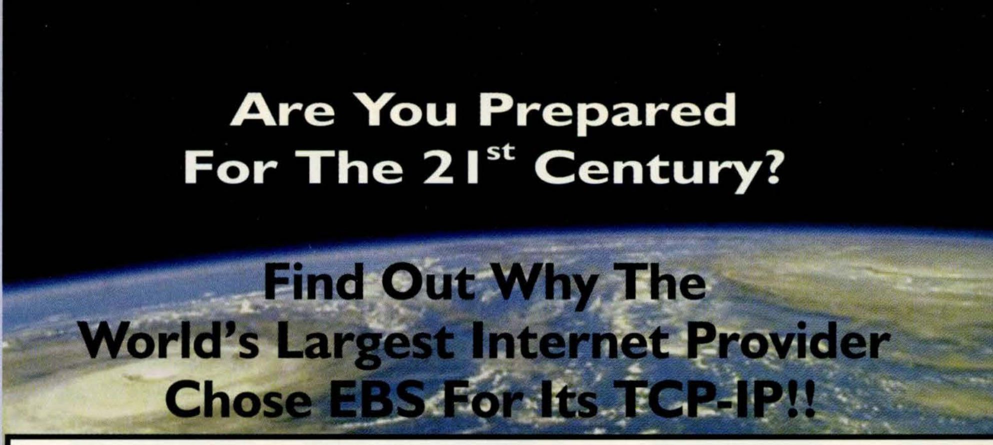
Microcontroller signal processors (MSPs)

To this point the discussion has centered on the deeply embedded applications that use eight- and 16-bit processors. A new breed of high-powered processor has recently been developed. Based on 32-bit microcontroller programming models, these new processors have all the control features and the rich peripheral set of a microcontroller with many of the hard signal processing architectural enhancements of a DSP. This crossbreed of the two programming models is called a microcontroller signal processor, or MSP. The design target of these processors is to build a core that has the following characteristics:

- The register-based programming model of a microcontroller
- Multiple MAC operations in a cycle, with extensibility for more MACs
- Multi-level interrupt priorities for programmable management of critical tasks
- Instructions and architectural enhancements to improve C compiler efficiency

These are cores built to serve both serious DSP and microcontroller functions in a single instruction stream. Both streams are closely related. (This does not include processors with instruction set enhancements such as Intel's Pentium with MMX instructions or Motorola's PowerPC with AltiVec extensions.)

Dual-core implementations of a



Are You Prepared For The 21st Century?

Find Out Why The World's Largest Internet Provider Chose EBS For Its TCP-IP!!

RTIP - EMBEDDED TCP-IP NETWORK STACK

Internet Protocols	- 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, SMTP, POP3, Virtual & Memory File Systems
Web Server	- Full featured embedded Web Server (with diskless option)
Web Client	- Portable embedded Web Client coming soon!

ERTFS - REALTIME FILE SYSTEM

ERTFS NOW SUPPORTS FAT32

Compatible	- Fully DOS compatible
Fast	- Realtime extensions for demanding applications
Complete	- Full file, directory and partition management API
Devices	- Floppy, IDE, ROMDisk, RAMdisk, PCMCIA Cards
Packages	- FDISK, CHKDISK, MKROM Tool

RTPX - REALTIME PORTABLE EXECUTIVE

Tasks	- Dynamically spawn new tasks
Semaphores	- Signal events with counting semaphores
Mutexes	- Exclusive access to resources
Task Control	- Suspend, stop, resume, reprioritize
Porting Layer	- Ports easily to any CPU

CDFS - CDROM FILE SYSTEM

Compatible	- Read data from ISO9660 CDROM drives

Leaders of Embedded Systems Technology

Join the long list of satisfied EBS customers ... when 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.etcbin.com

Embedded Systems Software ebs INC

1-800-428-9340

EBS/EBSnet, Inc.

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

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

Call for our latest brochure with up to date pricing, special features and product demo software!

microcontroller and a separate DSP on the same die is a less than optimal development solution. Dual-core architectures can suffer from the vagaries of interprocessor communications and pipeline interlocks. As for development tools, using conventional debugging techniques to simulate such an architecture is difficult, to say nothing of the cost of a special in-circuit emulator.

The simplicity of merging the microcontroller and DSP architectures into a single instruction stream seemed the natural step. By implementing parallel execution units, high clock speeds, and glueless interfaces for common memories, deterministic real-time signal processing performance for DSPs can exist simultaneously with the multi-level interrupt hierarchy and rapid context switching required for the controller needs of the system. MSPs also have wide data paths, on-chip cache, and 100MHz-plus clock speeds.

Development tools for MSPs are more complex than for microcontrollers, but the instruction sets and architectures lend themselves to easier implementations of C compilers.

At present, few implementations of MSPs are commercially available. Because of the high cost of developing a core with the complexity of an MSP, only the top processor companies have the capability to develop these devices.

Criteria of convergence

The traditionally divergent paths of microcontroller and DSP can cross occasionally, with each performing both tasks. Barriers to entry depend on configuration of the peripheral set, the ability to move multiple pieces of data, and quality of the development tools. Designers of today's emerging systems which require both real-time control and analog signal processing are looking for ways to reduce cost and speed development time in a market that is becoming increasingly competitive. For a microcontroller engineer to consider a DSP or for a DSP engineer to use a microcontroller, three strict criteria must be satisfactorily met: price/performance, peripheral set, and development tool quality. MSPs represent the absolute convergence of these two architectures. They have the most advanced features of both, as well as the fastest clock rates in the embedded market. **esp**

Acknowledgment

The author gratefully appreciates the support and prior research of Olivier Garreau of Siemens Microelectronics.

Bill Giovino is executive editor of Microcontroller.com, *a Web portal and online resource center for design engineers working with embedded systems. He has over 20 years experience in the embedded industry in technical, new product development, and marketing manager roles. Bill works with semiconductor companies as a consulting industry analyst, helping them understand and develop market strategies for devices targeting the embedded systems and embedded Internet markets. You can write him at editor@microcontroller.com.*

References

Eyre, Jennifer, "Diverging Architectures for Digital Signal Processing," *RTC*, September 1998.

Garreau, Olivier and Robert Owen, "Merged Architecture Approach Embeds Digital Signal Processing and Improves Real-Time Performance of Microcontrollers," *Embedded Systems Conference Proceedings*, Fall 1998.

Giovino, William A. "Microcontrollers and DSPs—Will the Two Worlds Ever Intersect?" Online position paper, *www.icspat.com*, 1998.

Texas Instruments, "Migrating μC-Based Control Systems into the New Millennium," Technical Overview, Austin, TX, 1998.

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

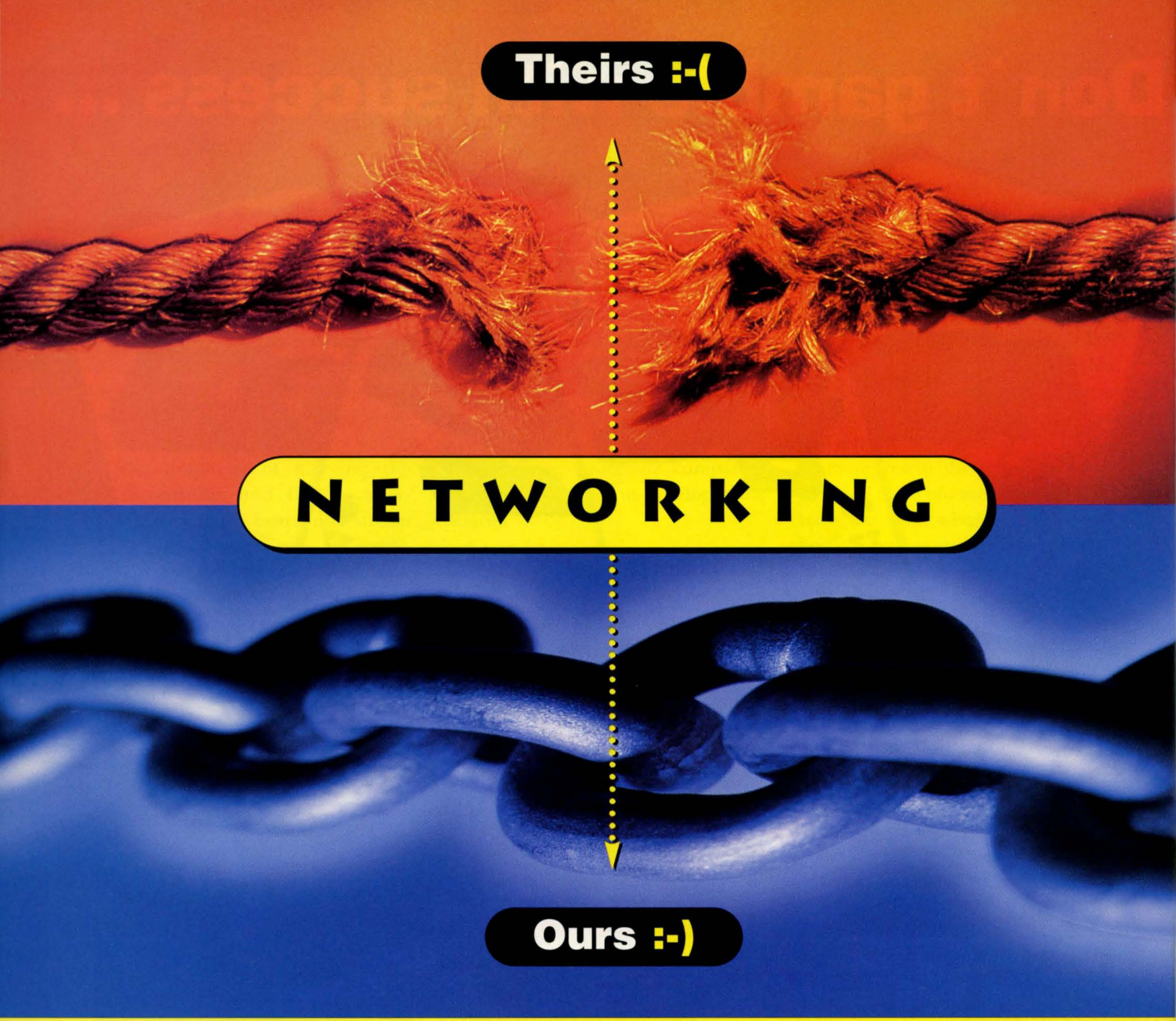
www.hitex.com

"The best emulator I have ever used!"

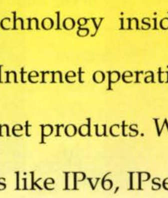
Follow this URL to a special promotion:
http://www.hitex.com/hitools/deal.htm

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



Our networking works! That's why so many leading Internet appliance and application vendors use BSDI. We're the operating system and networking technology inside industry-leading routers, firewalls, load balancers, web accelerators and other Internet devices. BSDI's Internet operating system gives you the reliability, high performance and scalability you need when developing Internet products. We've been building great networking technology since the Internet began, and our advanced features like IPv6, IPsec, packet filtering and rate control make us the most complete networking stack in the industry. If you're developing an Internet product, you need networking technologies that work. **So give us a call at 1-800-345-2731 or +1 719 535 8400, e-mail us at info@bsdi.com, or visit www.bsdi.com.**



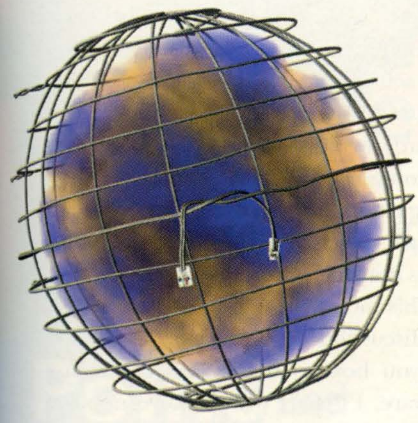
BSD™

BERKELEY SOFTWARE DESIGN, INC.

Berkeley Software Design, Inc. • 5575 Tech Center Drive
Suite 110 • Colorado Springs, CO 80919, USA

CALL US FOR A FREE
EVALUATION CD:
1-800-800-4BSD

Michael Barr



Internet Appliance Design

in this issue

Slow and Steady Never Lost the Race

You can't go anywhere these days without hearing or reading about the "emerging post-PC era." Although the treatment in the popular press often annoys me (to read it, you'd think embedded computing was a concept born yesterday), I do tend to agree that we are in a period of significant changes in how computers are integrated into our society. However, I can't decide which of the changes described by the popular press will be the most important in the long run. In fact, many of the changes anticipated there have already been going on for quite some time. Specifically, the miniaturization of computers and their transformation from general-purpose devices into devices that serve specific needs have both been going on for decades.

I think the single fact that underlies all of this talk of a post-PC era is that the demand for PCs is flattening. The PC as a product class doesn't seem able to make further inroads into consumer homes based solely on lower prices and increased computing power. That old successful formula—and the high profit margins that went with it—are seemingly dead. Most of today's computer buyers already have PCs at home and are reasonably satisfied with the amount of bang they are getting for their buck. And, more and more, all they really want is the cheapest PC in the store anyway.

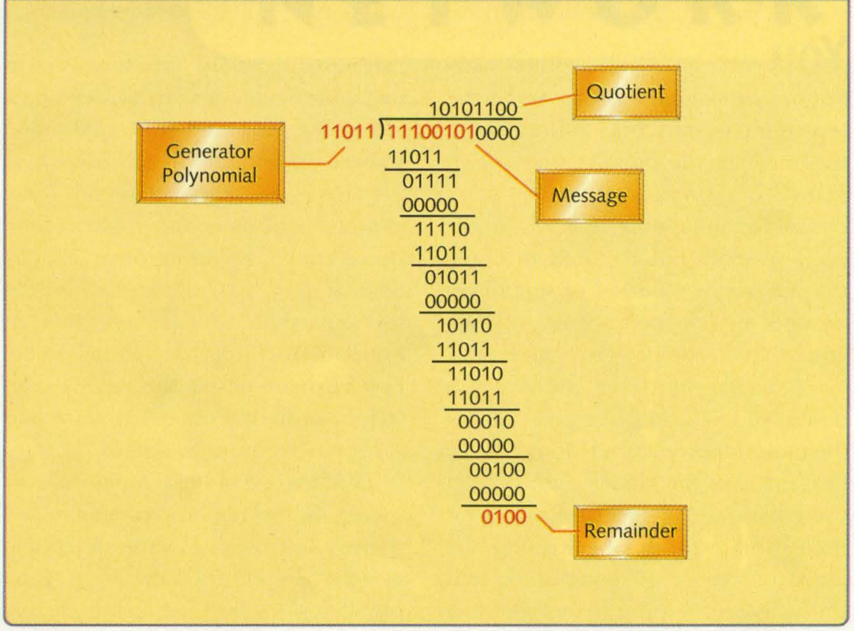
However, one thing that computer buyers still aren't satisfied with is the experience of owning a PC. There are hardware upgrades, operating system upgrades, new device drivers, new application software, and new application software versions, all of which must be installed by the untrained owner. Some of these installations and upgrades have ripple effects, requiring other installations and/or upgrades to be done at the same time. And far too often an installation or upgrade will fail as the new hardware of software is apparently "rejected" by the other hardware and software already in the system.

Perhaps one of the reasons that our society is looking toward embedded systems designers for leadership going into the post-PC era is that we've got a reputation for building computer systems that are stable and easily maintained. It may even be that this reputation is well deserved. I have yet, fortunately, to own a TV, VCR, stereo, digital watch, or microwave oven that required a software or hardware upgrade. However, let's not fool ourselves into thinking we're somehow better than the engineers and computer scientists working in the PC industry. We too are capable of creating systems that crash and that are difficult to maintain. And the more complicated the assignment, the more likely that is.

In my opinion, three key factors led to the complexity of maintaining a PC. The first is the requirement of backward compatibility. Each new PC operating system or processor is expected to run applications designed years before for a very different generation of computers. The second factor is the overall complexity of the problem. The designers of PCs, their operating systems, and applications have tried to capture in one system all of the possible ways a computer can be used. (The task of creating a video game console is much simpler than that of creating a device that can run both business applications and play video games.) Last, but certainly not least, the third factor is the "Christmas buying season." The annual rush to release new hardware and software in time for the Christmas buying season has inspired many engineering compromises. More recently, the phrase "Internet time" has made this into a year-round problem for all of us.

If we are indeed going to take a leadership role in the post-PC era, we need to be humble and to think and act carefully when designing and implementing complex systems. If we do those things, I think we can be counted on to produce a better class of computers that, though less general purpose in nature, are ultimately far more useful and likeable than the personal computers of today. Always remember that you are no smarter than your PC-industry counterpart, only more cautious. And that caution has served you and your past products well.

FIGURE 1 An example of modulo-2 binary division



In this month's Internet Appliance Design section, you'll find a pair articles that fit right into this theme. The first article is the second part of Thomas Herbert's introduction to the TCP/IP protocol suite. This month Tom takes a look at the steps embedded developers can take to make a given TCP/IP stack fit into resource-constrained systems and to improve its overall performance and reliability. The second article, by John Meadows, is an introduction to a different kind of protocol. The JetSend protocol was designed specifically to make the sharing of information between all sorts of different products simple. No special device drivers or format-specific processing software must be present on the receiving system. The result of this could be applications that require no upgrades. My own contribution to this month's section is the final installment of my three-part discussion of checksums.

Easier said than done

As I discussed last month, CRCs are among the strongest checksum algorithms available for detecting and/or correcting errors in communications packets. However, the mathematics used to compute CRCs doesn't map easily into software. In the best possible scenario, CRC computations can be done in hardware with the results passed up to the software for placement into an outgoing packet or verification of an incoming packet's contents. Unfortunately, this is not always possible.

This month I'm going to complete my discussion of checksums by showing you how to implement CRCs in software. I'll start with a naïve implementation and gradually improve the efficiency of the code as I go along. However, I'm going to keep the discussion at the level of the C language, so further steps could be taken to improve the efficiency of the final code simply by moving into the assembly language of your particular processor.

For most software engineers, the overwhelmingly confusing thing about CRCs is their implementation. Knowing that all CRC algorithms are simply long division algorithms in disguise doesn't help. Modulo-2 binary division doesn't map particularly well to the instruction sets of off-the-shelf processors. For one thing, generally no registers are available to hold the very long bit sequence that is the numerator. For another, modulo-2 binary division is not the same as ordinary division. So even if your processor has a division instruction, you won't be able to use it.

Modulo-2 binary division

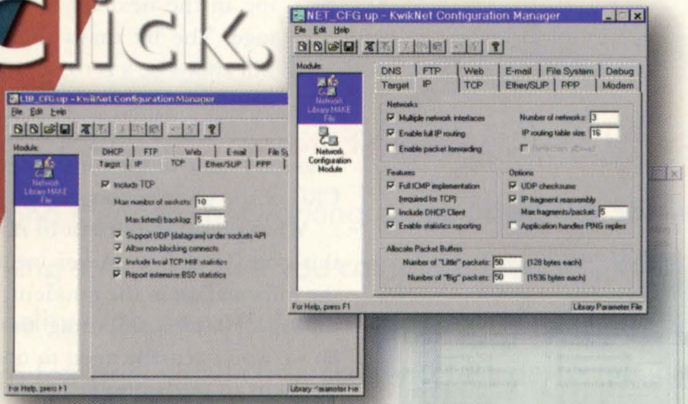
Before writing even one line of code, let's first examine the mechanics of modulo-2 binary division. We'll use the example in Figure 1 to guide us. The number to be divided is the message augmented with zeros at the end. The number of zero bits added to the message is the same as the width of the checksum (what I've been calling *c*); in this case four bits were added. The divisor is a *c* + 1-bit number also known as the generator polynomial.

KwikNet™

Adding network features to your embedded products?

KwikNet makes it quicker, easier, and less expensive.

Just Point and Click.



LIB_CFG.up - KwikNet Configuration Manager
File Edit Help
Module:
Network Library MAKE File
DHCP | FTP | Web | E-mail | File Sy
Target | IP | TCP | Ether/SLIP | PPP
For Help, press F1
Library Parameter File
NET_CFG.up - KwikNet Configuration Manager
File Edit Help
Module:
Network Library MAKE File
Network Configuration Module
DNS | FTP | Web | E-mail | File System | Debug
Target | IP | TCP | Ether/SLIP | PPP | Modem
For Help, press F1
Library Parameter File

LIB_CFG.up – TCP

☑ Include TCP
Max number of sockets: 10
Max listen() backlog: 5
☑ Support UDP (datagram) under sockets API
☑ Allow non-blocking connects
☑ Include local TCP MIB statistics
☑ Report extensive BSD statistics

NET_CFG.up – IP

Networks
☑ Multiple network interfaces
Number of networks: 3
☑ Enable full IP routing
IP routing table size: 16
☐ Enable packet forwarding
☐

Features
☑ Full ICMP implementation (required for TCP)
☐ Include DHCP Client
☑ Enable statistics reporting

Options
☑ UDP checksums
☑ IP fragment reassembly
Max fragments/packet: 5
☐ Application handles PING replies

Allocate Packet Buffers
Numbers of "Little" packets: 50 (128 bytes each)
Number of "Big" packets: 50 (1536 bytes each)

After more than 20 years of leadership in providing real-time multitasking kernels for mission-critical embedded applications, **KADAK** brings the same reliability and ease of use to TCP/IP stacks and web servers. With **KwikNet**.

When you use **KwikNet**, there is no need to spend months of time and incur the expense to manually create and test your TCP/IP stack.

We've already tested it.

And **KwikNet's** Configuration Builder lets you speed through the configuration complexities inherent in any TCP/IP stack!

Plus, as with any product from **KADAK**, you are given a royalty-free site license including source code.

You will also appreciate ***KwikNet's:***

- portability to any RTOS with the minimum required features
- modular ANSI C code which has been tested on a wide range of target processors with 12 popular compilers

And if you add ***KwikNet's*** *Web Server option, you'll enjoy even more benefits.*

Find out all that it can do for you! Visit us at www.kadak.com

KwikNet TCP/IP Stack and Web Server

KADAK

RTOS and Network Products

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

KwikNet and AMX are trademarks of KADAK Products Ltd.

LISTING 1 A first CRC implementation

```
#define POLYNOMIAL      0xD8        /* 11011 followed by 0's */

unsigned char
crcNaive(unsigned char const message)
{
    unsigned char  remainder;

    /*
     * Initially, the dividend is the remainder.
     */
    remainder = message;

    /*
     * For each bit position in the message....
     */
    for (unsigned char bit = 8; bit > 0;  bit)
    {
        /*
         * If the uppermost bit is a 1...
         */
        if (remainder & 0x80)
        {
            /*
             * XOR the previous remainder with the divisor.
             */
            remainder ^= POLYNOMIAL;
        }

        /*
         * Shift the next bit of the message into the remainder.
         */
        remainder = (remainder << 1);
    }

    /*
     * Return only the relevant bits of the remainder as CRC.
     */
    return (remainder >> 4);

}   /* crcNaive() */
```

The modulo-2 division process is defined as follows:

- Call the uppermost $c + 1$ bits of the message the remainder
- Beginning with the most significant bit in the original message and for each bit position that follows, look at the $c + 1$-bit remainder:
- If the most significant bit of the remainder is a one, the divisor is said to divide into it. If that happens—just as in any other long division—it is necessary to indicate a successful division in the appropriate bit position in the quotient and to compute the new remainder. In the case of modulo-2 binary division, we simply:
    - *Set the appropriate bit in the quotient to a one, and*
    - XOR the remainder with the divisor and store the result back into the remainder
- *Otherwise (if the first bit is not a one):*
    - *Set the appropriate bit in the quotient to a zero, and*
    - *XOR the remainder with zero (no effect)*
- Left-shift the remainder, shifting in the next bit of the message. The bit that's shifted out will always be a zero, so no information is lost

The final value of the remainder is the CRC of the given message.

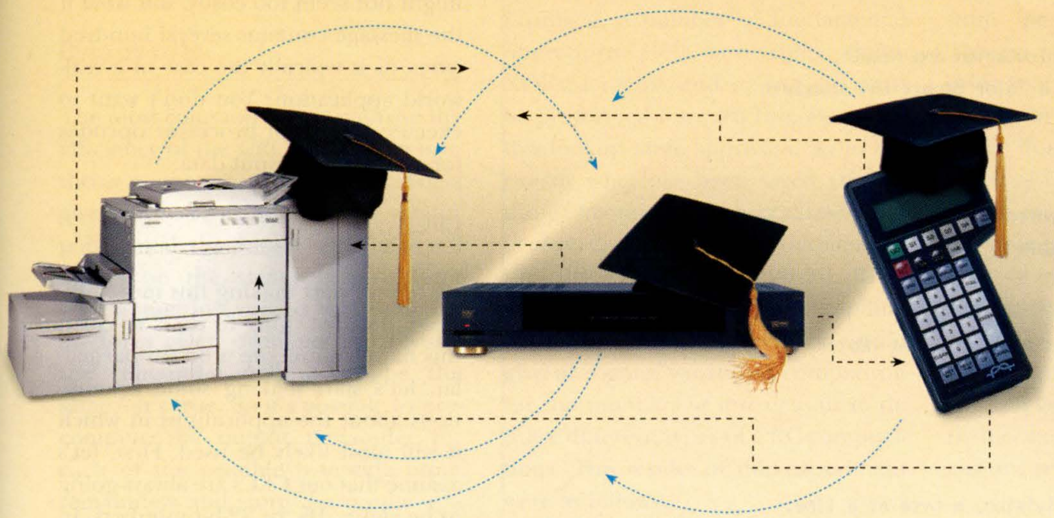
What's most important to notice at this point is that we never use any of the information in the quotient, either during or after computing the CRC. So we won't actually need to track the quotient in our software implementation. Also note here that the result of each XOR with the generator polynomial is a remainder that has zero in its most significant bit. So we never lose any information when the next message bit is shifted into the remainder. All of the parts of the above algorithm that have no effect are written in italics. These steps can be ignored in an actual CRC implementation.

Bit by bit

Listing 1 contains a naïve software implementation of the CRC computation just described. It simply attempts to implement that algorithm as it was described above for this one particular generator polynomial. Even though the unnecessary steps have been eliminated, it's extremely inefficient. Multiple C statements (at least the decrement and compare, binary AND, test for zero, and left shift operations) must be executed for each bit in the

SMART NETWORKING SOFTWARE CREATES

SMART PRODUCTS



Fast, compact embedded networking solutions

Go beyond standard networking capabilities; U S Software's embedded networking solutions give you complete Internet, Web and network functionality with powerful interface control. Our comprehensive software features small code size and fast performance.

NETPEER™

Distributed Operating Environment

This new and powerful distributed computing software provides real time control of your remote Internet appliance – in less than 16K.

- Remote GUI
- Remote database access
- Notification/diagnostic
- Remote file I/O

USNET®

Embedded TCP/IP Protocol Suite

A fast, compact networking suite, USNET is specifically tuned up for embedded processors and supports a variety of operating systems.

- TCP/IP protocols
- Embedded Web Server
- Internet Access Package
- SNMP

Try NetPeer for yourself!
Download our free NetPeer workstation application at www.ussw.com.

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	M*Core
386/486PM	ColdFire
680x0/683xx,	MIPS
DragonBall EZ	C166
PowerPC	CR16
ARM	8096/196
StrongARM	68HC11
SHx	68HC16
SPARCLite	H8/300
i960	8051/80251
NEC V85x	Z80/180

Other smart products from U S Software:

- **SUPERTASK!®** Multitasking RTOS development suite. ITRON 2.x/3.x API available.
- **USFILES®** Windows 95/DOS compatible file system with CompactFlash and CD-ROM support.
- **SOFT-SCOPE®** x86 remote-target debugger for real and protected mode applications.

Call us today at **800-356-7097** and connect with smarter embedded networking solutions.

800-356-7097

WWW.USSW.COM

LISTING 2 A more portable CRC implementation

```
/*
 * The width of the CRC calculation and result.
 * Modify the typedef for a 16 or 32-bit CRC standard.
 */
typedef unsigned char crc;

#define WIDTH    (8 * sizeof(crc))
#define TOPBIT   (1 << (WIDTH - 1))

crc
crcSlow(unsigned char const message[], int nBytes)
{
    crc  remainder = 0;

    /*
     * Perform modulo-2 division, a byte at a time.
     */
    for (int byte = 0; byte < nBytes; ++byte)
    {
        /*
         * Bring the next byte into the remainder.
         */
        remainder ^= (message[byte] << (WIDTH - 8));

        /*
         * Perform modulo-2 division, a bit at a time.
         */
        for (unsigned char bit = 8; bit > 0;  bit)
        {
            /*
             * Try to divide the current data bit.
             */
            if (remainder & TOPBIT)
            {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }
    }

    /*
     * The final remainder is the CRC result.
     */
    return (remainder);

}   /* crcSlow() */
```

message. Given that this particular message is only eight bits long, that might not seem too costly. But what if the message contains several hundred bytes, as is typically the case in a real-world application? You don't want to execute dozens of processor opcodes for each byte of input data.

Cleaning up

Before we start making this more efficient, the first thing to do is to clean this naïve routine up a bit. In particular, let's start making some assumptions about the applications in which it will most likely be used. First, let's assume that our CRCs are always going to be eight-, 16-, or 32-bit numbers. In other words, that the remainder can be manipulated easily in software. That means that the generator polynomials will be nine, 17, or 33 bits wide, respectively. At first it seems we may be stuck with unnatural sizes and will need special register combinations, but remember these two facts:

- The most significant bit of any generator polynomial is always a one
- The uppermost bit of the XOR result is always zero and promptly shifted out of the remainder

Since we already have the information in the uppermost bit and we don't need it for the XOR, the polynomial can also be stored in an eight-, 16-, or 32-bit register. We can simply discard the most significant bit. The register size that we use will always be equal to the width of the CRC we're calculating.

As long as we're cleaning up the code, we should also recognize that most CRCs are computed over fairly long messages. The entire message can usually be treated as an array of data bytes. The CRC algorithm should then be iterated over all of the data bytes, as well as the bits within those bytes.

The result of making these two changes is the code shown in Listing 2. This implementation of the CRC calculation is still just as slow as the previous one. However, it is far more portable and can be used to compute a number of different CRCs of various widths.

Byte by byte

The most common way to improve the efficiency of the CRC calculation is to throw memory at the problem. For a given input remainder and generator polynomial, the output remainder will always be the same. If you don't believe me, just reread that sentence as "for a given dividend and divisor, the remainder will always be the same." It's true. So it's possible to precompute the output remainder for each of the possible byte-wide input remainders and store the results in a lookup table. That lookup table can then be used to speed up the CRC calculations for a given message. The speedup is realized because the message can now be processed byte by byte, rather than bit by bit.

The code to precompute the output remainders for each possible input byte is shown in Listing 3. The computed remainder for each possible byte-wide dividend is stored in the array `crcTable[]`. In practice, the `crcInit()` function could either be called during the target's initialization sequence (thus placing `crcTable` in RAM) or it could be run ahead of time on your development workstation with the results stored in the target device's ROM.

Of course, whether it is stored in RAM or ROM, a lookup table by itself is not that useful. You'll also need a function to compute the CRC of a given message that is somehow able to make use of the values stored in that table. Without going into all of the mathematical details of why this works, suffice it to say that the previously complicated modulo-2 division can now be implemented as a series of lookups and XORs. (In modulo-2 arithmetic, XOR is both addition and subtraction.)

A function that uses the lookup table contents to compute a CRC more efficiently is shown in Listing 4. The amount of processing to be done for each byte is substantially reduced.

As you can see from the code in Listing 4, a number of fundamental operations (left and right shifts, XORs, lookups, and so on) still must be performed for each byte even with this lookup table approach. So to see exactly what has been saved (if anything) I compiled both `crcSlow()` and `crcFast()` with IAR's C compiler for the PIC family of eight-bit RISC processors.[1] I figured that compiling for such a low-end processor would give us a good worst-case comparison for the numbers of instructions to do these different types of CRC computations. The results of this experiment were as follows:

- `crcSlow()`: 185 instructions per byte of message data
- `crcFast()`: 36 instructions per byte of message data

So, at least on one processor family, switching to the lookup table approach results in a more than five-fold performance improvement. That's a pretty substantial gain considering that both implementations were written in C. A bit more could probably be done to improve the execution speed of this algorithm if an engineer with a good understanding of the target processor were assigned to hand-code or tune the assembly code. My somewhat-educated guess is that another two-fold performance improvement might be possible. Actually achieving that is, as they say in textbooks, left as an exercise for the curious reader.

CRC standards and parameters

Now that we've got our basic CRC implementation nailed down, I want to talk about the various types of

CRCs that you can compute with it. As I mentioned last month, several mathematically well understood and internationally standardized CRC generator polynomials exist and you should probably choose one of those, rather than risk inventing something weaker.

LISTING 3 Computing the CRC lookup table

```
crc  crcTable[256];

void
crcInit(void)
{
    crc  remainder;


    /*
     * Compute the remainder of each possible dividend.
     */
    for (int dividend = 0; dividend < 256; ++dividend)
    {
        /*
         * Start with the dividend followed by zeros.
         */
        remainder = dividend << (WIDTH - 8);

        /*
         * Perform modulo-2 division, a bit at a time.
         */
        for (unsigned char bit = 8; bit > 0;  bit)
        {
            /*
             * Try to divide the current data bit.
             */
            if (remainder & TOPBIT)
            {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }

        /*
         * Store the result into the table.
         */
        crcTable[dividend] = remainder;
    }

}   /* crcInit() */
```

In addition to the generator polynomial, each of the accepted CRC standards also includes certain other parameters that describe how it should be computed. Table 1 contains the parameters for three of the most popular CRC standards. Two of these parameters are the *initial remainder* and the *final XOR value*. The purpose of these two *c*-bit constants is similar to the final bit inversion step we added to the sum-of-bytes checksum algorithm two months ago.[2] Each of these parameters helps eliminate one very special, though perhaps not uncommon, class of ordinarily undetectable difference. In effect, they bulletproof an already strong checksum algorithm.

To see what I mean, consider a message that begins with some number of zero bits. The remainder will never contain anything other than zero until the first one in the message is shifted into it. That's a dangerous situation, since packets beginning with one or more zeros may be completely legitimate and a dropped or added zero would not be noticed by the CRC. (In some applications, even a packet of all zeros may be legitimate!) The simple way to eliminate this weakness is to start with a nonzero remainder. The parameter called initial remainder tells you what value to use for a particular CRC standard. And only one small change is required to the `crcSlow()` and `crcFast()` functions:

```
crc  remainder = INITIAL_REMAINDER;
```

The final XOR value exists for a similar reason. To implement this capability, simply change the value that's returned by `crcSlow()` and `crcFast()` as follows:

```
return (remainder ^ FINAL_XOR_VALUE);
```

If the final XOR value consists of all ones (as it does in the CRC-32 standard), this extra step will have the same effect as complementing the final remainder. However, implementing it this way allows any possible value to be used in your specific application.

In addition to these two simple parameters, two others exist that impact the actual computation. These are the binary values *reflect data* and *reflect remainder.* The basic idea is to reverse the bit ordering of each byte within the message and/or the final remainder. The reason this is sometimes done is that a good number of the hardware CRC implementations operate on the "reflected" bit ordering of bytes that is common with some UARTs. Two slight modifications of the code are required to prepare for these capabilities.

What I've generally done is to implement one function and two macros. This code is shown in Listing 5. The function is responsible for reflecting a given bit pattern. The macros simply call that function in a certain way.

By inserting the macro calls at the two points that reflection may need to be done, it is easier to turn reflection on and off. To turn either kind of reflection off, simply redefine the appropriate macro as `(X)`. That way, the unreflected data byte or remainder will be used in the computation, with no overhead cost. Also note that for efficiency reasons, it may be desirable to compute the reflection of all of the 256 possible data bytes in advance and store them in a table, then redefine the `REFLECT_DATA()` macro to use that lookup table.

Tested, full-featured implementations of both `crcSlow()` and `crcFast()` are available for download from *www.embedded.com/code.htm.* These implementations include the reflection capabilities I just described and can be used to implement any parameterized CRC formula. Simply change the constants and macros as necessary.

The final parameter that I've included in Table 1 is a *check value* for each CRC standard. This is the CRC result that's expected for the simple ASCII test message "123456789." To test your implementation of a particular standard, simply invoke your CRC computation on that message and check the result:

LISTING 4 A more efficient CRC implementation.

```
crc
crcFast(unsigned char const message[], int nBytes)
{
    unsigned char  data;
    crc            remainder = 0;

    /*
     * Divide the message by the polynomial, a byte at a time.
     */
    for (int byte = 0; byte < nBytes; ++byte)
    {
        data = message[byte] ^ (remainder >> (WIDTH - 8));
        remainder = crcTable[data] ^ (remainder << 8);
    }

    /*
     * The final remainder is the CRC.
     */
    return (remainder);

}   /* crcFast() */
```

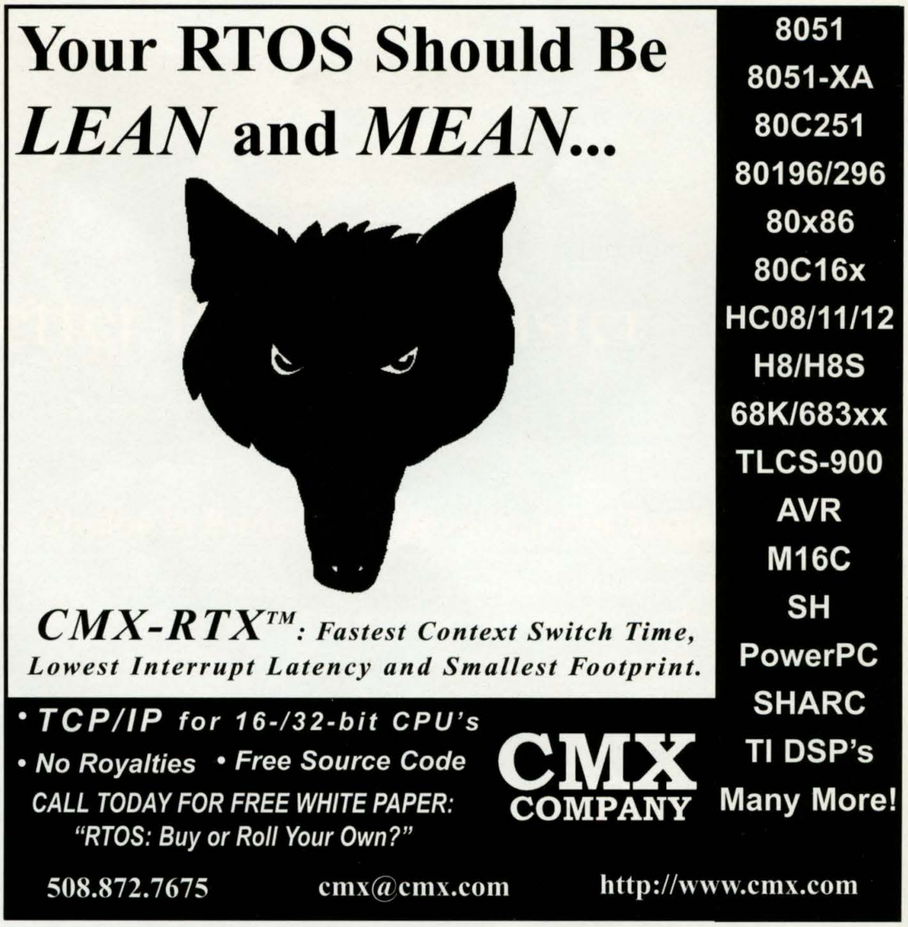


TABLE 1 Computational parameters for popular CRC standards

Standard Name	CRC-CCITT	CRC-16	CRC-32
Width	16 bits	16 bits	32 bits
(Truncated) polynomial	0 x 1021	0 x 8005	0 x 04C11DB7
Initial remainder	0 x FFFF	0 x 0000	0 x FFFFFFFF
Final XOR value	0 x 0000	0 x 0000	0 x FFFFFFFF
Reflect data?	No	Yes	Yes
Reflect remainder?	No	Yes	Yes
Check value	0 x 29B1	0 x BB3D	0 x CBF43926

LISTING 5 Reflection macros and function.

```
#define REFLECT_DATA(X)         ((unsigned char) reflect((X), 8))
#define REFLECT_REMAINDER(X)  ((crc) reflect((X), WIDTH))


unsigned long
reflect(unsigned long data, unsigned char nBits)
{
    unsigned long  reflection = 0;


    /*
     * Reflect the data about the center bit.
     */
    for (unsigned char bit = 0; bit < nBits; ++bit)
    {
        /*
         * If the LSB bit is set, set the reflection of it.
         */
        if (data & 0x01)
        {
            reflection |= (1 << ((nBits - 1) - bit));
        }

        data = (data >> 1);
    }

    return (reflection);

}    /* reflect() */
```

```
crcInit();
checksum = crcFast("123456789", 9);
```

If `checksum` has the correct value after this call, then you know your implementation is correct. This is a handy way to ensure compatibility between two communicating devices with different CRC implementations or implementors.

Other sources

Throughout the years, each time I've had to learn or relearn something about the various CRC standards or their implementation, I've referred to the paper "A Painless Guide to CRC Error Detection Algorithms, 3rd Edition" by Ross Williams.[3] There are a few holes that I've hoped for many years that Ross would fill with a fourth edition, but all in all it's the best coverage of a complex topic that I've seen. Many thanks to Ross for sharing his expertise with others and making several of my networking projects and this column possible.

This three-part discussion and the required C programming have been so much fun for me that I'm going to continue with such down-and-dirty subjects for a while longer. Next month's column will describe a virtual serial port concept that I invented a few years back to enable the use of off-the-shelf software such as SLIP/PPP, remote debuggers, and stdio over an ordinary PCI bus. Until then, stay connected... **esp**

Michael Barr is the technical editor of Embedded Systems Programming. *He holds BS and MS degrees in electrical engineering from the University of Maryland. Prior to joining the magazine, Michael spent half a decade developing 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@mfi.com.*

References

1. I first modified both functions to use unsigned char instead of int for variables nBytes and byte. This effectively caps the message size at 256 bytes, but I thought that was probably a pretty typical compromise for use on an eight-bit processor. I also had the compiler optimize the resulting code for speed, at its highest setting. I then looked at the actual assembly code produced by the compiler and counted the instructions inside the outer `for` loop in both cases. In this experiment, I was specifically targeting the PIC16C67 variant of the processor, using IAR Embedded Workbench 2.30D (PICmicro engine 1.21A).
2. Barr, Michael, "Leveraging the 'Net," *Embedded Systems Programming*, November 1999, p. 45.
3. This 1993 paper can be found at *ftp://ftp.rocksoft.com/clients/rocksoft/papers/crc_v3.txt.*

www.diabsds.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

THOMAS HERBERT

Embedding TCP/IP

Last month, the author introduced you to the basics of TCP/IP. In this second installment, he discusses the details of putting TCP/IP into a resource-constrained embedded system.

Embedded systems have inherited the programming practices used in larger systems. Network protocols, and TCP/IP in particular, incorporate programming practices used in larger systems. As discussed in the first part of this article, the history of TCP/IP is one of adapting and modifying the original sources written at the University of California at Berkeley to embedded systems. The Berkeley stack is the basis for most of these ports and is the basis of most of the commercial TCP/IP stacks for embedded systems. Of course, real-time and embedded systems face many issues that are unique. A straight port of the Berkeley stack is not the best implementation for the particular needs of an embedded and real-time system.

Most vendors have modified the Berkeley code over the years to improve the performance of the stack in embedded systems. Any ports or modifications of the original Berkeley sources should address the following issues. This rule applies to commercial canned stacks, as well as home-grown porting jobs. Certainly, if you're purchasing a TCP/IP stack, you will want to verify that your vendor has taken the following things into account:

Buffer management. The TCP/IP `mbuf` buffer management should be able to use pre-allocated buffers rather than allocating them from the global heap at run time (via `malloc`).

Timers. The timers used in the protocols for connection management, timeouts, and retries should be managed by the RTOS. They should not be a separate implementation that will secretly steal bandwidth from the CPU or cause concurrency problems.

Latency. If an RTOS is present, it should not add any additional latency. Interrupt-handling interfaces should be fast and deterministic. The RTOS should not add any latency to the interrupt processing required with the physical transmission and reception of a frame. The large amount of context switches and CPU processing required in dealing with a packet increases the importance of using an OS with minimal thread context switch time.

Concurrency. All buffering mechanisms should have semaphore protection to allow higher performance potential in real-time systems. The first TCP/IP protocol implementations were on Unix systems and depended on manipulating hardware interrupt levels to eliminate resource contention problems. Semaphore protection should be available to the timers to reduce concurrency problems.

Minimized data copying. The TCP/IP implementation should minimize the amount of data copying. The data within each frame can be maintained in the same buffer so it doesn't need to be copied and re-copied by the CPU at each stage of the protocol. The networking chip's DMA places the packets directly in the managed buffer pool where the packet is passed up through the stack by manipulating pointers and not by copying data. Also, some vendors have extended

TABLE 1 The `ifnet` structure used in BSD 4.3

char	*If_name	Two character interface name	Defines interface
struct ifnet	*if_next	Pointer to next ifnet	All of these are in a linked list
struct ifaddr	*if_addrlist	Pointer to address for interface	These are also in a linked list
int	if_pcount	Number of promiscuous listeners	Used for packet filtering
caddr_t	if_bpf		Used for packet filtering
u_short	if_index		Defines interface
short	if_unit		Defines particular interface instance
short	if_timer		Time until if_watchdog is called
short	if_flags		Flags keep track of state of interface such as up or down
struct if_data	if_data	Volatile statistics	Used to keep statistics about interface
Int	(*if_init)	Initialization function for this interface	
Int	(*if_output)	Output function	This function is to queue packets for sending
int	(*if_start)	Start function	This function is to initiate sending of packets
int	(*if_ioctl)	IO command function	This function implements IO commands specific to this interface
int	(*if_reset)	Device reset function	
void	(*if_watchdog)	Timer function	Checks to see that interface has not timed out

the `mbuf` mechanism to allow the data to be shared between `mbuf`s and `mblocks` where there are STREAMS protocols also present in the system.

Link layer multiplexing. Protocol implementation requires a framework with mechanisms for queueing and buffer management. Also, modern protocols require more flexible device driver interfaces and more flexible multiplexing. This is particularly true where serial point-to-point protocols such as PPP are now extended to support IP tunneling and Virtual Private Networks (VPN). The original Berkeley implementation isn't sufficiently flexible to meet all of these needs. The better protocol stack implementations use a framework that allows the stack to be extended as new protocols and interfaces are developed. This can be accomplished by extending the basic Berkeley driver interface scheme, or the protocols can be rewritten to use a different framework

CPU bandwidth. Each embedded system application has different requirements for its TCP/IP stack. For example, a TCP/IP stack in most Internet appliances probably would not be considered real time. Also, if the network is used for control and management functions, the hard bandwidth requirements will be fairly low. On the other hand, if the application is streaming video or voice, the faster packet rates would qualify the application as a real-time application.

Link layer interfaces and device drivers

As I've described, the OSI model shows an interface between the physical and data link layers. In actual implementation, this interface is implemented in several ways.

BSD 4.3. Most of the examples in this article show the BSD 4.3 type of structures and interfaces. The traditional Berkeley stack could multiplex between multiple interfaces if they were using a common IP stack. Originally, it could only interface cleanly with link layers that were compatible with Ethernet and only among IP and its related protocols, such as ARP and RARP. Subsequently some implementors have hacked the BSD code to allow it to be used with serial interfaces such as SLIP (Serial Line Interface Protocol) or PPP. This was generally a somewhat kludgy way to make them look like Ethernet and as such, was not particularly efficient. Also, each vendor of a TCP/IP stack for embedded systems has put its own nuances in the interface mechanism as well.

The BSD 4.3-compatible stacks uses the `ifattach()` function and the `ifnet` structure. This structure and its associated attachment mechanism were inherited by most ports of the Berkeley stack used in embedded systems. Typically, the network device driver initialization function is specific to a particular OS. When the device specific initialization function is called, it first allocates space for its own internal data structures, often called a "softc" structure. This data structure generally includes space for the `ifnet` structure. It determines its own MAC address by reading it from the hardware. Then it fills in the fields in the `ifnet` structure. It sets the device-specific information such as the MTU, the MAC address, and the `if_name` and `if_unit`. It then fills in the function pointer fields with pointers to the driver's interface functions. Once this structure is appropriately initialized, the initialization code calls the `if_attach()` function with a pointer to the `ifnet` structure as an argument.

The `ifnet` structure is illustrated in Table 1. The `ifnet` structure contains, among other things, a pointer to a list of address structures for each interface. This address structure, called `ifaddr`, contains the interface's MAC address and the broadcast address.

BSD 4.4. BSD 4.4 extends the traditional interface in a few significant ways. Inside the stack, BSD 4.3 had hardcoded address lengths in its `sockaddr` structure. MAC addresses are also stored in the sockaddr and similar structures, and in BSD 4.4, these were extended to support variable length addresses to allow for PHYs other than

Ethernet. Also, the device driver interface was generalized to make it less specific to Ethernet. In addition, a pointer to the interface, `netif`, is passed along by the link layer when incoming data is queued to IP or another protocol in the network layer.

Data link provider interface. The data link provider interface (DLPI) is found in most implementations of STREAMS. The DLPI interface is not specific to TCP/IP and can be generalized for almost any protocol. It does, however, require that the protocols be implemented as STREAMS modules and the link layer be implemented as a STREAMS driver. It also requires 802.2-type framing to properly multiplex between the link layer and the network layer of the protocols bound to the link layer.

The legacy BSD interface between the link layer and the network layer doesn't incorporate any mechanism for interfacing specifically to connection-oriented or connectionless link layers. On the other hand, DLPI provides support for both connectionless or connection-oriented link layer binding. The connection-oriented service would be available if the Service Access

Promiscuity

Most network interfaces can be set to *promiscuous* mode so that all packets are received. For example, Ethernet interfaces can be set to receive all packets on the wire, including broadcast and multicast packets, as well as those that have someone else's MAC address in the destination field. It is interesting to note that most protocols would work even if the interfaces were accidentally set for promiscuous mode. Packet filtering then takes place at higher levels of the stack, which is very inefficient but probably functional. Some software above the link layer may want to use promiscuous mode by telling the interface to receive everything and send it all up. An example of this is a software protocol analyzer such as the `tcpdump` or `snoop` utility found in Unix systems.

At the transport layer, the API provides a promiscuous socket. Generally, you can't stop processing and sending other information merely because the user wants to observe the network traffic. To provide the capability of receiving promiscuously while preserving other network functionality, the data link interface must support some sort of promiscuous binding. In other words, it has to provide a mechanism for the layer above to request all packets.

FIGURE 1 Data link LLC type encapsulation

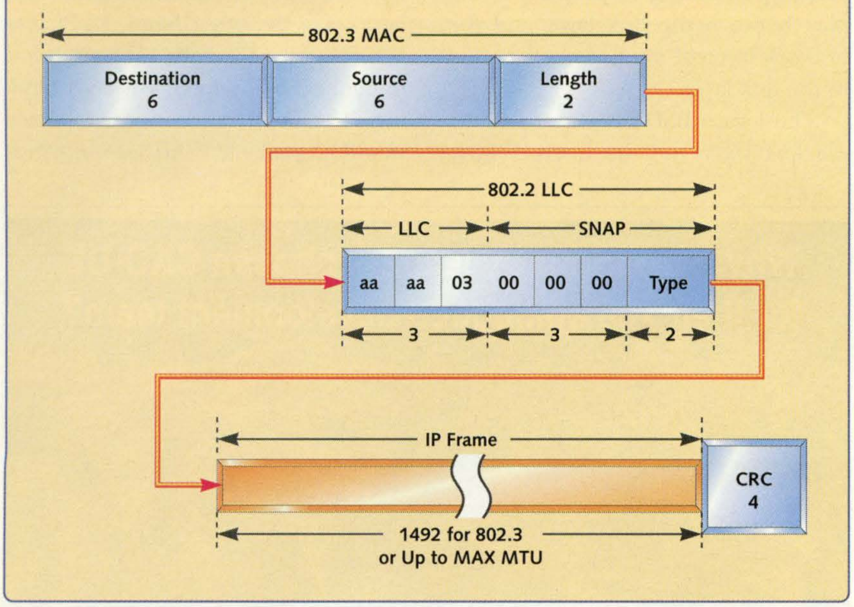
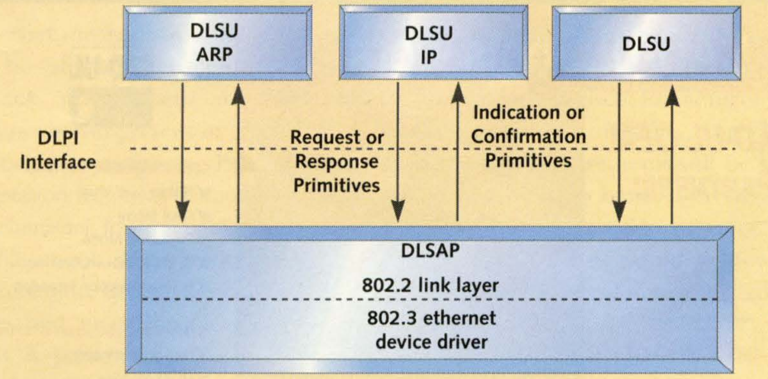


FIGURE 2 Data link provider interface


DLSU ARP
DLSU IP
DLSU
DLPI Interface
Request or Response Primitives
Indication or Confirmation Primitives
DLSAP
802.2 link layer
802.3 ethernet device driver

Provider (SAP) is capable of supporting connection-oriented transmission. Also, DLPI allows the data link SAP (DLSAP) to identify itself as a promiscuous SAP, that is, one that can grab all the packets on the net, not just those directed to it. See the sidebar called "Promiscuity" for more information.

Fundamental to DLPI is the concept of the SAP. In Figure 1 you can see how the SAP identifiers are incorporated in the LLC framing. The network layer above the interface is a data link service user (DLSU) and the link layer driver is a DLSAP. DLPI uses a set of request primitives passed as messages from the DLSU to the DLSAP. In response, the DLSAP passes a set of acknowledgment primitives back to the DLSU.

Figure 2 shows the relationship between the SAP and the DLSU. DLPI manages the state of the relationship between the DLSU and the DLSAP with a state machine. Table 2 lists some of the common primitives likely to be used for TCP/IP connectionless link layer service and their responses.

Extended multiplexing interface. Extended multiplexing interfaces have been developed by some commercial RTOS vendors to extend the BSD 4.4 interface to include link layer multiplexing and multiprotocol capability. This allows some of the capabilities found in the STREAMS DLPI interface to be available to TCP/IP-based networking. A good implementation will do this while still maintaining compatibility with the traditional Berkeley-type TCP/IP stacks and drivers.

Traditionally, various attempts have been made to allow drivers to support simultaneous interfaces to both Berkeley and STREAM-type stacks. This was done with a layer of glue code that would copy the messages from STREAS `mblocks` to Berkeley-type `mbufs`. A better mechanism should allow the actual data in the frame to be shared between the `mblock` and `mbuf` header structures.

A good extended multiplexing interface mechanism gives you a DLPI-type binding mechanism for binding a stack to an arbitrary link layer interface. It should do this while providing as much backward compatibility as possible with Berkeley-style implementations. As in DLPI, this advanced mechanism should allow multiple stacks to be bound to the same link level interface. It provides a generic mechanism for address resolution between MAC addresses and protocol addresses. This address resolution capability should still support the BSD ARP protocol previously discussed under the section about BSD 4.3. As with DLPI, the advanced interface has a state machine to keep track of the relationship between the stack and the link layer. It also provides a means of enabling multicast addresses on an interface. This type of mechanism was absent from BSD 4.3 and only partially present in BSD 4.4.

Hosting TCP/IP in your embedded system

Since the BSD stack has been available in source code form for many years, most people implementing TCP/IP for embedded systems have used it as a

TABLE 2 Primitives used in STREAMS DLPI

Message	Response	Purpose
DL_ATTACH_REQ	DL_BIND_ACK	Used to tell driver to initialize hardware and associate physical layer with a STREAM. Required for Style 2 providers to initialize themselves
DL_INFO_REQ	DL_INFO_REQ	Asks the provider to send information about the interface including MAC type, service mode, QOS (quality of service)
DL_BIND_REQ	DL_BIND_ACK	Requests DLP to bind an SAP to a STREAM. Used to indicate the type of service, connection oriented or connectionless SAP
DL_ENABLEMULTI_REQ	DL_OK_ACK	Requests the provider to enable a muticast address for the specific DLSAP
DL_PROMISCON_REQ	DL_OK_ACK	Asks provider to enable promiscuous mode for this SAP
DL_PROMISCOFF_REQ	DL_OK_ACK	Asks provider to disable promiscuous mode for this SAP
DL_UNITDATA_REQ	DL_UNITDATA_IND	Requests connectionless datagrams be passed to the DLS user from the SAP. The datagrams are passed in a message block preceded by DL_UNITDATA_IND primitive

base. Hardly anyone implements his or her own protocol suite from scratch. You'll have a number of fundamental choices if you want to put TCP/IP in a product for the first time, whether you want to use a commercial of TCP/IP or do your own port.

You may want to look at a number of factors before you decide which path to take for incorporating TCP/IP. You'll want to ask yourself basic questions about the connectivity requirements in your design. For example, how sensitive is your project to unit manufacturing cost? Ask about the future of your design. What is its reuse potential? Is the project to be a platform for launching future projects? Following are a few broad categories of products used in connected embedded systems. Each of these categories suggests a different direction for your implementation of TCP/IP in your embedded design.

1. The embedded product is based on a legacy system. There is only a limited need for remote access. So far, remote access has only been available through a serial port. TCP/IP allows access to this serial port. Potential increase in manufacturing cost is not the determining factor when adding TCP/IP
2. The product should be relatively easy to maintain. A network connection is needed for remote management of the product
3. The networking connection is required for data capture and analysis
4. The product is a router, gateway, switch, broadband modem, or a similar product in which networking is a fundamental element
5. The product is a consumer device with a graphical display, such as a personal digital assistant. The ability to browse the web is a fundamental part
6. The device is associated with a measurement and control system. An easy method must exist for reaching the device from a PC with a web browser. An embedded web server is a fundamental requirement

Below I list some of the choices you have today for incorporating TCP/IP in your product.

Total hardware implementation
A number of companies are developing completely self-contained cards in a small form factor that add network connectivity. The stack is implemented as part of an opaque system and is generally packaged as a PCM card which can be interfaced to your system's serial port. Your system is made to think it is talking to a serial port. This solution may be best for a category 1.

Advantages

- Vendor owns all networking problems
- Easy to specify a working solution
- Very little product redesign required for legacy products
- You can concentrate on your product's added value

Disadvantages

- Not a flexible solution
- Can't be configured to give remote users an elegant interface
- Lack of networking integrated into your product can make the solution seem clunky
- Higher added manufacturing cost

Roll your own stack with no RTOS
Using no RTOS involves snarfing a free (public domain) or source-licensable TCP/IP stack and porting it straight to the hardware without the benefit of a commercial RTOS. At minimum, you will need to implement a basic scheduler, a timer mechanism, and a buffer allocation mechanism. This solution may be appropriate for categories 1 and 2, but only if the requirement domain is well defined and there is little need for reuse.

Advantages

- Low manufacturing cost because of reduced royalties
- High engineering cost
- You own all the sources
- It might be fun to do

Disadvantages

- You own all the problems
- No RTOS to help allocate CPU bandwidth between the network stack and other parts of your application
- Inflexible and doesn't allow for future product growth
- Can't be easily brought up to date

World's Smallest 28-Lead OTP Microcontroller in Chip Scale Packaging (CSP)

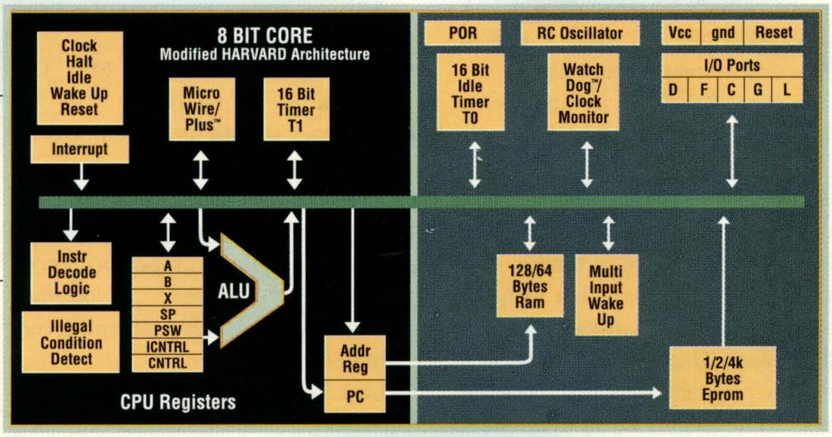
National's COP8SAC7SLB Combines Ease of Design with Low System Cost

- On-Chip Power-On Reset
- Four Clock Options with On-Chip RC Oscillator
- WATCHDOG™ and Clock Monitor
- Up to 12 High Current Outputs
- 16-Bit Timer
- Processor Independent PWM Mode
- MICROWIRE/PLUS™ Serial Interface
- Absolute Security Feature
- 4K One-Time-Programmable (OTP)

Ideal for Space Constrained Designs such as Portable Appliances, Wireless Designs, Cordless Phones, and Security Systems.

For More Information on COP8SAC7SLB:
www.national.com/COP8
1-800-272-9959

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



National Semiconductor®

with new networking standards

- Compliance and interoperability testing is an enormous burden
- Upgrading is difficult because of incompatibility with other implementations
- High engineering cost
- Bad for time-to-market-constrained projects

Third-party integration

With this method, you would purchase an RTOS from one vendor and a TCP/IP stack from another.

Advantages

- You can use any RTOS, even one that is fairly low in cost and/or has no royalties

Disadvantages

- It is up to the you to select and integrate your own TCP/IP stack
- You may have to design and implement your own transport and link layer APIs
- Not time-to-market friendly
- You own all the interface problems
- Little or no support available

All in one

With this option, you would purchase a bundled product from a vendor who has a complete solution that includes an RTOS, development tools, and a TCP/IP stack. This solution can be used for all the product categories above. Of course, with category 1 you would have to incur some redesigning to incorporate the new RTOS into your design.

Advantages

- Best for design cycles with critical time-to-market constraints
- Scaleable to meet evolving needs
- Should give you access to added value upper-layer protocols, such as embedded web servers and clients, network management, and other applications

Disadvantages

- May add some royalty cost to your product
- Some redesign of legacy products may be required to add the networking feature
- RTOS may be more than you need

Selection criteria for commercial RTOS and TCP/IP products

The following is a list of items you may want to consider when selecting an RTOS and network stack vendor:

- Choose a vendor who offers a variety of interfaces and available device drivers. It should be able to smoothly support interfaces other than LAN PHYs, such as Ethernet
- The vendor should have stable and tested support for your target CPU and networking interfaces
- If you're building a platform to be used as a base for a variety of products, make sure that your vendor can provide you with an upgrade path as your requirements grow and change
- The TCP/IP should be BSD 4.4-compatible (not 4.3)
- The stack should have the hooks to allow it to be managed remotely via SNMP or RMON
- Zero copy buffers should be an available option for performance. This is accomplished in conjunction with some kind of enhanced buffer management
- Link layer multiplexing should be available to allow the TCP/IP stack to co-exist with other protocols, as well as support multiple interfaces

An established solution

As a technology, TCP/IP is close to 20 years old. It was developed as part of one of the first attempts to establish an internetwork. The protocol suite has outlasted more recent LAN and WAN networking protocols, and by now many of these other protocols have been dropped or forgotten. TCP/IP and the concept of the Internet is elegant in its simplicity. Largely because of its ubiquity and its simplicity, this old TCP/IP-based Internet has become the global fabric uniting much of modern life and is now the basis for much of the hot new "high technology" of today.

Obviously, I shouldn't try to dictate the choices you make as an embedded engineer when you move your legacy design to the Internet-enabled world or as you develop your brand new Internet appliance. As an embedded engineer, you will follow your own dictates given the business and technical constraints particular to your project. As I discussed above, there are many choices for selecting and integrating TCP/IP in your design. Many of the new ASICS designed today have extra gate capacity as well as extra bandwidth in the CPU core, and often memory is not as constrained as in previous designs. After making the right choice for your design, you may find that in today's world most designs have enough headroom to incorporate Internet ability. Even on many small embedded systems, adding TCP/IP probably won't be a large draw on your system for CPU and memory. **esp**

Thomas Herbert is a field consulting engineer with Wind River Systems. Before that, he worked in embedded systems in various applications for many years for such companies as Xerox Corp., Eastman Kodak, and as an independent consultant. Tom has contributed several articles to Embedded Systems Programming *over the past few years and holds a patent in pattern recognition. He can be reached at tom.herbert@windriver.com*

Further Reading

Many excellent books have been written about TCP/IP internals, application programming using TCP/IP, and maintenance and configuration of IP networks. I would like to suggest a few references below which I have found valuable for explaining the internals of TCP/IP that may be useful to the embedded engineers who want to bone up on networking.

The first and most important references for

TCP/IP are the following two volumes. If you really need both to see how the protocols work and how they are implemented, these books are must-haves:

Stevens, W. Richard. *TCP/IP Illustrated, Vol. 1, The Protocols*. Reading, MA: Addison-Wesley, 1994.

Wright, Gary R. and W. Richard Stevens, *TCP/IP Illustrated, Vol. 2, The Implementation*. Reading, MA: Addison-Wesley, 1995.

For an authoritative work on BSD 4.4, I suggest the following book:

McKusic, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4BSD Operating System*. Reading, MA: Addison-Wesley, 1996.

For general information on STREAMS:

Herbert, Thomas F., "Implementing Network Protocols and Drivers with STREAMS," *Embedded Systems Programming*, April 1997, p. 28.

Or you may want to read the definitive reference on SVR4 STREAMS. Even if you aren't specifically interested in STREAMS, chapter 11 on DLPI (Data Link Provider Interface) is a good look at what functionality needs to be in a good multiplexing data link interface. Unfortunately, at the time of the writing of this article, this book is out of print.

Unix Press. *STREAMS Modules and Drivers*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

STREAMS

STREAMS is a generic framework and API developed at AT&T Bell Labs for implementation of layered networking protocols. Several commercially available implementations of TCP/IP use STREAMS and there are several implementations of the STREAMS framework available for embedded markets. STREAMS is well suited for systems that have to multiplex packets to and from multiple protocol stacks. Since the whole world is standardizing on the Internet, less interest exists in supporting protocols other than TCP/IP. Therefore, most embedded TCP/IP implementations are based on Berkeley. In this article, I concentrate on the Berkeley-based TCP/IP implementation, but in the last section, I've included some references about STREAMS.

Being Royalty Free Is not Enough.

You work under a lot of pressure to meet production demands and schedules. So the RTOS you choose has to have more than one outstanding quality to make you feel comfortable.

ThreadX, the leading royalty-free, source-code RTOS gives you more, much more...Like state-of-the-art technology with preemption-threshold and a picokernel design that automatically scales to your needs. Processor support for most processor families. Documentation so clear and concise that ThreadX's User Guide constantly receives kudos from our customers. And such a wide array of tools that chances are there is ThreadX-aware debugging already built-into your favorite tool chain.

Call **888.THREADX** or visit **www.threadx.com** and find out more about ThreadX, the RTOS with all the features to make you comfortable.

Processors Supported

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

the comfortable RTOS

eL
Express Logic, Inc.

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

JOHN MEADOWS

An Introduction to the JetSend Protocol

JetSend is a media-independent communications protocol that can exchange information in its proper context without any product-specific knowledge or device drivers. This article provides an overview of the JetSend protocol.

JetSend is a media-independent communications protocol, developed by Hewlett-Packard, that provides interoperability among a wide variety of devices in differing vertical markets. You may be thinking, "Do we really need another communications protocol?" The answer is yes, and here is the reason: a need exists for a protocol that supports device-to-device content exchange. True, protocols already exist for point-to-point data exchange (FTP, IrTranP, and so on), but the problem with these protocols is that they don't fully describe the content being exchanged, and as a result the data transferred is both machine dependent and OS dependent.

The fact that JetSend is able to exchange information in its proper context, without any product-specific knowledge or device drivers, gives it tremendous value. Increasingly, embedded devices are required to exchange data with other devices directly (that is, no PC will be required). With JetSend, the user of a device doesn't need to know what protocol is being used to transfer data. These devices might be digital cameras, cell phones, PDAs, printers, scanners, or set-top boxes, to name just a few. Another compelling reason for using JetSend is that it already has support in many of the devices just listed. All of Hewlett-Packard's standalone printers and scanners support JetSend. Also, JetSend printer support for Windows CE PDAs—although not native—can be obtained via download from HP. This article will attempt to provide an overview of the JetSend protocol.

Overview of JetSend

The HP JetSend protocol is a peer-to-peer communication protocol. The protocol allows two devices to connect, negotiate data types, and exchange informa-

FIGURE 1 Two-page document with images and text

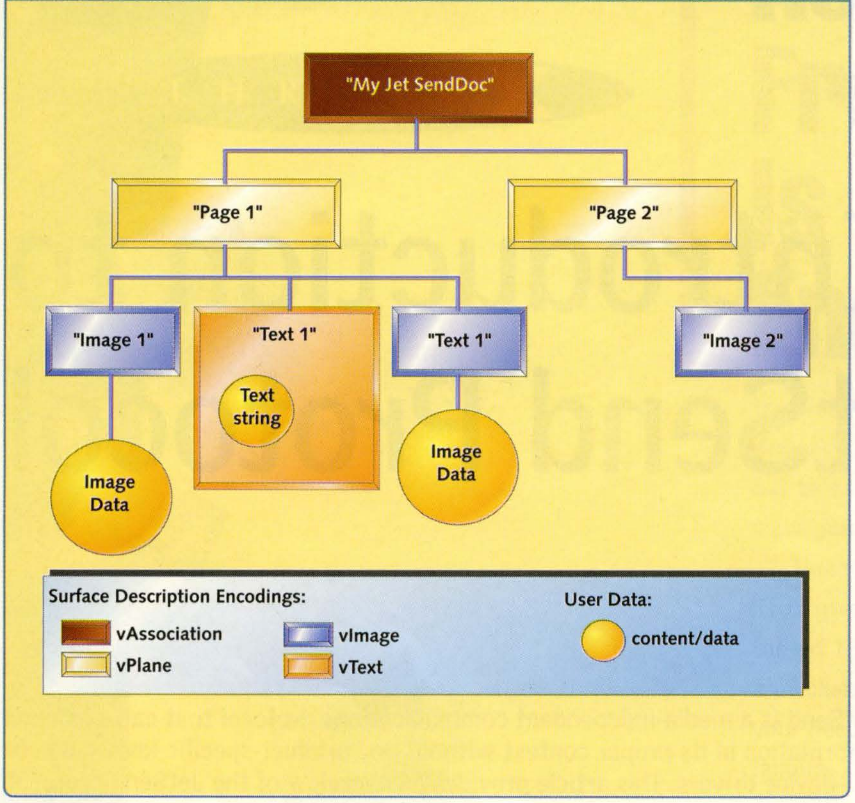
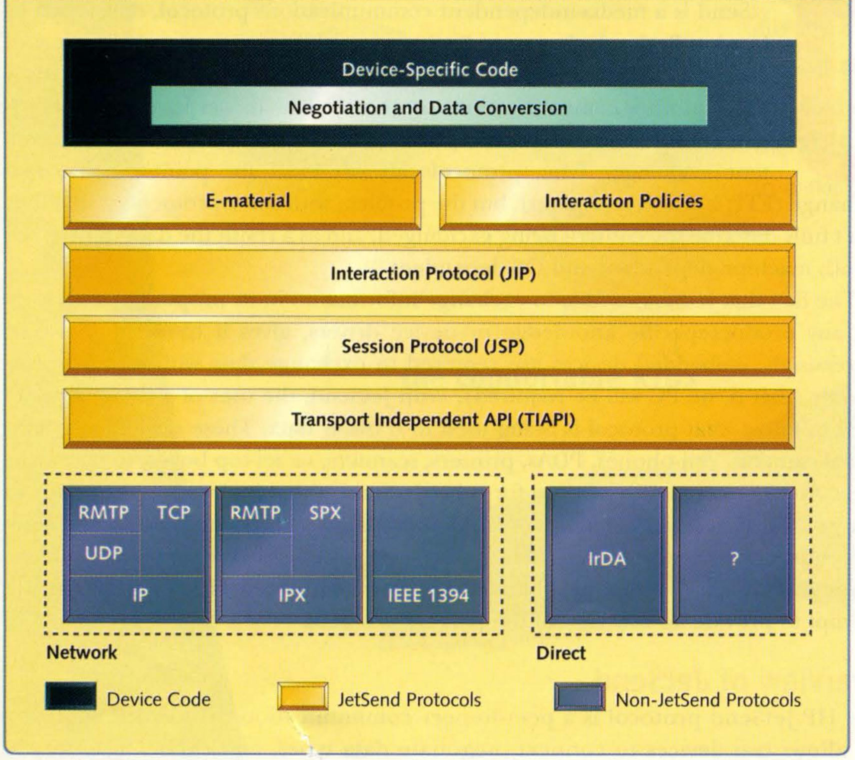


FIGURE 2 The JetSend protocol stack



tion. With this new protocol comes a new paradigm for job control in embedded devices. For example, say a person wants to print a picture from a digital camera. The current job control model requires the user to transfer the picture to a printer through a two-step process. First the user must launch a compatible digital camera application on a PC to receive the desired pictures from the camera. Then a host application—possibly the same one—prints the desired pictures using a compatible printer driver. In this job model, the host computer acts as a translator between two devices that speak different languages.

The JetSend job control model differs in that it is a one-step, peer-to-peer communication process. A host PC no longer controls the exchange of data between two devices, which removes the need to load a special driver on the sending device because both devices speak a common language. One side effect, however, is that the host PC can no longer be relied upon for any data translation.

Within JetSend, all data exchanged between devices is accomplished using surfaces. Each surface object has a name, a description, and content. The description is analogous to a file header, and contains information about the type and content of the surface. The content portion may be null, data, or a reference to another surface (a child surface). The data contained in a surface is called electronic material (e-material).

As an example, a two-page document, one with image and text and the second with just an image, can be represented by six surfaces, as shown in Figure 1. The encodings of the surfaces are also shown. In a printing example, the *vAssociation* encoding describes the entire document or print job. The *vPlane* encoding controls the layout of pages to be printed. The *vImage* and *vText* encoding describe the content of the image and text.

Accelerated Technology
INCORPORATED

Accelerated Technology allows you the freedom to choose only the embedded software products that your project requires. Nucleus software, implemented in thousands of products worldwide, is scalable to accommodate any embedded application ranging from the smallest to the largest design.

Nucleus embedded software provides a robust solution with full source code and without charging royalty fees. Accelerated Technology's Nucleus products are enthusiastically pursued and increasingly supported by leading Third Party CPU and tool partners throughout the embedded market.

Accelerated Technology provides the leading source code, no royalties real-time operating system software with the same, full functionality of higher end companies. ATI offers cutting-edge embedded products, developed in-house, for seamless and thorough integration.

Accelerated Technology has recently been recognized by Deloit & Touche, Inc. Magazine, and Software Magazine as one of the fastest growing technology companies in the embedded industry with technology implemented in applications ranging from wireless, networking, industrial control, to multimedia and more.

NO ROYALTIES....
IT JUST
MAKES
SENSE
ANY CPU.......

NUCLEUS
Embedded Software
www.atinucleus.com
1.800.468.6853

In this example the "Text1" child surface on the first page of the document is a text description of the "Image1" picture. This description might be something like "Billy's first birthday—January 01, 1999." A raster image of this text string is also offered in case the receiving device does not support the *vText* encoding. In other nonprinting applications, surface encodings may take on different meanings.

LISTING 1 Activity Manager API routines

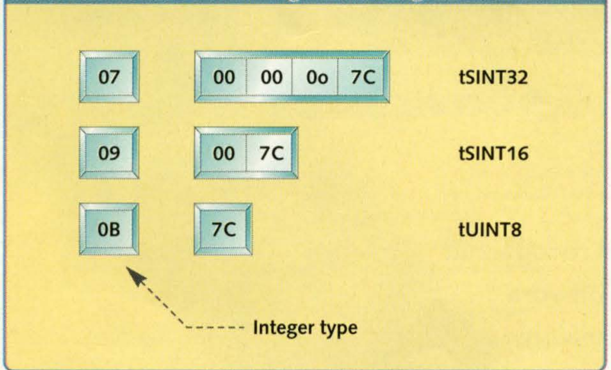
```
/* Initialization */
Tstatus jamInitialize();
Tstatus jamShutdown();

/* Buffer Management*/
BD* jamCreateBuf(Tu32 iSize);

/* Event Handling*/
Tstatus jamGetNextEvent(JAMEVENT **pEvent);
Tstatus jamFreeEvent(JAMEVENT *pEvent);

/* Session Management*/
Tstatus jamStartSession(Tchar *pAddress);
Tstatus jamEndSession(void);
Tstatus jamListenForSession(Tchar *pDomain, Tbool
bAccept);
```

FIGURE 3 JetSend integer encodings



Surface interaction model: the rules of the language

JetSend is a layered protocol designed to be machine independent as well as transport independent. The components of the protocol shown in Figure 2 are as follows:

- The device-specific code is the application itself
- The Interaction Policies control the method of interaction between devices by enforcing certain types of policies. One of the policies—the Job Policy—will be covered in detail in the section on using the JetSend API
- The e-material routines allow the device-specific code to format data to be communicated in a standard JetSend format that another JetSend enabled device will be able to understand. Surfaces are e-material
- The JetSend Interaction Protocol (JIP) manages the sending and receiving of surfaces
- The JetSend Session Protocol (JSP) is responsible for managing a reliable data connection with a remote device
- Transport Independent (TI) layer decouples the JetSend stack from the OS-dependent network layer

Data sent by the application layer (device-specific code) must progress down through each layer before being sent out over a transport medium and then back up through the JetSend stack of the receiving device.

Machine independence is achieved through the use of e-material, which specifies data types and formats that do not rely on software language data types. An example of this would be the e-material definition of a simple integer type. The first byte of the type definition is a value type identifier, which tells what type of integer it is, as shown in Figure 3. The byte(s) that follows is the actual value of the integer in network byte order (big endian—most significant byte first).

The TI layer is responsible for providing a generic API to the non-JetSend transport protocols. Operating system-specific code is needed to translate generic TI API messages into OS network driver calls and vise versa.

Using the JetSend API

To communicate using JetSend, device-specific code must interact with an API composed of three main sections: the Activity Manager, Interaction Policies, and e-material routines.

Activity manager API. The Activity Manager is the heart of the JetSend API and fulfills two main functions. First, the Activity Manager is responsible for managing JetSend sessions. A session is a reliable full duplex communication channel established between two JetSend-enabled devices. All user data transferred from one device to another is conveyed over one or more sessions. The second main function is event handling. The Activity Manager manages events coming from the lower layers of the JetSend stack by providing an event queue and polling mechanism to process and free events. Listing 1 shows the routines that make up Activity Manager API. More about how the Activity Manager works will be discussed in the section on sending application data.

JetSend interaction policies. The second main section of the JetSend API is concerned with supporting one or more of the JetSend Interaction Policies. This component of the JetSend Protocol defines various typical sequences of interactions that devices agree to follow. Most of the toolkits available implement the session policy and a portion of the job policy. Altogether five policies have been defined as follows:

- *Session policy*—how to establish and disconnect sessions between devices
- *Job policy*—how to send documents between senders and receivers
- *Self policy*—how to exchange global information about a device, such as label, icon, and passwords
- *Status policy*—how to give status about a device and about jobs
- *Address policy*—how to program devices with new destination addresses

Of the five policies, the session policy is supported directly by the Activity Manager through the use of session management routines. The session policy defines how sessions are created and destroyed. In the session policy, roles for both passive and active devices are defined. The passive device acts as a consumer and the active device acts as a producer. In order for two devices to communicate, one device must begin listening for another device to connect to it on a given transport channel. The listening device opens a passive session and waits for notification of a connecting device.

The remaining policies—job, self, status, and address—define how data is formatted and exchanged between devices.

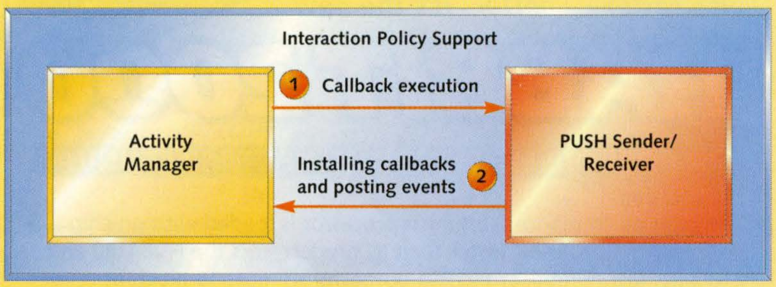
The job policy is used to send and receive application data over JetSend sessions. Two methods for exchanging data, the PUSH method and the PULL method, are defined by the job policy. As the name implies, the PUSH method is used to push data from a sending device to a receiving device, as would be done for printing, faxing, and so on. I will give an example of a PUSH sender/receiver in the section on sending application data. The PULL method is used when the receiving device needs to be able to select certain pieces of data from a sending device, as would be the case for some type of monitoring device. An example of this might be a JetSend-enabled digital Walkman that is able to download selected songs from a jukebox-like storage device using the PULL method.

The self policy, status policy, and address policy together provide the mechanisms to convey device attribute

The status policy, as the name implies, allows a device to obtain the status of another device. Using this policy it is possible to report conditions like paper jam, low battery, and so on.

and status information. The self policy provides the ability for a remote device to obtain device attribute information. This information includes the device's address, name (in text and in a Windows icon bitmap), and PIN number to allow support for user authentication. The self policy is often used in conjunction with the address policy.

FIGURE 4 The PUSH sender installs callbacks for events it will handle as part of the job policy


Interaction Policy Support
1 Callback execution
Activity Manager
Installing callbacks and posting events 2
PUSH Sender/ Receiver

The status policy, as the name implies, allows a device to obtain the status of another device. Using this policy it is possible to report conditions like paper jam, low battery, and so on.

The last policy in this grouping, the address policy, defines a mechanism to convey addressing information about additional devices available on a network. The JetSend protocol doesn't have a mechanism to automatically discover JetSend devices on a given network, so the address of at least one JetSend device must be known in order to use JetSend to accomplish a given task. Users program a device with the addresses of the devices they wish to use. For point-to-point transports like IrDA, this is as simple as programming a single default address into the device. On network transports like IP, the address policy can be used to program devices with additional addresses. The address policy allows a network administrator to set up a well-known device address on a network that holds the addresses of other JetSend devices. Once addresses are obtained, the type and capabilities of a device can be discovered by using the self policy.

E-material routines. The e-material routines are used to create and parse surfaces in order to intelligently exchange data. E-material data structures have predefined data elements and data types that include values, strings, and lists. Using e-material routines, simple data types like integers and strings are converted to e-material types to build surface descriptions. E-material surface descriptions are constructed to fully describe user data and list possible encodings in which the sending device can transfer the data. This gives the receiving device the ability to choose an encoding that best suits its devices' capabilities. The process of selecting a data transfer format is called *device negotiation.* To ensure that devices of the same class will be able to exchange data, a default data format is defined. For example, the default encoding for image devices is 300 x 300 dpi, monochrome, RLE compressed raster data. All JetSend-certified image devices (printers, scanners, copiers, cameras, whiteboards, and so on) must be able to send and/or receive this encoding (Section 2.2.3 of the JetSend Protocol Specification, v. 1.5). The ability to add new encodings while enforcing support for a default encoding will allow JetSend to maintain backwards compatibility. You have a device like a digital Walkman that supports audio clips in MP3 format. The sending device might support new audio formats while still supplying the old digital Walkman with MP3 audio clips. The new digital Walkman can thus take advantage of a new format, while the old devices—having only the old format—are still supported.

Sending data

The sample PUSH sender given in Listing 2 at *www.embedded.com/code.htm* will be used to describe the steps required to transfer user data from one device to another (see also Figure 5). At the beginning of the routine `RunSimpleSender()` the Activity Manager, which is responsible for managing sessions, is initialized by calling `jamInitialize()`. This sets up an event queue inside the Activity Manager that can be polled by the device-specific code using the function `jamGetNextEvent()`. This polling mechanism allows the device-specific code to process events that have propagated up through the JetSend stack. Next, the PUSH sender is initialized by calling `psInitialize()`. Upon initialization the PUSH sender installs callbacks for events it will handle as part of the job policy (shown graphically in Figure 4). To communicate from one JetSend-enabled device to another, an active JetSend session must be created, which is accomplished by calling jamStartSession(). The transport address is specified in this call. In this example IrDA is selected as the transport. For a session to succeed in opening, a JetSend receiver must be listening for a session on the same network channel selected in the jamStartSession() call.

When a session is first established (that is, the session opens successfully) the PUSH receiver constructs an "IN" surface and sends it over the newly opened session. The purpose of the IN surface is to inform the connecting device that a receiver is available which supports the job policy. Upon receiving the remote device's IN surface, the PUSH sender posts a psInArrivedEvent event to the Activity Manager. The sending device-specific code then reads this event via jamGetNextEvent() and begins the process of sending a document.

To illustrate how a document is sent, let's once again use the document in Figure 2. The *vAssociation* surface is constructed using routines from the e-material library and then sent out over the open session to the remote device. The PUSH receiver, called by the Activity Manager via a callback routine, posts a prJobStartEvent when the *vAssociation* surface is received. The remote device code, in processing this event, parses the *vAssociation* surface description and determines what portion of the data it wishes to receive.

For this example we will assume that the receiving device will ask that the entire document be sent through a series of requests to the sending device. First the receiver requests the first *vPlane* surface (or page) in our document. The sender receives this request as a psSurfaceRequestEvent from the jamGetNextEvent() routine. A description of the data contained in the requested *vPlane* must then be constructed via e-material routines and sent to the remote device. The *vPlane* description defines how a page is laid out. In our sample document this description would contain (*x,y*) coordinates of both the image and text objects. The receiving device then knows where to place the objects for display or printing. At this point the receiver has the knowledge that the current job consists of a two-page document, and that the first page has both an image object and a text object on it. So far, no user data (content) has been transferred.

In the next step, the sending device requests the *vImage* surface which contains a description of the image data. This description is in the form of a list of possible encodings supported by the sender. The

New e-material encodings such as *vCard* and *vCalendar* are currently being added to the specification to enable the exchange of contact and calendar information.

receiving device reads the list of data encodings and requests that the content be sent in a format that best suits its device capabilities. The sender—upon receiving a request—sends content messages until the entire body of the *vImage* is sent. The remote device then requests the *vText* surface be sent. Our example document has the text content as part of the *vText* object on page one, so once the *vText* surface is sent, the first page is fully transferred and can be displayed, printed, or stored. Since there are two *vPlanes* in the *vAssociation* surface of our example document, the remaining *vPlane* will be requested using the same techniques.

When all the data is sent, the session can be closed or left open. A session would be left open if additional data were to be transferred. For example, the sending side could have additional documents (or jobs) that it desires to send. The additional documents would be sent in the same manner as I previously described. The possibility also exists that the receiver would want to get updates of any changes made to a received document by the sending device. An example of this might be a JetSend-enabled display connected to a portable computing device.

A few good reasons

The JetSend protocol is well suited for embedded devices for a couple of reasons. First, the protocol has been kept simple by defining everything in terms of surfaces. Therefore, a typical implementation of the JetSend stack on an embedded device will be somewhere around 60K of code. The second reason is that JetSend is peer to peer, and will allow devices to communicate with a range of other devices, a feature which will likely become a requirement as the number and usage of new devices increase. The third reason why JetSend is a good fit for embedded devices is the negotiation aspect of the protocol. Negotiation allows embedded devices to support a small minimum set of data formats that will be directly used by the device. For example an image class device does not have to support TIIF, JPEG, GIF, Windows bitmap, and so on, but only needs to be able to support raster image data in at least the minimum 300dpi RLE encoding. As long as memory is not free, the negotiation feature of JetSend will allow embedded devices to communicate with a wide variety of other devices at a reasonable cost.

So far I have covered only a few of the possible uses of JetSend. Hewlett-Packard continues to develop the protocol and has added JetSend support to many of their printers and scanners. New e-material encodings such as *vCard* and *vCalendar* are currently being added to the specification to enable the exchange of contact and calendar information. A new interaction policy, the control policy, has been developed to support remote control of JetSend-enabled devices. Also added to the specification is support for proxies. As an example, an e-mail proxy can be used to provide e-mail support for devices that do not support e-mail, but do support sending to proxies. To learn more about these new features, as well as for

TOMORROW'S TECHNOLOGY TODAY

WHY PACIFIC SOFTWORKS ?

Pacific Softworks is in a unique position to offer the embedded developer a complete suite of software components required to develop Information Appliance products. Today, Pacific Softworks is pleased to introduce *i*ApplianceNET.com, the world's first "Application Service Provider" (ASP) created exclusively for Internet Appliances. With Pacific Softworks expertise in Internet and Web protocols, its technical "know how" for hardware design and integration, and now, together with *i*ApplianceNET.com, we can work with our embedded developer customers to take their Internet Appliance as far as the imagination can go!

FUSION APPLIANCEWARE

The only complete suite of Internet and Application products from one supplier. All integrated for a fast "time-to-market" solution.

Visit our website for more in-depth information on our complete solution offering for your, Information Appliance. www.pacificsw.com, email: sales@pacificsw.com Pacific Softworks provides a single supplier solution for your Information Appliance for today and tomorrow.

INTRODUCING *i*APPLIANCENET.COM

Wondering how to deploy and connect your Internet Appliance product over the Internet? We have the answer. *i*ApplianceNET.com, the world's first "Application Service Provider" (ASP) is a private, dedicated network for connecting Internet Appliance thin clients with content and management servers. With *i*ApplianceNET.com you can monitor, manage, and control your deployed products anywhere in the world!

Web Protocols:	Micro Browser (100K), Micro Server (13K), E-Mail Client, Portable GUI (150K)
Internet Protocols:	TCP/IP, ARP, RARP, ICMP, IGMP, BOOTP, DHCP, PPP, Satellite TCP, Mobile IP*
Applications:	FTP, TFTP, TELNET, DNS, SMTP, POP3, IMAP, RPC, NFS
Management:	SNMP v1, v2, v3, MIB Compiler
Security:	IPSEC*, SSL
Routing:	RIP2, OSPF, BGP4, NAT Router
RTOS:	Thread-X, G-OS (based on iTRON 3.0)
Tools:	Compilers, Debuggers, Simulation
ASIC Design:	SOC Development (System on a Chip)

**Product in development*

Pacific Softworks
our wave... your Success.

California Corporate Headquarters:
tel: 805.499.7722 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

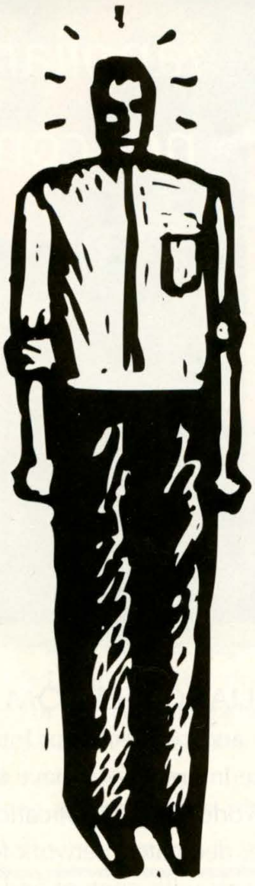
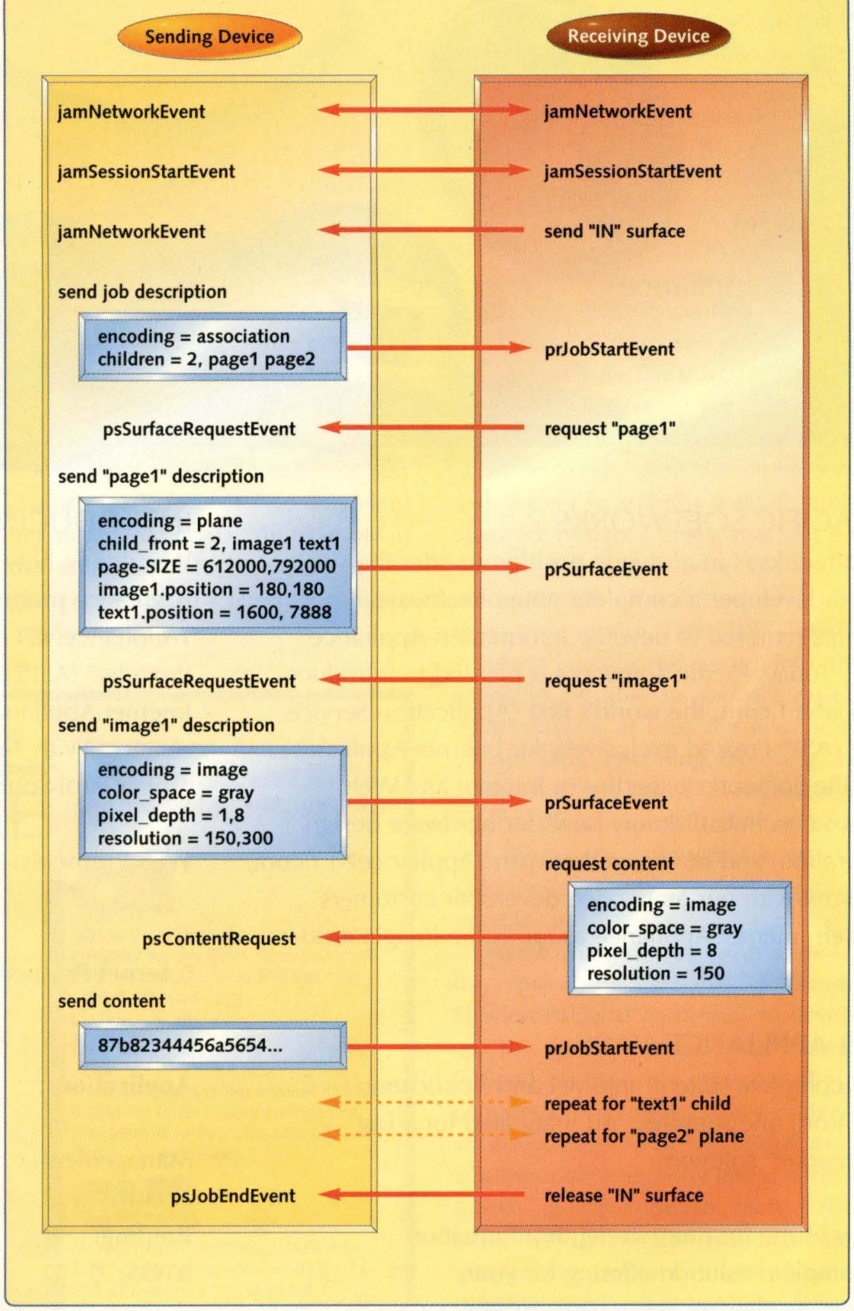


FIGURE 5 JetSend transaction



additional information on JetSend, visit *www.jetsend.com.* **esp**

John Meadows is a member of the technical staff in the E-Services Practice at Questra Corp. He has almost 14 years of experience in the industry, focused primarily on networking and communication protocols. Prior to joining Questra, he worked as a software specialist for Redcom Laboratories which designs and manufactures Telecom switching equipment for hostile environments. John has an ASEE from Alfred State College and a BSCS from the Rochester Institute of Technology.

RESUME

Name: HI-TECH C

Address: 7830 Ellis Rd., Suite 105, Melbourne, FL 32904

Phone: 800 735 5715

Fax: 407 722 2902

Email: hitech@htsoft.com

Web site: http://www.htsoft.com/

DOB: Jan 1st, 1984

Salary: Peanuts. Really. But a small signing bonus would be nice.

HIRED

* Interview ✓
* References ✓
* Qualifications ✓
* Call Tomorrow

Skills: Highly experienced in the translation of C programs into machine code for a wide variety of microntrollers. Can produce working code up to ten times faster than a human assembler programmer. Skilled in debugging complex programs.

Education: George Institute of Technology, Cornell University, University of California, State University of NY, University of East London, MIT, Sydney University, University of Waterloo, Worcester Polytechnic, Ecole Polytechnique, Fachhochschule Hannover and many others too numerous to mention here.

Employment: Currently employed at Microchip Inc, Cisco Systems, NASA, Federal Reserve Board, British Railways, Intel, Siemens, Motorola, Philips, Garmin, Hitachi and way too many more to list here. I like to spread myself around.

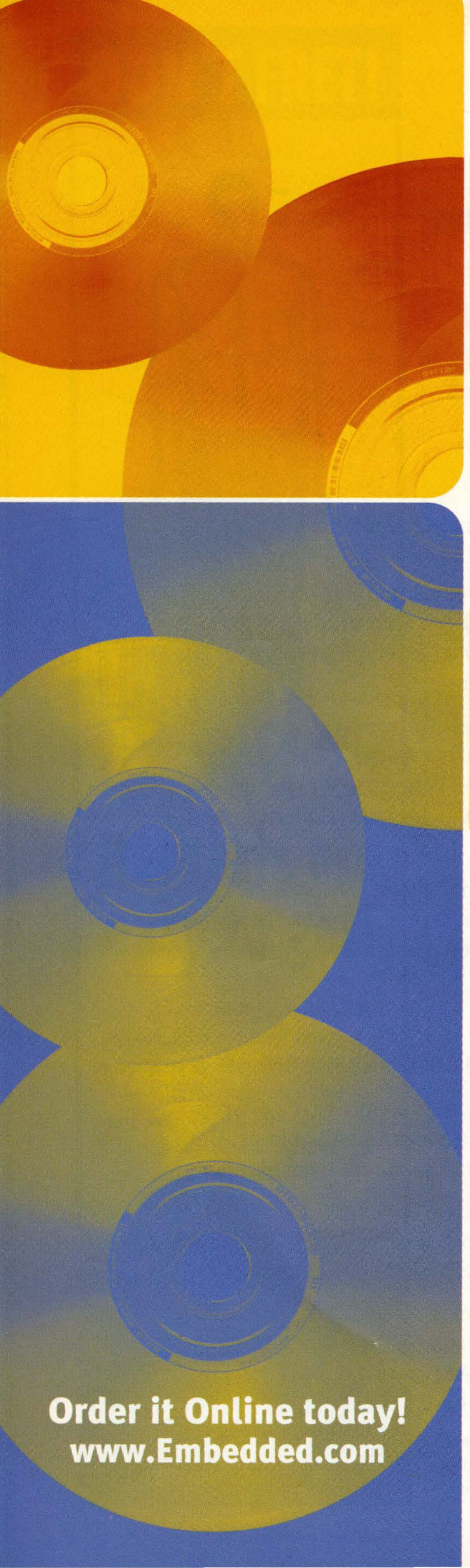
Languages: C, assembler.

Architectures: 8051, PIC, 68000, 68HC11, 6805, V8, Z80, 8086, H8/300.

Personal: Very easy-going, undemanding but highly supportive of co-workers. Not a self-starter, I prefer to work in a team, small or large. I'm tireless and will work 24 hours a day if required, even skipping lunch breaks.

Interests: Wow, where do I start - I've been involved in rocketry, space, entertainment, defence, railroads, motor vehicles, domestic appliances, toys, you name it!

Goals: I want to be on your desktop. I want to make your application sing. I want to bang your bits, clear your RAM and crunch your calculations. Just give me a chance - I promise you won't regret it!!



Order it Online today!
www.Embedded.com

New for 2000

You know you need it... ...so get it!

$79.95 new
$29.95 upgrade

EmbeddedSystems PROGRAMMING

CD–ROM Library 1988–1999

The *Embedded Systems Programming* CD-ROM Library Release 5.0 contains all columns, features, news items, editorials, and source code from the 1988 premiere issue through the December 1999 issue. This time-saver contains a powerful text search engine and is a must-have for veteran readers and for those new to *Embedded Systems Programming*, the pre-eminent source of embedded development for more than 11 years.

Special Features Include:

- A powerful text search engine
- More than 1,200 articles—all columns and features
- The entire 2000 Buyer's Guide—more than 1,200 products covered in detail
- Code you can copy directly into your designs
- Windows, Unix, and Mac compatibility
- All past and present author biographies

ways to order:

online
www.Embedded.com

e-mail
orders@mfi.com

phone
(800) 444-4881 U.S./Canada
(785) 841-1631 other countries

Reserve your CD-ROM now!

Dan Saks

More on Overloading with const

Last month, I showed you a concrete example of overloaded functions that differ only in their use of const qualifiers ("Overloading with `const`," December 1999, p. 81). My focus was on explaining why you would want to overload functions this way. This month, I'll build on that example by showing you alternative implementations for those functions, which yield either more compact code or faster performance.

Searching sequentially

My example is a pair of overloaded functions named `find`. Both functions search an array of `double` for a particular value; however, one searches a nonconstant array, while the other searches a constant array. The function declarations are:

```
double *
find(double v, double *a, int n);

double const *
find(double v, double const *a, int n);
```

Each function has a return type that's the same as its pointer parameter type. The first function searches a nonconstant array and returns a pointer to a nonconstant element in that array. The second one searches a constant array and returns a pointer to a constant element in that array. I'll refer to the first of these functions as `find` (nonconstant) and the second as `find` (constant).

For example,

```
p = find(d, x, m);
```

finds the first occurrence of value `d` within the first `m` elements of array `x`. If `x` is an array of nonconstant `double`, then this expression calls `find` (nonconstant) and returns a "pointer to nonconstant `double`." If `x` is an array of constant `double`, then this expression calls `find` (constant) and returns a "pointer to constant `double`." In either case, if the function fails to find a matching element, it returns a null pointer.

Overloading on the const qualifier improves type safety, sometimes at the cost of greater code size. Here's how to avoid that cost, or get faster code in the bargain.

Avoiding duplicate code

Last month, I presented implementations for both functions. Not surprisingly, they are nearly identical. The definition for `find` (nonconstant) is:

```
double *
find(double v, double *a, int n)
    {
    double *p = a;
    int i;
    for (i = 0; i < n; ++i)
        {
        if (*p == v)
            return p;
        ++p;
        }
    return NULL;
    }
```

The definition for `find` (constant) differs from the one above only by the const qualifier in three places:

```
double const *
find(double v, double const *a, int n)
    {
    double const *p = a;

    // the rest is the same as above

    }
```

According to the C++ standard, casting a pointer-to-constant into a pointer-to-nonconstant does not, by itself, produce *undefined behavior.* (Undefined behavior is essentially erroneous run-time behavior.)

As I explained last month, overloading the `find` functions provides greater functionality without comprising safety. Specifically, this overloading lets you search constant as well as nonconstant arrays, and capture the results, without using casts in the function calls. Unfortunately, it also increases the size of the executable program by duplicating code.

You can eliminate the duplication by defining one of the `find` functions so that it simply returns the result of calling the other, as in:

```
inline
double const *
find(double v, double const *a, int n)
  {
  return find
   (v, const_cast<double *>(a), n);
  }
```

This definition for `find` (constant) hands all the work to `find` (nonconstant). Although the call in the return statement appears to be recursive (a function calling itself), it isn't. The `const_cast` in the argument list yields an argument of type "pointer to (nonconstant) `double`," so overload resolution selects `find` (nonconstant) as the best match for the call.

Yeah, yeah, I know it uses a cast, and I've been telling you that casts are hazardous, and that you should stay away from them. They are hazardous, but they still have occasional uses and this is one of them. Using a cast here is an engineering trade off. I'm risking a tiny bit of safety to avoid duplicating code.

Actually, there's hardly any risk here. I'm using a cast in only one spot, packaged so that code that calls either `find` function never sees the cast. That's clearly better than scattering casts around the program. Moreover, you can easily verify—just by reading the code—that this one cast does no harm.

According to the C++ standard, casting a pointer-to-constant into a pointer-to-nonconstant does not, by itself, produce *undefined behavior.* (Undefined behavior is essentially erroneous run-time behavior.) Undefined behavior results only if the program tries to use the converted pointer to write to a constant object. However, it's easy to see that `find` (nonconstant) does not write to any elements in the array that it searches, so it cannot produce undefined behavior even when searching an array that's actually constant.

The previous function heading uses the keyword `inline` to ask the compiler to translate calls on the function as inline expansions. When it expands a function call inline, the compiler generates code for the function body at the call site instead of generating code that jumps to a function elsewhere in the program. Since a `const_cast` is just a compile-time type conversion that doesn't generate any code, the entire function body of `find` (constant) is just a call to `find` (nonconstant). Thus, a call to `find` (constant) should generate a call directly to `find` (nonconstant). In effect, the compiler generates code as if there's only one `find` function in the program.

Improving speed

If faster execution speed is more important to you than keeping the code small, you can use a slightly different, and faster, algorithm for `find` (nonconstant).

The loop inside `find` (nonconstant) performs two comparisons on each iteration:

```
for (i = 0; i < n; ++i)
  {
  if (*p == v)
      return p;
  ++p;
  }
```

The comparison in the `for` statement (`i < n`) terminates the loop after the search has examined every element in the array. The comparison in the nested `if` statement (`*p == v`) terminates the loop if it finds an element equal to `v`.

If you place a copy of `v` just beyond the end of the array, then the loop no longer needs a separate test to prevent it from searching too far. It will always find a matching element. If it finds a match before the end of the array, then the search succeeded. Otherwise, the search failed. The loop simplifies to just:

```
for (p = a; *p != v; ++p)
    ;
```

but it must be followed by a test such as:

```
if (p < a + n)
    return p;
```

to determine if the search succeeded.

The problem with this approach is that it assumes that array `a` has room to store an element at index `n`. This is not always a safe thing to do. The `find` (nonconstant) function searches an array of `n` elements. It doesn't know whether these `n` elements are the entire array, or part of a larger array. If the array has only `n` elements, then there's no room to store an element at index `n`.

A safer approach is to store a copy of `v` into the last element of the array (at `a[n-1]`), rather than just beyond the array (at `a[n]`). Even though find (nonconstant) searches a nonconstant array, it shouldn't alter that array permanently. It should save the old value of `a[n-1]` and put that value back when it's done.

Using this approach, the complete definition for find (nonconstant) looks like:

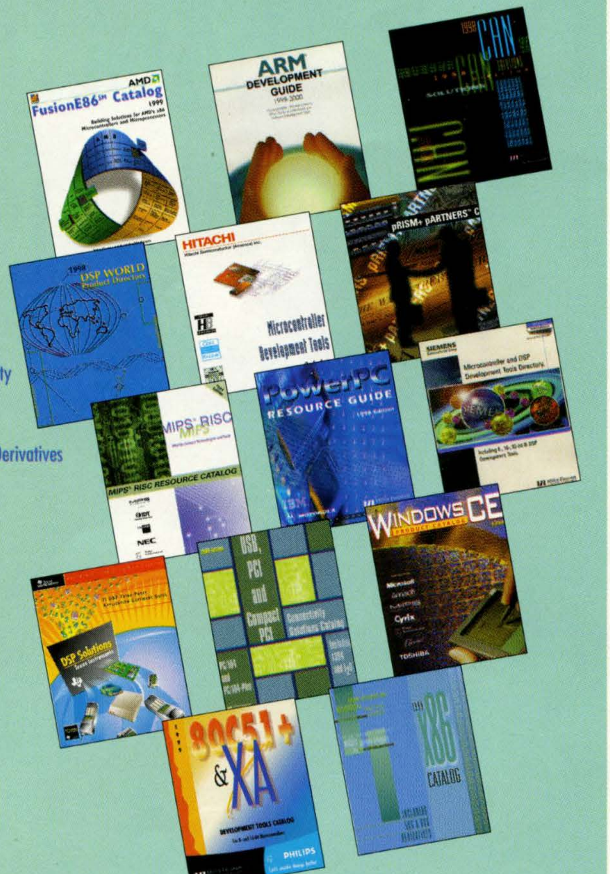
```
double *
find(double v, double *a, int n)
    {
    double *p;
    double t;
    if (n > 0)
        {
        t = a[n - 1];
        a[n - 1] = v;
        for (p = a; *p != v; ++p)
            ;
        a[n - 1] = t;
        if (*p == v)
            return p;
        }
    return NULL;
    }
```

The function copies a[n-1] into local variable t, copies v into a[n-1], and then searches a. It copies t back into a[n-1] after searching but before comparing *p with v to determine if the search succeeded. That way, if the search terminated with p pointing to a[n-1], the success of the search will be determined by the original value in a[n-1], not by the copy of v that was there temporarily.

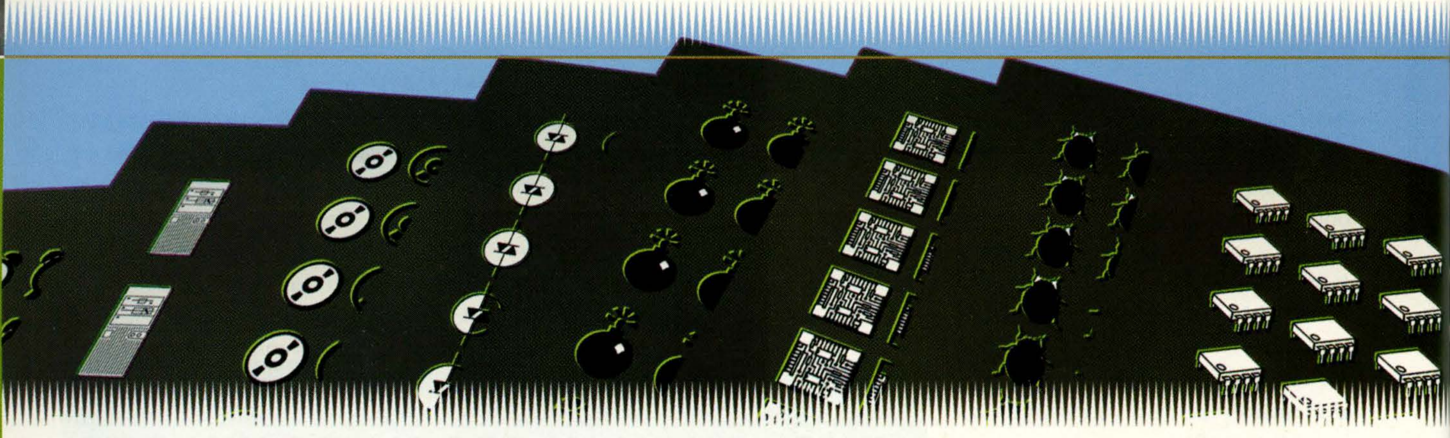
If you use this faster algorithm for find (nonconstant), it's no longer safe to use an implementation for find (constant) that calls find (nonconstant). If find (constant) passes its array argument to find (nonconstant), and the array happens to be in read-only memory, the program will lapse into undefined behavior when find (nonconstant) tries to store v into a[n-1].

Thus, if find (nonconstant) uses an algorithm that modifies the array, even temporarily, then find (constant) must use a different algorithm (such as the original one) that never modifies the array. The price for using this faster algorithm is a program that will be larger because it contains code for two different find algorithms. **esp**

Dan Saks is the president of Saks & Associates, a C/C++ training and consulting company. He is also a contributing editor for the C/C++ Users Journal. *He served for many years as secretary of the C++ standards committee and remains an active member. With Thomas Plum, he wrote* C++ Programming Guidelines. *You can write to him at dsaks@wittenberg.edu.*



HUGH O'BYRNE



Data Memory Paging Management

Mapping logical registers to physical registers requires the management of paged data memory. This two-part article explains a method to detect any potential paging errors in assembly programs.

Something is wrong, and yet nothing is wrong. I've discovered the counter isn't counting. I have the bug traced down to one line, which I *know* is executing:

```
INCF counter,f
```

The register refuses to change value, no matter how long or how hard I stare at the instruction. I check the top of the file for the line:

```
counter equ h'20'
```

I check the data book for the syntax of the instruction. I check that the object file has the correct binary values for opcode and parameter. Everything seems correct, but the program does not work. Disbelief, confusion, and even despair attack me. Finally, I painstakingly trace the flow of execution of the program, and discover that the data memory page isn't set correctly when the instruction is executed. After a day of effort, this little piece of information leads me to add just one more instruction, and the entire program works. I simultaneously feel frustration that such a small obstacle could have stymied me for so long and relief that the program is finally working properly.

This kind of scenario *used* to happen to me, until I devised a method to detect any potential paging errors in the assembly programs I write. This two-part article describes the method I use, which is applicable to any processor that has paged data memory. The first part expains what paging is, why it's implemented on processors, how problems can arise, and the essence of the method for preventing such bugs. The second part adds details so that you can use the method to write nontrivial assembly language programs. Anyone who has suffered from a bug due to a paging error may find that part interesting and useful. I still occasionally make an error in paging, but much less frequently; and when it does happen, I find the error far more quickly than before.

Why paging?

To begin, I think an explanation is in order as to why paging exists in the

I check the data book for the syntax of the instruction. I check that the object file has the correct binary values for opcode and parameter. Everything seems correct, but the program doesn't work.

first place. The reasons go back to processor design. One of the key decisions in designing a processor is what instruction set to give it. An instruction set is essentially a set of rules that defines what the processor does for each combination of values stored in program memory.

Designers want to make the instruction set as small as possible, to most efficiently use the limited program memory. One subset of this instruction set may add the contents of a register to an accumulator, that is, the set of instructions `{add register 0 to accumulator, add register 1 to accumulator, ...}`. The instruction set is limited, which means this subset is limited, so only a limited number of registers can be added to the accumulator at any time. For example, if 16 values in the instruction set correspond to this `add` instruction subset, then any one of 16 registers may be added to the accumulator. But a processor may have more registers—perhaps 64 of them. Naturally, you'd wish to be able to add *any* register to the accumulator.

You can do so by using a subtle trick. The set of `add` instructions is redefined as `{add logical register 0 to accumulator, add logical register 1 to accumulator,...}`. It remains true that under a fixed set of circumstances, an `add` instruction can only reference one of 16 registers. The trick is to have different sets of circumstances, so that a *logical* register does not always reference the same *physical* register. If you have four different sets of circumstances, the `add` instruction could be used to access any one of the 64 physical registers. For instance, under one set of circumstances, logical register 0 references physical register 0, logical register 1 references physical register 1, and so on, through logical register 15 references physical register 15. Under a second set of circumstances, logical register 0 references physical register 16, logical register 1 references physical register 17, and so on, through logical register 15 references physical register 31. Under a third set of circumstances, logical registers 0 to 15 reference physical registers 32 to 47. Under the fourth set of circumstances, logical registers 0 to 15 reference physical registers 48 to 63. In this way, any one of 64 physical registers may be referenced while using an instruction (sub)set a quarter of the size that would be needed if paging weren't used. The set of circumstances that determines how the logical registers map to physical registers is called the *current page* setting, or the *active page.*

This economy of memory, however, can lead to tricky bugs arising in low-level code.

Pseudocode

Assembly languages and paging mechanisms can vary significantly among processors, so I use pseudocode in my examples. I shall fully define an imaginary processor's instruction set and paging, so no one is at a disadvantage for being unfamiliar with a given platform. The methods can easily be translated to many real assembly languages.

The pseudocode I use has one accumulator, labeled `A`. The assembly instructions all have at most one argument, labeled `X`, which is a register (`reg`) or an immediate value (`imm`). Anyone who has done extensive programming would recognize a table with mnemonics on one side and expressions such as `A<-A+X` on the other. Instead of using mnemonics, I shall use the expressions themselves. A single pseudo-assembly instruction is of the form: *expression, argument type, argument.* Thus, to add a register named `AddMe` to the accumulator, the instruction is:

```
A<-A+X  reg     AddMe
```

The `X` in the expression stands for the argument. The `reg` indicates that the argument is a register, and `AddMe` is the argument itself. To increment `A`, the pseudo-assembly instruction is:

```
A<-A+X  imm     1
```

`A` is the destination of all such "common" expressions except one:

```
X<-A    reg     RegName
```

Two "uncommon" expressions exist:

```
Page<-A
A<-Page
```

The first sets the active page to `A`. The second puts a value in `A` corresponding to the current active page.

There is one more instruction:

```
GOTO    imm     AddressLabel
```

Address labels occur at the beginning of lines, with a colon after them:

```
AddressLabel:
```

(I'll discuss the pseudocode for bit addressing, conditional statements, and subroutine calls later.)

The argument in the `GOTO` instruction must always be a label, and a label cannot be on the same line as an instruction. The semicolon (`;`) is the symbol used for comments. Wherever a semicolon symbol appears on a line, it is ignored, along with the rest of the line.

There are 16 logical registers, 46

RUPERT ADLEY

> **Different areas of data memory may even have different, independent paging schemes. What's worse is that different areas may have different but dependent paging.**

physical registers, and four pages in this (imaginary) processor. The 16 logical registers are named r0 to rF. Forty of the physical registers are paged. They are in four pages of 10 registers each, labelled p00 to p09 for page 0, p10 to p19 for page 1, p20 to p29 for page 2, and p30 to p39 for page 3. Six physical registers are unpaged, and these are labelled puA to puF. Register puA is the accumulator. Logical registers rA to rF always refer to the unpaged physical registers puA to puF. Logical registers r0 to r9 refer to paged physical registers. When page 0 is active, r0 to r9 refer to p00 to p09; when page 1 is active, r0 to r9 refer to p10 to p19; and so on. All instructions that reference a register (that is, all instructions with `reg`) reference by logical register.

Note that this isn't the only way to manage paging. For some processors, a physical register may map to logical registers in more than one page, but not in all pages, or a physical register may map to one logical register in one page and a different logical register in another page. Different areas of data memory may even have different, independent paging schemes. (What's worse is that different areas may have different but dependent paging.) I've chosen the previous example because it's rather simple. The methods I describe in this article can be modified to accommodate other paging schemes, if necessary.

Problem

Imagine that register p20 were to be used as a counter, and register p23 were to be used as a mask register. An assembly language programmer might use the following lines:

```
#define CounterReg r0
#define MaskReg r3
```

and code such as the following may appear in the program:

```
Label1:
    A<-X   imm  2
    Page<-A
    A<-X   reg  CounterReg
    A<-A+X imm  1
    X<-A   reg  CounterReg
    GOTO   imm  WhatNext
...
WhatNext:
    A<-X   reg  MaskReg
...
```

Nothing is wrong with this code as it stands. If execution starts from `Label1`, then `Page` is set to 2. `CounterReg` (r0) refers to p20, as intended. The program goes to `WhatNext`, `Page` is still 2, and `MaskReg` (r3) refers to p23 as intended. If, however, the programmer uses p05 as a checksum holding register, there may be more `#define`s:

```
#define ChecksumReg r5
#define DataReg rD
```

Elsewhere in the program, unless the programmer is careful, might be another code fragment:

```
Label2:
   A<-X   imm  0
   Page<-A
   A<-X   reg  ChecksumReg
   A<-A+X reg  DataReg
   X<-A   reg  ChecksumReg
   GOTO   imm  WhatNext
```

Neither code fragment alone is wrong. But if they're in the same program, trouble will arise. When execution starts from `Label2`, `Page` is set to 0. References to `ChecksumReg` (r5) will access p05, as intended. `DataReg` (rD) will reference puD regardless of the value of `Page`. After going to `WhatNext`, with `Page` still at 0, `MaskReg` (r3) will not map to the intended physical register (p23) but to a different register (p03). This kind of trouble can be serious and difficult to track down. When an instruction references a paged register, the current page must be set correctly. To check that this is the case, you must follow every possible path of execution the program may take to that instruction and find the last write to `Page` before the instruction. Checking execution paths from a particular point can be relatively easy, but checking every execution path to a point is usually much trickier.

The fundamental cause of the paging problem is that the assembler will only recognize logical registers, not physical registers, and it does not keep track of possible values held in `Page` at every instruction that references a paged register. (Actually, expecting an assembler to do so would be unreasonable. The process is quite difficult to automate completely and correctly.) In the above example, the register whose task it is to hold the mask is physical register p23, but it can only be referred to in assembly language as r3, and r3 doesn't unambiguously reference p23; it could also reference p03, p13, or p33.

Register names and elementary paging tracking

The purpose of this article is to describe a method of checking that each identifier of a paged physical register actually refers to the intended register, before the assembly is performed. The first step in eliminating paging errors is to remove the ambiguity. A register identifier must represent a unique physical register, rather than a logical register. The ambiguity is inherent in the assembler, so the programmer must enforce this rule. Adding a prefix or suffix to a register name to indicate which page it's in would be a suitable naming convention. Thus, in the previous example, the counter and mask registers may be called `CounterReg_P2` and `MaskReg_P2`, and the checksum register may be called `ChecksumReg_P0`. Note that

ICSPAT Call For Papers

International Conference on Signal Processing Applications and Technology

Abstracts due March 1, 2000

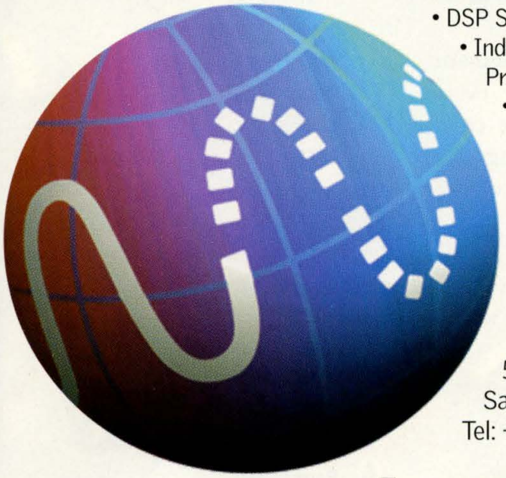
Post Abstracts to: www.dspworld.com

Submission of an abstract implies a commitment to present a paper.
Participation in the program is conditional upon adhering to the deadline and procedures outlined in this notice.

ICSPAT
October 16-19, 2000
Dallas, TX, USA

APPLICATION AREAS:

• Active Noise Control • Advanced TV and Video Technology • Aerospace • Audio • Automotive • Biomedical • Communications • Consumer Products • Data Communications • DSP Architectures • DSP Systems • FPGAs in DSPs • Image Processing • Image Understanding & Restoration • Industrial Applications • Multimedia • Neural Networks • Parallel Processing • Radar Rapid Prototyping • Speaker Identification • Speech Analysis & Synthesis • Speech Recognition • Telephony • Underwater/SONAR • VLSI Architectures • Wavelets and more. . .



PLEASE SEND A 500-WORD ABSTRACT VIA THE WEB TO WWW.DSPWORLD.COM BY MARCH 1, 2000. IF SENDING AS AN ATTACHMENT, PLEASE SEND TEXT IN MICROSOFT WORD OR ASCII TEXT FORMAT. IF YOU NEED TO MAIL OR FAX YOUR PAPERS, THEY MAY BE SENT TO:

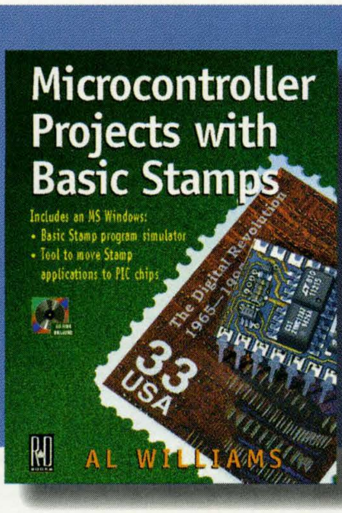
Deanna Young • ICSPAT
CMP Media Inc.
525 Market St., Suite 500
San Francisco, CA 94105 USA
Tel: +1 415.538.3860 • Fax: +1 415.278.5390

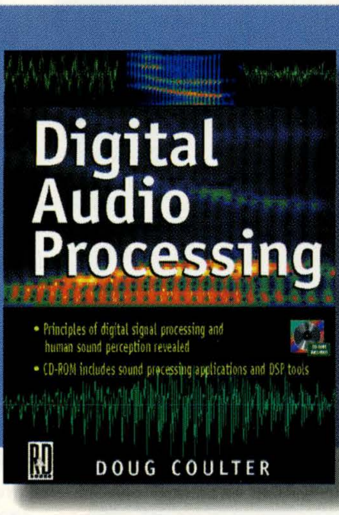
For more information on submitting an abstract, visit our Web site at www.dspworld.com or check out the 1999 "virtual conference" at www.icspat.com.

CEMF

CMP

www.rdbooks.com


Microcontroller Projects with Basic Stamps
Includes an MS Windows:
• Basic Stamp program simulator
• Tool to move Stamp applications to PIC chips
The Digital Revolution 1965—196
33 USA
AL WILLIAMS


Digital Audio Processing
• Principles of digital signal processing and human sound perception revealed
• CD-ROM includes sound processing applications and DSP tools
DOUG COULTER

Beginners will find clear explanations of the principles needed to design hardware and program the Basic Stamp. Seasoned designers will discover a fast and easy means of prototyping microcontrollers – with a task-oriented command reference, vital information on analog I/O, serial communications, and LCDs. Inludes a tool to convert Basic programs to assembly language required to run the PIC microcontroller.

ISBN 0-87930-587-8 $44.95

DSP and human sound perception explained in accessible language! Includes techniques to compensate for acoustic and instrument limitations, a treasure trove of C++ software tools for Windows 32-bit sound programming, and DSP and User Interface advancements for the WinTel platform.

ISBN 0-87930-566-5 $59.95

R&D BOOKS

Find R&D Books in your local bookstore

or call 800-500-6875 / Fax 408-848-5784
e-mail: rd@rushorder.com

ESP999

The programmer must take on another task, which is to determine at all points in the program whether any particular page is active and if so, which one.

DataReg needs no suffix because it's accessible no matter which page is active.

The programmer must take on another task, which is to determine at all points in the program whether any particular page is active and if so, which one. For example, using {} to enclose the relevant information, the first code fragment would be written:

```
Label1:
    ; {Page=0,1,2,3}
    A<-X  imm   2
    ; {Page=0,1,2,3}
    Page<-A
    ; {Page=2}
    A<-X  reg   CounterReg_P2
    ; {Page=2}
    A<-A+X      imm     1
    ; {Page=2}
    X<-A  reg   CounterReg_P2
    ; {Page=2}
    GOTO  imm   WhatNext
...
WhatNext:
    ; {Page=2}
    A<-X  reg   MaskReg_P2
    ; {Page=2}
...
```

The comments merely reflect the current status of the active page. For most instructions, the comment on the line after the instruction is exactly the same as the the comment on the line before. There are two exceptions: instructions that modify Page, and flow-control instructions (such as GOTO). For instructions that write to Page, the comment on the line after reflects the effect of that write. For GOTO, the comment on the line after the *destination* (not after the GOTO itself) must accommodate the comment before the GOTO instruction. For labels, the comment after a label must accommodate the comment before (if there is a relevant comment immediately before) and must also accommodate the comments before any GOTOs to that label.

You can see that this code fragment will run correctly because for every paged register referenced, the comment on the line preceeding the instruction matches the page of the register.

Attacking the problem

Now, the second code fragment would be written:

```
Label2:
    ; {Page=0,1,2,3}
    A<-X     imm    0
    ; {Page=0,1,2,3}
    Page<-A
    ; {Page=0}
    A<-X     reg    ChecksumReg_P0
    ; {Page=0}
    A<-A+X  reg     DataReg
    ; {Page=0}
    X<-A     reg    ChecksumReg_P0
    ; {Page=0}
    GOTO     imm    WhatNext
```

You can see that the two code fragments, when written like this, will not work together because the comment after the label WhatNext doesn't accommodate the comment before the GOTO imm WhatNext instruction. (And if the comment after the label WhatNext were to be changed, you can see that the register MaskReg_P2 may be referenced when Page is not 2.) Two ways to modify the code exist such that the two fragments will run together. The problem could be fixed before the GOTO, so the second fragment would be rewritten:

```
Label2:
    ; {Page=0,1,2,3}
    A<-X      imm    0
    ; {Page=0,1,2,3}
    Page<-A
    ; {Page=0}
    A<-X      reg    ChecksumReg_P0
    ; {Page=0}
    A<-A+X   reg    DataReg
    ; {Page=0}
    X<-A      reg    ChecksumReg_P0
    ; {Page=0}
    A<-X      imm    2
    ; {Page=0}
    Page<-A
    ; {Page=2}
    GOTO      imm    WhatNext
```

Or the problem could be fixed after the GOTO destination, so the first fragment would be rewritten:

```
Label1:
    ; {Page=0,1,2,3}
    A<-X      imm    2
    ; {Page=0,1,2,3}
    Page<-A
    ; {Page=2}
    A<-X      reg    CounterReg_P2
    ; {Page=2}
    A<-A+X    imm    1
    ; {Page=2}
    X<-A      reg    CounterReg_P2
    ; {Page=2}
    GOTO      imm    WhatNext
...
WhatNext:
    ; {Page=0,2}
    A<-X      imm    2
    ; {Page=0,2}
    Page<-X
    ; {Page=2}
    A<-X      reg    MaskReg_P2
    ; {Page=2}
...
```

In this type of situation, setting the page after the GOTO destination is more common, as in the latter example. If there are several GOTOs to the same label, setting the page once after the label is usually better than doing so several times before the GOTOs. This strategy makes it easier to modify the code later; if you add another GOTO, you don't have to remember to set Page beforehand.

Shorthand

If you were to follow this method rigorously, you'd need a comment for

Every instruction that accesses a paged register is effectively accompanied by a proof that the correct page is active at the time, regardless of which execution path led to that instruction.

nearly every instruction and label in the program, which would make the wordiness of the comments in the previous examples tiresome. For shorthand, I use four characters to show the possible paging values. I use a - symbol for a page that is definitely not selected, and a # symbol for a page that may be, or definitely is, selected. Thus, when the current page is unknown, the comment would look like: *;####*. When you know the active page is page 1, the comment would be *;-#--* (taking the first character to represent page 0). When the active page may be either 1 or 2, the comment would be *;-##-*. In addition, I put comments on the same line as the previous instruction (or label). Thus, using the second fix described above, the code fragments (put together) look like this:

```
#define DataReg rD
#define ChecksumReg_P0 r5
#define CounterReg_P2 r0
#define MaskReg_P2 r3
...
Label1:                       ;####
   A<-X imm  2                ;####
   Page<-A                    ;--#-
   A<-X reg  CounterReg_P2    ;--#-
   A<-A+X    imm  1           ;--#-
   X<-A reg  CounterReg_P2    ;--#-
   GOTO imm  WhatNext
...
WhatNext:                     ;#-#-
   A<-X imm  2                ;#-#-
   Page<-X                    ;--#-
   A<-X reg  MaskReg_P2       ;--#-
...
Label2:                       ;####
   A<-X imm  0                ;####
   Page<-A                    ;#---
   A<-X reg  ChecksumReg_P0   ;#---
   A<-A+X    reg  DataReg     ;#---
   X<-A reg  ChecksumReg_P0   ;#---
   GOTO imm  WhatNext
```

Note that the rule for GOTOs and GOTO destinations is not as clear as for the other instructions: from one place, page 2 may be active before a GOTO instruction and from another place, page 0 may be active before a GOTO instruction. If they both jump to the same place in the program, that place may have either page 2 or page 0 active. Symbolically, where the comment before the GOTO has a #, so must the comment after the label. Where the comment before a GOTO has a -, the comment after the label may have either symbol.

Summary of the rules to date

This is a good point to take stock of what we've learned. The logical rules we have gathered are as follows:

- L1: variables must be identified and tracked not by ambiguous logical registers, but by unambiguous physical registers
- L2: wherever a paged variable is referenced, the page associated with the variable *must* be active—and equally important, all other pages *must be inactive*
- L3: an instruction that modifies Page will change the active page
- L4: any page that may be active before an instruction that does not modify Page and is not a flow-control instruction may be active after that instruction
- L5: any page that may be active before a flow-control instruction may be active at the destination(s)
- L6: any page that may be active before encountering a label may be active after the label

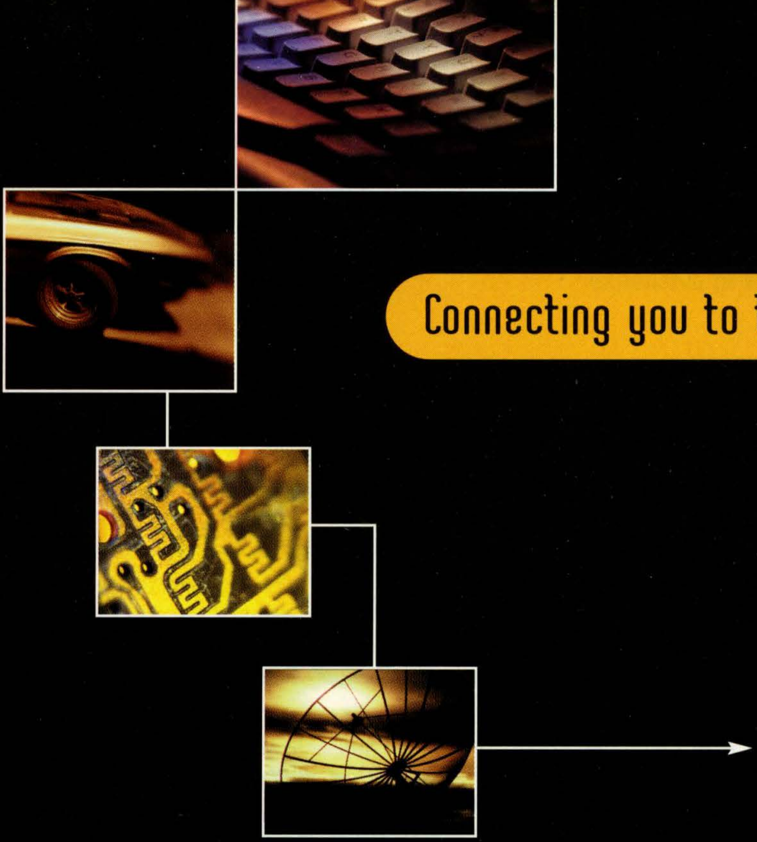
These rules can be expressed in symbolic terms:

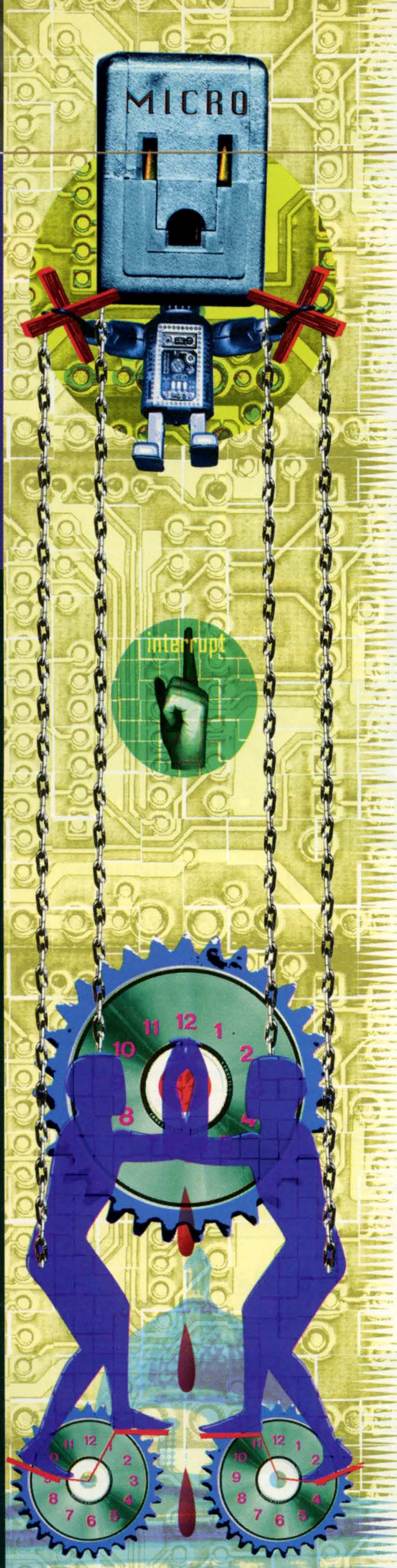
- S1: any variable name given to a register in page 0 must have the suffix _P0, any variable name given to a register in page 1 must have the suffix _P1, and so on
- S2: any instruction that references a variable name with a suffix _P0 must be accompanied with a comment: *;#---*. Any instruction that references a variable name with a suffix _P1 must be accompanied with a comment: *;-#--*, and so on
- S3: an instruction that modifies Page has a comment after it that reflects the new value of Page
- S4: where a # appears in a comment before an instruction that does not modify Page and is not a flow-control instruction, so must a # appear in the comment after it
- S5: where a # appears in a comment before a GOTO, so must a # appear in the comment after the destination label
- S6: where a # appears in a comment before a label, so must a # appear in the comment after the label

The purpose of rule 1 is to identify variables not just by logical register, but by physical register as well. The purpose of rule 2 is to ensure that every reference to the logical register associated with a variable will, in fact, access the correct physical register. The purpose of rules 3 to 6 is to keep track of which page(s) may be active at any time.

This article represents the essence of my paging method. When the method is strictly adhered to, every instruction that accesses a paged register is effectively accompanied by a proof that the correct page is active at the time, regardless of which execution path led to that instruction. In the second part of this article, I will expound on the rules to cover conditional statements and subroutines, thus making the method suitable for use in nontrivial programs. **esp**

Hugh O'Byrne graduated from University College Dublin in 1995 with a BS in computer science. He's currently pursuing an MS in mathematics.





RICHARD WALL

How to Increase Interrupts in an MCU Design

In this article, the author looks at how a hardware/software co-design solution can stretch a given processor's usefulness. He describes an interrupt controller with a serial interface for microcontrollers.

When I teach microcontroller design concepts, I teach the students how to manage three of the most important processor resources, namely time, memory, and I/O pins. The challenge for the engineer is to squeeze the last drop of functionality out of a microcontroller before submitting to a more expensive processor or changing design platforms. Interrupts are the key to real-time control using microcontrollers. Many microcontrollers include interrupts as a resource for real-time management of internal resources such as timers and A/D converters, but allow for limited numbers of interrupts from external sources. In this article, I will look at three resources—time, I/O pins, and interrupts—and will demonstrate how a hardware/software co-design solution using commonly available components can stretch a given processor's usefulness.

Microcontrollers have found a

The challenge for the engineer is to squeeze the last drop of functionality out of a microcontroller before submitting to a more expensive processor or changing design platforms.

niche in real-time control because they are inexpensive and reliable. Fundamental to real-time control is the ability to control some aspect of the environment in response to some external stimuli, normally called events. Two approaches to detecting events are commonly used: polling and hardware-generated interrupts.

When a microcontroller polls for the occurrence of an event, the I/O pin or bit in a status byte is continually tested to determine its high or low state and subsequently execute the appropriate code. Obviously, when the microcontroller is busy testing bits it can't perform other tasks. If the processor is off running some low-priority task when an event requiring immediate attention occurs, the response to the high-priority event may be delayed or missed entirely. The necessity of more timely responses to an occurrence requires more time spent looking at the bit that indicates the status of the event.

No new information can be gained from a signal that never changes. Just like watching out the window for the mailman, watching for bits to change is wasted time—unless, of course, you have nothing better to do. Polling should be restricted to low-priority events and those with relatively lengthy persistence.

Hardware interrupts have the advantage of more timely execution of event processing. Of course, a conservation of woes exists in that hardware interrupts make processor designs more complex and hence more costly. In addition, you must devote engineering effort to manage multiple autonomous events. Many good texts describe different ways to manage interrupt-driven real-time operating systems. My experience with embedded microcontroller design has taught me that interrupt-handling schemes vary depending on how much the application pushes the capabilities of the chosen microcontroller.

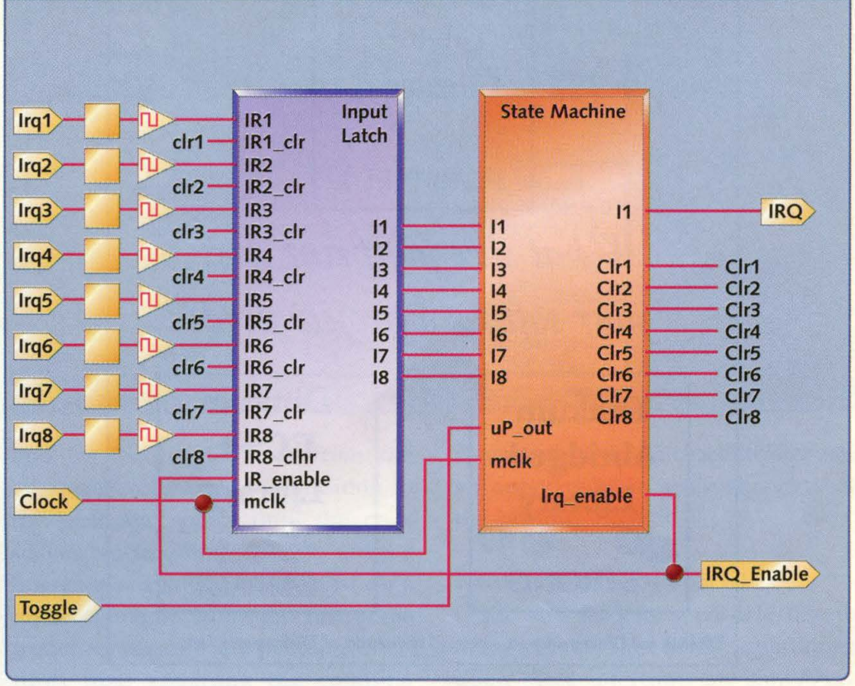
Microprocessors used in PCs and workstations have relatively few external interrupts and rely on priority interrupt encoder ICs to manage external interrupts. They interface to the microprocessor using I/O addressing space or mapped-memory I/O. Either way, they interface through the address and data buses. Microcontrollers frequently have self-contained memory systems and, hence, no external address and data buses. Most, if not all, microcontrollers use a variety of internally and externally generated interrupts to manage the timely processing of scheduled and irregularly occurring events. External interrupts require input pins to sense the event.

Regardless of the number of I/O pins a microprocessor may have, a truism in microprocessor design maintains that the application will always expand to the point that additional I/O is needed. Frequently, design requirements force the engineer to seek resources beyond the capability of any microcontroller, such as real-time clocks, 12-bit analog conversion, and large amounts of nonvolatile memory. When the speed of the application permits, I look for ICs that have serial interfaces with these off-chip resources. The design that follows is an interrupt controller with a serial interface for microcontrollers. I call this design SOIE, which is an acronym for serial output interrupt expander.

What can it do?

The PLD-based SOIE design presented here is capable of prioritizing eight interrupts while using only one inter-

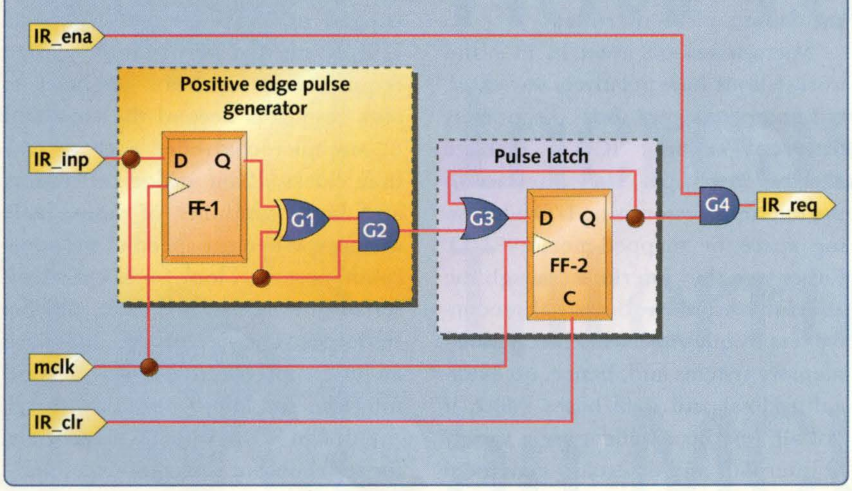
FIGURE 1 Basic block diagram of the microcontroller interrupt expander



BEN FISHMAN

rupt pin and one output pin on the microcontroller. The SOIE has only two constraints: the microcontroller must be able to sense the state of the interrupt pin and an external master clock must be provided. The frequency of this master clock must have a period that is less than the minimum time of any interrupt that is expected to be asserted.

FIGURE 2 Interrupt input edge trigger and latch logic



Basically, the resource management strategy trades time for I/O pins. Once an interrupt on one of the `Irq1` through `Irq8` inputs to SOIE is asserted, the `IRQ` goes high to interrupt the microcontroller. The microcontroller then toggles the `Toggle` pin until the SOIE `IRQ` output line goes low. The number of times the output pin is toggled corresponds to the highest priority interrupt asserted. `Irq1` has the highest priority in this design.

Here the output interrupt signal and the input interrupt request signals are all positive level or active high. If a particular application requires either or both inputs and output to be active low, the changes to the design border on trivial, as you will soon see.

How does it work?

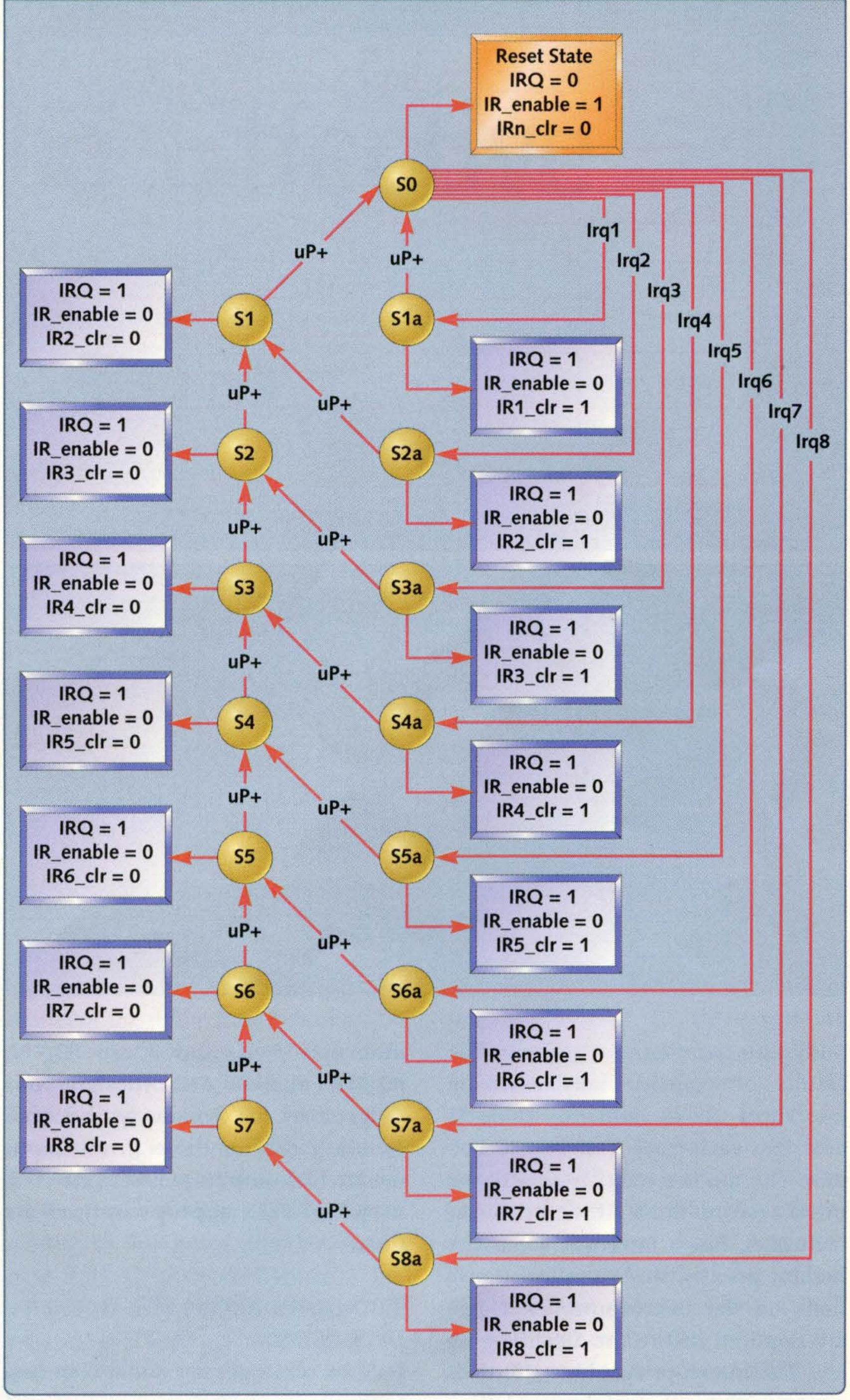
As the block diagram in Figure 1 shows, the design is broken into two parts: the input latching module and the pulse-counting state machine module. I completed the design using ISP Synario starter software by Lattice Semiconductor Corp. I designed the input latch module using schematic entry and the `IRQ` state machine module using ABL HDL. As with most schematic entry design tools, connectivity can be achieved in one of two ways: by using lines to connect pins together or by calling multiple signals by the same name. The signals labeled `clr1` through `clr8` are examples of connection by label.

Each interrupt input signal is first preprocessed using the circuit shown in Figure 2. Although synchronizing can delay interrupt by one master clock cycle, it guarantees that the `Irq` inputs will have a minimum persistence and eliminate glitches caused by internal logic race conditions. The logic provided by `FF-1`, `G1`, and `G2` generates a pulse on each positive transition of the interrupt input signal. This ensures that the input signal generates only one interrupt each time the signal goes high. Since the SOIE master clock provides the `mclk` signal, the interrupt input signal must be asserted as long as the period of the master clock. The pulse that is generated by the positive transition detector is latched in FF-2 and is stored until the latch is cleared by the IR_clr signal using one of the clear signals clr1 through clr7 generated by the IRQ state machine. G4 inhibits the interrupt stored in FF-2 from interfering with the IRQ state machine if the state machine is in the process of resolving a previous interrupt.

Since the SOIE design is capable of handling eight interrupts, eight of the input logic modules shown in Figure 2 are connected together. The master clock signal, mclk, and the interrupt enable signal, IR_enable, are common to all eight modules. The IRQ state machine module generates the individual clear interrupt signals, IR$_n$_clr.

Because only the simplest of state machines makes any sense in schematic form, the design now uses the ABL-HDL to describe the IRQ state machine module. Figure 3 graphically describes the transition control and outputs for the program in Listing 1, which you can find on-line at *www.embedded.com/code.htm.* In the reset state, the IRQ output to the microprocessor is low, the interrupt enable is high, and clear outputs to the input latch modules are low. Only two types of events cause transitions in the state table: positive transitions on the toggle input from the microcontroller and a positive transition on one of the interrupt inputs. The highest priority pending interrupt is selected by the conditional transitions from the reset state.

FIGURE 3 State diagram for SOIE state controller



When one or more interrupts are asserted from the input latch module, the state machine transitions to the "a" state associated with *N*, the highest priority interrupt currently asserted. While in the *Na* state, the clear output for module *N* is set to clear the latched interrupt event as previously described. When the input signal from the microcontroller makes the next positive transition, state *Na* transitions to state *N – 1* and the clear output is reset. Each subsequent positive transition on the toggle input from the microcontroller causes a transition to the next lower state until the reset state is reached. The IRQ line and the interrupt enable are conditional on a low-level toggle from the microcontroller to prevent any pending interrupts from restarting the counting chain before the microcontroller has an opportunity to sense the state of the IRQ line.

How does it perform?

I will now present a particle test to demonstrate the SOIE operations when three interrupts are simultane-

FIGURE 4 Results of three simultaneous interrupts

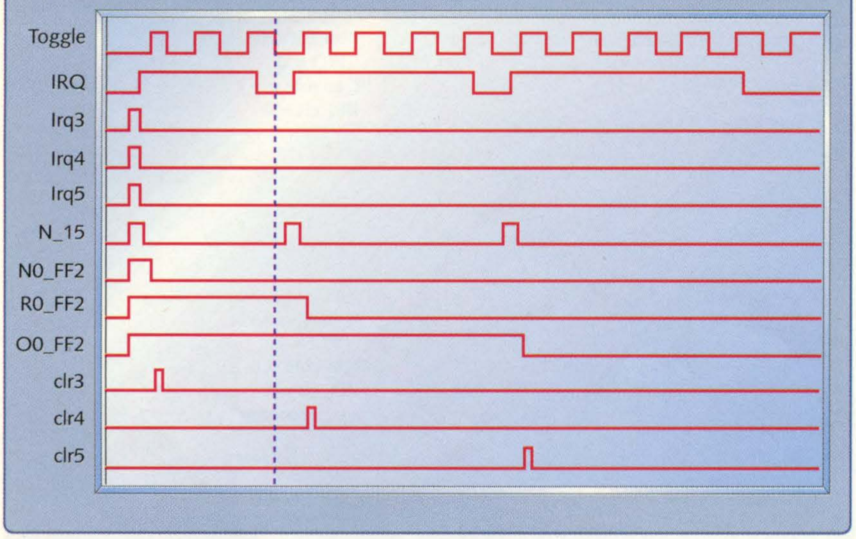


TABLE 1 Trace definitions for Figure 4

Trace	Function or Output
Toggle	Toggle output from the microcontroller to the SOIE
IRQ	The Interrupt request form the SOIE to the microcontroller
Irq3	Interrupt #3 input to the SOIE
Irq4	Interrupt #4 input to the SOIE
Irq5	Interrupt #5 input to the SOIE
N_15	Interrupt enable from the IR state machine module to the input latch module
NO_FF2	Latched interrupt #3 FF output
RO_FF2	Latched interrupt #4 FF output
OO_FF2	Latched interrupt #5 FF output
Clr3	Interrupt #3 clear signal from the IR state machine module to the input latch module
Clr4	Interrupt #4 clear signal from the IR state machine module to the input latch module
Clr5	Interrupt #5 clear signal from the IR state machine module to the input latch module

ously asserted. Figure 4 presents the results of coincidentally asserting interrupts 3, 4, and 5. Table 1 describes each trace from top to bottom. The top two traces represent the interface from the SOIE to the microcontroller. Since interrupt 3 has the highest priority, three positive transitions on the microcontroller toggle are required before the `IRQ` line goes low. The interrupt enable (`N_15`) does not go high until after the toggle line has gone low. After that time, interrupt 4 generates the next interrupt request. After four positive transitions on the microcontroller toggle, the `IRQ` line again goes low, after which time interrupt 5 generates the next interrupt request.

The latched interrupt outputs for interrupts 3, 4, and 5 are `NO_FF2`, `RO_FF2`, and `OO_FF2`, (designations assigned by the Synario synthesizer). As illustrated on those three traces, the latched outputs persist until `clr3`, `clr4`, or `clr5` appropriately clears them.

Thoughts about the design process

I always try to get my students to step back after completing a project and reflect on what they've learned from doing a design. Now it's my turn. I'm pleased with the EDA design tools for both simulating and synthesizing PLA designs. The hierarchical design approach can be worked from either a bottom-up or top-down approach with equal ease. I found that mixing HDL modules with schematic entry modules made the design proceed more quickly than attempting to use either method exclusively.

I wrote an HDL module representing the logic of Figure 2 to see which approach resulted in the simplest design. As I guessed, the results were identical. This outcome isn't surprising because Synario reduces all schematics to logic expressions before reducing them to a minimum realization. Since the SOIE design can be synthesized in a number of parts, the design present here becomes independent of the particular PLA chosen for implementation.

Good default setting on design tools speeds the design process. The design process is further simplified by eliminating the need to research the peculiarities of the particular parts chosen to implement the design. This is particularly important to those learning a new technology or tool, in which case the expert tool designers have better design knowledge than the users.

The SOIE hardware/software codesign solution takes advantage of the strengths of both the microcontroller and the PLD to inexpensively increase the capability of a microcontroller for real-time control. The speed of the PLD ensures that multiple events are recognized, even though their persistence may be very short. And the interface to the microcontroller requires only two I/O pins and minimal support code—a small price to pay for a useful solution. **esp**

Richard W. Wall received his BSEE from Pennsylvania State University and MEEE and PhD EE from the University of Idaho. He worked for Idaho Power Co. for 18 years in the communications, protective relaying, and R&D departments before joining the University of Idaho Department of Electrical Engineering. He now teaches Embedded Microcontrollers and Senior Design. His research interests include protective relaying and networked distributed control.

Don Morgan

Simple and Effective

For the past few months, I've been talking about how the technology of signal processing, specifically filter technology, is based on precepts that each of us knows and uses everyday. Much of the discussion has involved averages and moving averages that have yielded simple but effective FIR and IIR filters.

In this issue I'll present a simple filter, based on an analog filter, that's easy to work with and produces nice results: the state variable filter. It's made up of two integrators and some summing, just as with its analog equivalent. It's a remarkably simple device that produces some interesting effects.

State variable filter

The state variable filter is a popular design for analog products. It's easy to design, and easy to tune and stabilize compared to the Biquad and other multi-feedback filters. One of the most interesting aspects of this filter is that it will also produce as many as four possible outputs at once—lowpass, highpass, bandpass, and notch. As you can imagine, this capability makes it useful for many applications.

I've discussed many simple digital filters in this series, but this is the first second-order IIR filter. I'll begin by presenting the parameters necessary to make this filter work, and then I'll illustrate ways to use the filter.

Compared to the analog version, the digital state variable filter is also easy to design. Actually, it is even easier to design and implement than the analog version. It's efficient, especially at the lower frequencies where other filters will begin to fail, and requires little math for what it does. You need only to tell it the cutoff frequency (this must be less that 1/3 the sampling rate) and the *Q*, or, inversely, the damping.

Q is a measure of resonance—the higher the *Q*, the narrower the bandwidth at a particular pole location. On the *s*-plane, the closer a pole is to the imaginary axis, the higher its *Q*. In digital terms, the higher the *Q*, the closer the pole is to the unit circle. For those mechanical types, *Q* is a measure of the spring tension used in the spring-mass model of the differential equation.

A digital state variable filter is easy to design, tune, and stabilize. With it, you can create as many as four possible outputs at once.

As a side note, *Q* is one of the main points of difference among the standard filter types. For the Butterworth filter, the *Q* is $1/\sqrt{2}$. This produces the nice round transition point at cutoff frequency. A higher *Q* will result in a bump at that point (that becomes a peak if the *Q* is increased further). For parametric filters and parametric *EQ* systems, *Q* is the component that you manipulate to vary the peaks and valleys in the spectrum. In the routine below, I am actually using the inverse of the *Q*, known as the *damping factor*.

The other factor that you must supply is the proportion of the sampling frequency you desire as a cutoff, or corner frequency. Normally, this would be a simple matter of dividing the cutoff frequency by the sampling frequency (*fs/fc*) but since there are some nonlinearities involved at the higher frequencies, we use:

$$a = 2 * \sin\left(\pi * \frac{fs}{fc}\right)$$

We will call this the *cutoff_ frequency*.

Because of its ease, and these two tuning controls, the state variable filter is popular for very low frequencies and very high *Q* ($Q \geq 50$). You will quickly realize that because of the simplicity of this filter, you can create still other outputs and effects by adding poles, or zeros, in well-chosen locations. And, since the notch output is the sum of the lowpass and highpass outputs, it's simple to make a tunable output by using a coefficient that is less than one for one (or both) of the components. In this way, providing a shelf as well as controllable corner frequency for your filter is possible.

In pseudocode, we may describe this filter as:

```
lowpass_out = last_lowpassout + (last_
bandpassout * cutoff_ frequency)
lp(n) = lp(n-1) + (a * bp(n-1));

highpass_out = input-lowpass_out-
(damping_factor * last_bandpassout)
```

FIGURE 1 The lowpass output alone with a corner frequency of 100Hz and Q of three

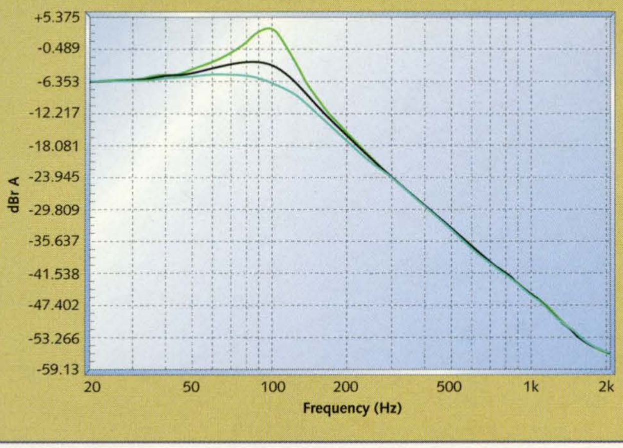
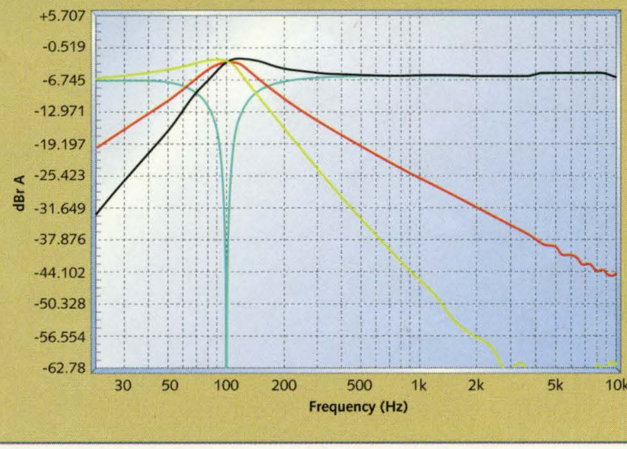


FIGURE 2 A plot of all the outputs of the state variable filter with a Q of three



```
hp(n) = x(n) - lp(n) - (k * bp(n-1));
```

bandpass_out = (cutoff_ frequency *highpass_out) + last_bandpassout

```
bp(n) = (a * hp(n)) + bp(n-1);
```

notch_out = lowpass_out + highpass_out

```
ntch(n) = lp(n) + hp(n);
```

Actual code

All the plots were created using the DSP code in Listing 2 (all the code for this article can be found on the *ESP* Web site at *www.embedded.com/code.htm*) and an Audio Precision audio spectrum analyzer. If you don't have access to this equipment, but you do have Matlab, you should get comparable plots using the code in Listing 1.

Listing 1 shows a Matlab M-file that you can use to test this filter. You may use it to generate some of the plots and some of the filter configurations that we will talk about in a moment.

If you prefer, Listing 2 shows the basic code implemented for DSP56002. Embedded within the last few lines are the changes necessary to produce the other examples I'll mention later.

This code was not optimized. Instead, I opted for clarity and easy adaptability. I encourage you to play with and improve it.

How can we use this filter?

Obviously, you can generate the necessary basic functions that you use so often in any application: lowpass, highpass, bandpass, and notch. But because of the controls this filter uses, you can do some things here that you may not have wanted to try with other forms of second-order filters, such as the Biquad.

With the state variable filter, we have direct control of the corner frequency and *Q* of the resulting filter, without resorting to complex calculations or tables. This means that you can easily create tracking filters, parametric filters, and sliding and shelving filters that work in real time without a lot of fuss. First, let's just examine the basic operation.

Figure 1 is a plot of the lowpass output. Here the sampling rate was 48kHz, the cut-off frequency was 100Hz, and the Q is three. I opted to make the *Q* and cutoff frequency memory variables so that they would be easy to adjust on the fly.

Figure 2 is a plot of all the possible outputs from the filter. The frequency is still 100Hz and the *Q* is still three. Please note that this is a relatively high *Q* and may not be suitable for all applications. As I mentioned, a Butterworth filter would have a *Q* of $1/\sqrt{2}$ which produces the smooth turnover point that is so familiar. Varying the *Q* on a lowpass or highpass filter will vary the smoothness of the turnover point, while varying the *Q* on a bandpass filter will narrow the bandwidth and increase the gain at the peak.

Parametric filters

By definition, a parametric filter is one that allows you to control every aspect of its construction. In audio applications, this often means control over gain, *Q*, and frequency. There are several reasons to want such control. This is useful for correcting the output of your equipment for aspects of the environment that are not as they were when the audio was recorded. Two aspects of this environment are the room and speakers (or headphones). Spectral charts are available for many of the speakers on the market today, so that the poles and zeros of these transducers can be corrected for in the amplifiers. In addition, the room will contribute its own resonant points and points of attenuation that may need to be overcome.

Parametric filters with adjustable Q are built into many crossovers, allowing the user to adjust for these things. A two-way crossover will have a highpass filter and a lowpass filter whose skirts meet at 90 degree phase shift, making this state variable filter a viable candidate for this purpose. The crossover will also have a Q adjustment to allow some frequencies to peak in each band.

In this case, you may use the filter as is, adjusting the corner frequencies and Q to match. If other peaks are needed, additional state variable filters could be created using the bandpass output for frequency and Q for peaking.

Parametric equalizers are similar, but have many more bands. If you prefer to make an IIR equalizer, the state variable may serve you as well with its bandpass or notch outputs. Peaks and valleys in the spectrum may be created by varying the Q and summing with the bandpass outputs. Figure 3 illustrates what the bandpass output would look like with varying Q.

FIGURE 3 The bandpass output with a Q varying between 20 and 1

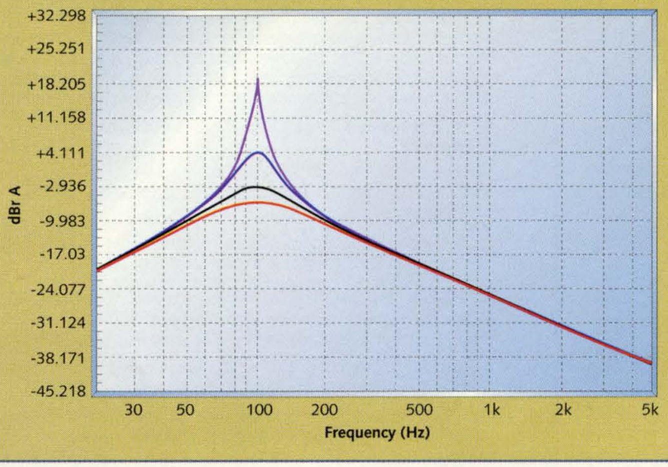
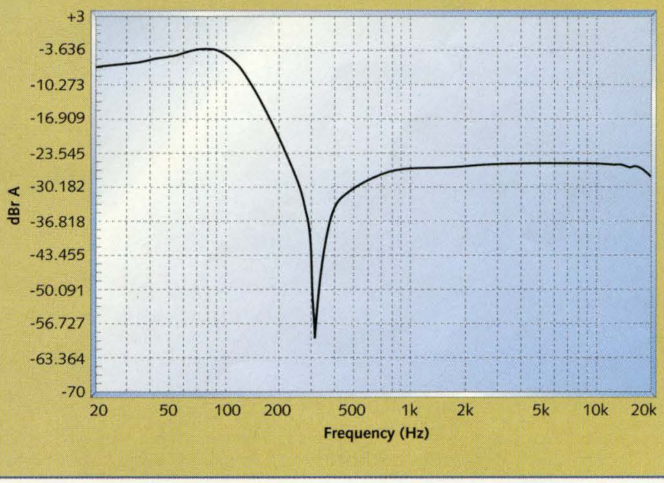


FIGURE 4 Shelving filter using the state variable filter



Shelving/sliding filters

Many forms of the sliding and shelving filters exist. I have described many of them in this column.

One of the most popular applications for this filter mechanism is in noise suppression. The sliding filter uses a control to detect where the actual (desired) audio is and it moves the corner frequency to a point just outside that part of the spectrum. This may be a highpass or lowpass filter. Anything outside that portion of the spectrum is attenuated.

A shelving filter is similar to the sliding filter except that it has an adjustable stopband. That is, the stopband may be adjusted up or down in level, to suit. In audio applications, the shelf is usually quite deep for high levels and shallow for low levels.

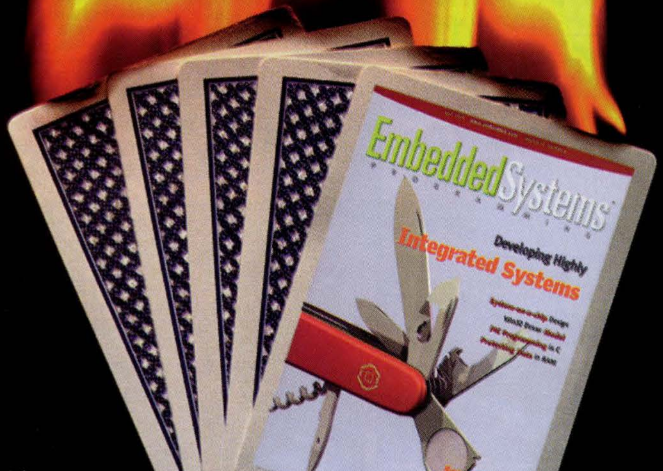
This is the first time I present a sliding/shelving filter with a Q adjustable on the fly. We will make this filter from the notch output simply by adding one more variable: shelf. You can see it in the DSP code in Listing 2. The variable is used as a coefficient for either the highpass portion or lowpass portion of the sum. It is typically less than one.

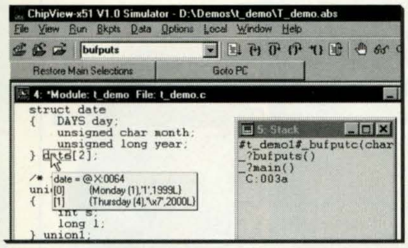
Thus, we multiply either the highpass input or lowpass input by this value and perform the sum as usual. Figure 4 shows an example output. In this case, the shelf was applied to the highpass portion. The result is not a perfect shelving filter due to the deep attenuation where the two filters meet.

Next month...

Next month we will explore some other simple but useful filter mechanisms. Following that, I'll start looking at how to make sound using digital signal processing. **esp**

Don Morgan is senior engineer at Ultra Stereo Labs and a consultant with 25 years experience in signal processing, embedded systems, hardware, and software. Morgan recently completed a book called Numerical Methods for DSP Systems in C *(M&T).*

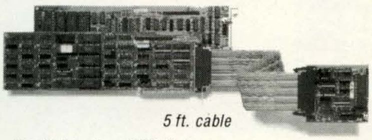

EmbeddedSystems
PROGRAMMING
Developing Highly
Integrated Systems
Special Reports

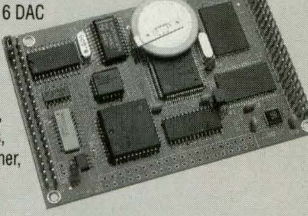

ChipView-x51 V1.0 Simulator - D:\Demos\t_demo\T_demo.abs
File View Run Bkpts Data Options Local Window Help
bufputs
Restore Main Selections
Goto PC
4: *Module: t_demo File: t_demo.c
5: Stack

```
struct date
{   DAYS day;
    unsigned char month;
    unsigned long year;
} date[2];

/*  date = @X:0064
uni [0]     {Monday (1),'1',1999L}
{   [1]     {Thursday (4),'\x7',2000L}
    int s;
    long l;
} union1;
```

```
#t_demo1#_bufputc(char
_?bufputs()
_?main()
 C:003a
```





ChipMax

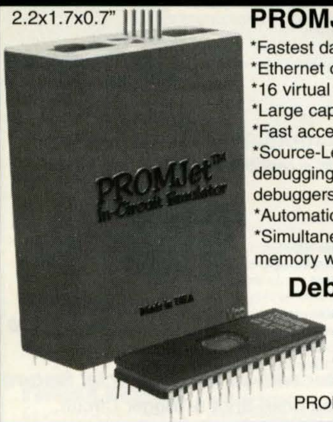
$345

- Programs over 1200 devices (E(E)PROM, Flash, Serial, PALCE, GAL, 875x/895x, 93Cxx, 17xx, PIC16xx)
- Fast parallel link to any PC, even laptops
- 40-pin universal pin driver and current limit
- On-board processor and built-in power supply
- Unbeatable programming speed
- Checks for incorrect device insertion
- Automatic EPROM ID search
- Supports WIN95/98/NT
- NO fans & modules are required in circuit
- Made in USA, Lifetime free software
- Visit web site for demo software

EE Tools

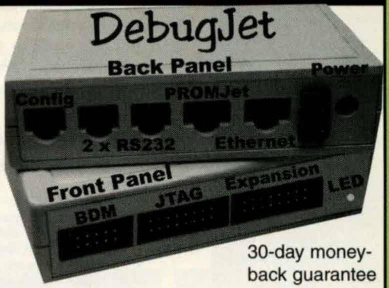
Sunnyvale, California, USA
Tel: 408•734•8184
Fax: 408•734•8185
www.eetools.com

PROMJet® DebugJet®



PROMJet

*Fastest data transfer 1.2~4Mbit/S.
*Ethernet or Parallel connection.
*16 virtual COM channels.
*Large capacity (up to 32Mb).
*Fast access time (up to 25ns).
*Source-Level and RTOS task-aware debugging with industry leading debuggers.
*Automatic support for 2-5V targets.
*Simultaneous access to emulation memory while target is running.



30-day money-back guarantee

DebugJet

*Supports standard JTAG protocol for testing and debugging.
*Supports BDM protocol for CPU32 and Power PC debugging.
*Optional TraceJet for hardware tracing and Breakpoint support.
*Target network connectivity without ethernet on target by using PROMJet.

EmuTec Inc. Tel: 425.267.9604 Fax: 425.267.9507
Web:www.emutec.com Email:emutec@emutec.com

PromICE with Trace

The Leader in Memory Emulation

- Trace to pinpoint startup problems and isolate real-time bugs.
- Code Coverage to verify execution and speed up QA.
- Ultra-fast downloads via Ethernet, parallel and serial ports for Unix, Windows 95/NT and DOS.

Grammar Engine Inc.
Call Toll Free:
1-800-776-6423
www.gei.com

It's Small, but Don't Let the Half-Size Fool You

New half-size CPU boards from MCSI are workhorses. Wether you are looking for a 386 or Pentium/MMX, MCSI has what you need. Don't take our word for it. See for yourself.

- Customer service
- Engineering services
- Evaluation program
- Competitive pricing

MCSI
"The Embedded PC Specialists"

Call Now 888 232-7826

MCSI is an ISO 9001 certified company.
www.mcsi1.com • e-mail: mcsi@mcsi1.com

SB-56K Multi-DSP Emulator

Support for the Motorola DSPs: DSP560xx, DSP563xx, DSP566xx, DSP568xx

SB-56K supports any combination and any count (up to 255) of the devices from the above families. With its accurate counter allowing to measure code execution (benchmarking), small size (1"x2.5"x4"), high speed RS-232 interface, the SB-56K can provide independent support for multiple devices with option to access each device on the target board from different workstations connected through LAN, WAN or Internet.

Domain Technologies, Inc.
1700 Alma Dr.,#495, Plano, TX 75075
Tel.:(972) 578-1121 / Fax: (972) 578-1086
info@domaintec.com
www.domaintec.com

SX Debugger+Programmer

Complete System **$375**
FREE
Demo/Prototype board

All tools **qualified** by Scenix Semiconductor

SX-ISD

• Supports SX18/20/28/48/52 • In-circuit run-time debugging • Real-time code execution • Source level debugging • Built-in programmer • Real-time breakpoint • Conditional animation break • External break input • Frequency synthesizer • Selectable internal frequency • External oscillator support • Software trace • Unlimited watch variables • Parallel Port Interface • Runs under Win 95/98/NT • Comes with SASM Assembler

Single, Gang Programmers and SMT adapters are also available

Advanced TransdAtA Corporation
Dallas, Texas
Tel 972.980.2960 **www.adv-transdata.com**

PIC Real-time 'Emulator'

$159

PIC-ICD

PIC-ICD = Debugger + Programmer + DemoBoard

• Designed for 16F87X • Can also support most 16C6X/7X • In-circuit run-time debugging • Real-time code execution • 2.5-6V operating voltage • Source level debugging • Built-in programmer • Real-time breakpoint • Conditional animation break • 2 External break inputs • Selectable internal frequency • Software trace • Unlimited watch variables • Parallel Port Interface • Runs under PICICD IDE (Win95/98/NT) or MPLAB (Win95/98)

Advanced TransdAtA Corporation
Dallas, Texas
Tel 972.980.2960 **www.adv-transdata.com**

PIC16/17 Emulator

New Low Prices

Complete system starts at **$599**

RICE17A

• For PIC12/16/17 • 3-5volt emulation • 64K program memory • 32K real-time trace • 12-clip external probe • Source level debugging • External break input • Trigger and break output • Real-time breakpoint • Unlimited software break and trigger points • Selectable internal frequency • Unlimited watch variables • Parallel Port Interface • Runs under Win95/98/NT

Probes for **16F87x** and **16C77x** by Jan 2000 With data break support

Advanced TransdAtA Corporation
Dallas, Texas
Tel 972.980.2960 **www.adv-transdata.com**

TechTools™ Embedded Systems Development Tools

TechTools ™ ROM Tools

Save Development Time by using a TechTools Emulator in your memory Socket. Modify, download and test code in a matter of seconds. Become even more productive with the Advanced features only available from TechTools.

FREE CD!

Easy to use, Economical EPROM Emulator.

High-speed, Reliable, EPROM Emulation at a reasonable price.

- Fast Download (i.e. 512Kbit file in 1.5 seconds or less).
- Fast 90ns & 45ns access times.
- Read-back, Verify and Self-test Functions.
- Full-screen editor, QuickLoader, batchable loader & utilities included.
- All sizes and speeds can be Daisy-chained together and individually addressed from one LPT port.
- Memory retention for true power-up emulation.
- Low voltage Target Adapter with Ext. Power supply available.
- 8 bit & 16 bit Models available up to 4 Mbit.

*Econo*ROM™ *II*
From $149.00

Ultra FAST FLASH and EPROM Emulator with powerful special features:

EPROM and FLASH Emulator with Address Snap-Shot & Trigger Circuit.

*Flex*ROM™ *II*
From $349.00

- Target Write-back. • Arbitration support. • Fast 90ns & 45ns access times.
- Ultra Fast Download (up to 2.5 Mbit per second).
- Daisy-chain port for multi-unit operation.
- Full-screen editor, QuickLoader, batchable loader & utilities included.
- C Library & DLL • On-The-Fly editing.
- Snap-Shot circuitry captures the target's most recent access.
- Trigger circuitry generates a trigger each time the target accesses a user-specified address.
- Low voltage Target Adapter with Ext. Power supply available.
- 8 bit & 16 bit Models available up to 8 Mbit.

Advanced Memory Emulator with No-Impact, 'LIVE' Access.

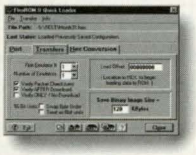
"**LIVE**" Editor Included

UniROM not only emulates EPROM, Flash and SRAM, but also adds hardware assisted debugging, Live editing, Live Watches and a robust library for custom applications. UniROM is the only Memory Emulator that allows real-time 'Live' editing and monitoring with zero impact on the target system.

From $595.00

*Uni*ROM™

QuickLoader™ Software



Win95/NT QuickLoader Software provides a quick "1 Click" method of converting and downloading files to the emulator from the Windows Environment.

(Included with all FlexROM II and EconoROM II Emulators.)

TechTools

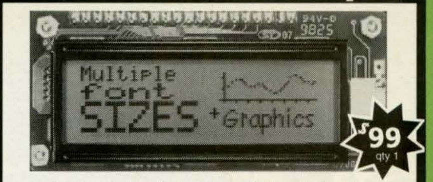
Email Request:
sales@tech-tools.com
Telephone: 972-272-9392
Fax: 972-494-5814

www.tech-tools.com

Copyright © 1998, TechTools, P.O. Box 462101, Garland, Texas 75046-2101 • EconoROM, FlexROM, UniROM, QuickLoader, the "Wizard" symbol and TechTools are trademarks of TechTools, P.O. Box 462101 Garland, Texas 75046.
• All other trademarks are trademarks or registered trademarks of their respective company.

Serial in, graphics out.

Almost *too* easy.



G12032 Serial Graphics Display
120x32-pixel LCD w/backlight

Jump-start your design project with our easy-to-use serial graphics LCDs. Works like a tiny terminal at 2400 or 9600 bps. Stores custom fonts, bitmap screens in EEPROM. Order today, show a product with a graphics display tomorrow!

www.seetron.com
Scott Edwards Electronics, Inc.
phone: 520-459-4802 fax: 520-459-0623

68HC12

Real-Time Debugging

- DG128, DA128, D60, BC32, B32, A4 etc...
- Flash and EEPROM programming
- Page mode debugging
- Hardware breakpoints
- Hot plug
- Cosmic, Hiware, IAR, Introl support

Noral Micrologics, Inc.
The Debugging Company

Call toll free:
888-88-DEBUG
for free evaluation

Tel: (508) 647-0103
Fax: (508) 653-1828
harrye@tiac.net
http://www.noral.com

noral
The Debugging Company

JUMP on the NET!

The µFlashTCP gives you 10BASE-T Ethernet connectivity, a full-function TCP/IP stack and 2 serial ports in a package 30% smaller than PC/104 solutions.

Field-proven TCP/IP stack, DOS and PC-compatible BIOS make development quick and easy.

Prices start at $169 qty 100.
Development kits are available.

Call 530-297-6073 Fax 530-297-6074
Check our web site for the latest updates
www.jkmicro.com/uflash

JK microsystems

From the Author of WATTCP

eRTOS

DOS Realtime Kernel with TCP/IP Support

- ***Preemptive & Cooperative threads***
- ***Critical Section Protection***
- ***Interthread Messaging***
- ***Complete Re-entrant TCP/IP***
- ***Web, CGI, FTP, Email, Telnet***
- ***Web Graphics***
- ***Interrupt-driven Serial Support***

www.**ertos**.com
Call 530-297-6073 Fax 530-297-6074

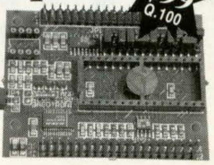
JK microsystems

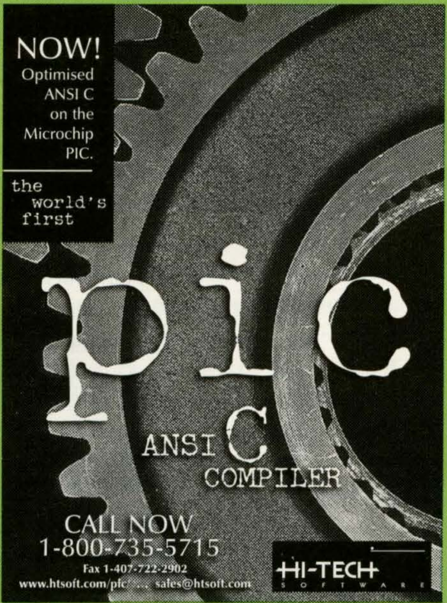
Vesta Technology, Inc.
Single Board Computers for the 21st Century

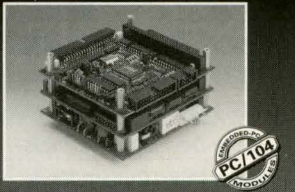
SBC2000-332

Vesta BASIC
or C Software
32 bit 68332
Low Power
2 x RS-232
Watchdog
Real Time Clock
E²PROM
LCD/Keypad
1 Meg FLASH
1 Meg ROM
1 Meg RAM
$299 @ 1
$188 @ 100

(303) 422-8088
www.sbc2000.com







ADVERTISER INDEX

This page is provided as a service to readers. The publisher does not assume any liability for errors.

Jack G. Ganssle

Cores, Cards, and Tubes

Our author traces the predecessors of today's embedded systems back to his long-haired, havoc-wreaking, hippie days.

Last month I reminisced about the early days of the microprocessor, about building embedded systems when edit/assemble/link times were measured in days rather than seconds.

But embedded systems are just the latest incarnation of electronic digital computing, a field that arguably goes back for more than half a century. Long before 32-bit CPUs sporting millions of transistors crammed into a chip the size of a fingernail existed, a huge computer industry built on much less friendly technology thrived.

The Univac 1108 I grew up on was a mainframe of gigantic proportions: the computer room covered a good quarter acre, holding the CPU itself plus disks, drum memory, tape drives, and printers.

Drum memory? A minivan-sized box contained two counter-rotating six-foot-long drums, each coated with ferromagnetic material. Hundreds of recording heads darted across the surface, allowing this beast to store a few tens of megabytes of data. Head crashes were so common that the school used two redundant drum subsystems, one operating, and one usually under repair. In fact, two Univac servicemen worked full-time in this computer room, in a reasonably successful effort to keep the machine mostly running.

Tape drives? Sixties-era sci-fi movies always used a bank of whirring tape drives to indicate the ultimate in high-tech. A dozen refrigerator-sized tape drives fed information into our 1108. When drums and disks were measured in tens of megs, tape was essentially the only form of personal data storage. Each reel held 50Mbits—a vast amount of data in those days—and were stored in a tape library just off the computer room. When you needed your data, you'd send a message to the operator who located and loaded the proper spool. Needless to say, access times were measured in minutes to hours.

Our 1108 was a dual-processor model, with one unit dedicated to managing all of these disks, drums, tape, and printers. No peripheral had built-in computers in those days. The other CPU ran users' programs as well as the OS. Each processor was surprisingly small considering the vast enterprise surrounding them, being about the size of two fridges.

Open the back door of a CPU and you'd be confronted by an ocean of blue wires. Univacs had hundreds of circuit boards plugged into a backplane; all of the boards were interconnected by wirewrap wires. Need to change an interrupt level on a peripheral? Get out the wire wrap gun and start making mods.

These were 36-bit machines. Their peculiar wordsize came from Sperry's adoption of a pre-ASCII notation called Fielddata, which used six bits to store letters, numbers, and punctuation. Printers produced only uppercase output, so the six-bit (64 symbols) limit wasn't much of a hindrance.

Input devices were limited to a bank of teletypes and punched cards—both also restricted to uppercase. I talked about teletypes in the context of microprocessors last month. In the mainframe world the overwhelming aspect of these beasts was their noise. A jet engine couldn't compete for attention in a roomful of chattering teletypes. Unless, of course, you were foolish enough to hack into the accounting system and take over the entire computer, in which case all of the other machines fell silent while just your teletype continued to chatter, with 50 pairs of eyes staring accusingly at you. But that's another story.

Most mainframe computing used punched cards to store code and data. The programmer would tediously—oh so tediously!—enter the program on a card punch, storing one line of Fortran or Algol per card. Two thousand cards filled a card box. Big programs might take a half dozen or more boxes of cards, quite a tower of disaster when the poor student tripped, spilling his

The 1108 was completely transistorized. Here, in the first month of 2000, it's awfully hard to imagine that people once designed with individual transistors instead of clumps measured in millions.

precious intellectual efforts all over the wet parking lot.

After preparing the cards, the programmer carried the stack to the Computer Operator, a position of such power and glory it cannot be imagined today. These were the chosen few who had hands-on access to the machine. When the poor supplicant submitted a card deck, the Computer Operator grandly gave an estimated turn around time, typically 24 to 48 hours. After a day or more our humble programmer returned for his printout and the often mangled card deck. Imagine his frustration when stupid mistakes mandated another run, burning up another day or two as the project deadline drew ever nearer.

The 1108 was completely transistorized. Here, in the first month of 2000, it's awfully hard to imagine that people once designed with individual transistors instead of clumps measured in millions. Old timers will remember the first transistorized radios from Japan, with marketing campaigns loudly proclaiming "six transistor receiver!" Six transistors! Can you imagine building anything with so few active elements?

But transistors were an astonishing advance over the technology of the '40s to the '60s, that of vacuum tubes ("valves" to our European friends). Though tube-based computers predate all of my experiences, as a teenaged ham radio enthusiast I often built radios, transmitters, and "hi-fi" equipment using tubes. All computers from the Eniac until the late '50s used these devices exclusively.

The vacuum tube was actually accidentally invented by Edison, though he didn't understand the importance of his experiments and discarded the technology.

Fleming and others understood that a heated filament emitted a stream of electrons that could be regulated by inserting a grid of wires between the electron source and a receiving end. The quantum complexities that govern the behavior of today's devices were refreshingly absent on the macroscopic scale engineers used during the tube era. They would fire a stream of electrons from the filament towards the plate through a mesh "grid." A small negative charge on the grid would repel the negatively charged stream, effectively gating (or "valving") the flow, and thus creating an amplifier, logic element, or oscillator.

Compared to today's submicron geometries, the scale of these devices was vast. By the '60s, twin triodes were the big thing, units containing (count 'em) two active elements, equivalent to two transistors. The 12AX7, a twin triode of the era, was almost an inch in diameter and about two inches long, with eight leads sticking out the bottom. We could easily pack a hundred billion transistors into this form factor today.

Though we may feel smugly superior in that we now deal with logic blocks instead of individual transistors, in fact blocks were always a part of the electronics landscape. DEC created the "Flip-Chip," standard circuit boards that might each contain a flip-flop. They built the early minicomputers out of hundreds of these boards.

Before DEC, though, vacuum tube engineers, too, created logic blocks. I once had a large box of surplus logic elements. A single flip flop was a tower with a tube on top, electronics below, and under that a standard tube-type connector. It is possible, after all, to build a simple flip flop from only two active elements. Entire computers were built of thousands of these large (an inch and a quarter square and about five inches high), hot (because of the filaments), high-voltage (most tube circuits ran at about 300V to propel the electrons across a centimeter or so of vacuum) devices.

Perhaps one of the most profound differences between the digital and analog designs of the tube era and those of today is in the use of active elements. Tubes were expensive, power hungry, large beasts with relatively short lifetimes. All electronic design was an exercise in optimization. "Hi-fi" amplifiers typically had a few tubes. TV sets might use eight. Can you imagine building anything with less than a few thousand transistors today? The cafeteria-sized Eniac had 18,000, which, though a vast number, compares pitifully with today's transistor counts. How many hundreds of millions of transistors are in your 3-lb. laptop?

The first number in a tube designator (like the 12 in 12AX7) specified the filament voltage. The most common were 12.6V and 6.3V, which became the butt of a joke in the '70s when Signetics published an April 1 data sheet for their highly integrated Write Only Memory (on-line at *www.ganssle.com/misc/wom.html*). A 6.3V power supply was required, as a footnote said, "for the filaments," a joke lost on the post-tube generation.

Core memory

Before RAM, before EPROM, OTP, flash, EEPROM, or any of the other zero-cost, high-density memory arrays we now take for granted, computers stored data and programs in core.

As the microprocessor came of age, my engineering career was split between working on micro-based embedded systems and similar products that "embedded" minicomputers, either PDP-11s from DEC or Novas from Data General.

Throughout this period, core was the only random access read/write memory in common use. It wasn't till the very late '60s that even the smallest MOS memory chips became available.

Each core is a ferrite bead, perhaps the size of a small "o" on this page. Four wires run through the center of each core, four wires tediously strung, by hand, by poor workers who, no doubt, worked for a pittance.

Cores are tiny magnets, each remembering just one bit of information. The trick is to flip the magnetic field of the cores—one direction is a "one;" the opposite field indicates a "zero."

As we know from basic electromagnetics, a changing voltage creates a magnetic field, just as a change in a magnetic field induces a voltage. The wires running inside of the ferrite beads create the fields that flip the direction of magnetization, writing a zero or a one in the process. They also sense the magnetic field so the computer can read the stored data.

Two of the wires organize the core into an X-Y matrix. The core plane is an array of vertical and horizontal wires with a bead at each intersecting node. Run 50% of the power needed to flip a bit down each wire—at the intersection there's all that's needed to flip just that one bit. What a simple addressing scheme! As the bit changes state, it induces a positive or negative pulse in a third wire that runs through all of the cores in a plane. Sensitive amplifiers convert the positive or negative signal to a corresponding zero or one.

Since the amplifiers detect nothing unless the core changes state, reads are destructive. You've got to toggle the bit, and then write the data back in on each and every read cycle. It sounds terribly primitive until you think about the awful things we do to keep modern DRAM alive.

Before microprocessors quite caught on, the instrumentation company where I worked embedded Data General Nova minicomputers into products. The Nova used core arranged in a 32K x 16 array. The memory was nonvolatile, remembering even when no power is applied.

We regularly left the Nova's boot loader in a small section of core. My fingers are still callused from flipping those toggle switches tens of thousands of times, jamming the binary boot loader into core each time a program crashed so badly it overwrote these instructions.

For some reason these Nova memories suffered a variety of ills. Core was expensive—around $2,000 for 32,000 words, a lot of money in 1974 dollars. A local shop repaired damaged memory, somehow restringing cores as needed, and tuning the sense amplifiers and drive electronics.

As we worked through these reliability issues, my boss—who was the best digital designer I've ever met—told how some military and space projects actually employed core as logic devices. In a former job he designed systems composed of strands of core strung together in odd patterns to create computational elements.

One of my holy relics is a 3-lb., 13,000-bit core array acquired in 1971. A few days after my high school graduation I hitchhiked with a pal to Boston (those were kinder, gentler days) to find treasures in the disorganized depths of a surplus shop.

Was it our long hair? Maybe the fact that we were warned three times to get off the New Jersey Turnpike had something to do with it. Somehow Gary and I found ourselves in a New Jersey jail cell, busted for hitching. The police, expecting to find a stash of drugs in our backpacks, were surprised to discover instead my 13,000 bits of core.

The officers were unaware of the pending microprocessor revolution, and were equally disbelieving about my story about "a computer on a chip." They saw me as the vanguard of an invasion of commies bearing death-ray components.

"What's this?" the chief growled. I timidly tried to convince him it was computer memory. These were the days when computers cost millions and were tended by an army of white-robed technicians, not hitchhiking long-hairs. All of the cops looked dubious, but could find nothing to dispute my story. They eventually let us go, me still clutching the memory which today sits on my desk.

A few years later I experienced an eerie echo of this incident when I lived in a VW microbus. Coming back from Canada into a remote Maine town, the local constabulary, sure I was running contraband, stripped the van. Must have been the long hair. They found a 6501—the first low-cost microprocessor chip. MOS Technology amazed the electronics world when they released this part—the predecessor of the 6502 of Apple II fame—for only $20. I just had to have one, though I just tossed the part in the glove compartment. The officers were unaware of the pending microprocessor revolution, and were equally disbelieving about my story about "a computer on a chip." They saw me as the vanguard of an invasion of commies bearing death-ray components.

The computer industry is still quite as exciting as it was in those earlier days, though at least now the excitement never includes a view from within a jail cell. Well, at least not recently. **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.

P.J. Plauger

What Barely Works

The revised C standard has just recently been approved. Known informally as C9X, it managed to stay true to its name, just barely, by getting formal approval in late 1999. While it contains a number of improvements, C9X does not rock the boat. It remains highly compatible with the original ANSI C standard approved back in 1989, which was then adopted as an ISO standard in 1990. Some would argue, in fact, that the revision failed to address areas that are known to be weak in C89. Having been a party to the original standardization effort, as well as much of the revision process, I mostly disagree.

That original ANSI standard for C perforce contains compromises. Many of these balance the conflicting needs of users and implementors. Some balance conflicts between different classes of users, some between different styles of implementation. Whatever the driving force, some of the compromises are better than others. Admittedly, some of the compromises are less successful than others. Some machinery barely works, or is easily misused. But for those cases in particular, knowing how the compromise came about and why the language was not made better is worthwhile, if only to minimize problems that can arise as a result of the compromise.

Users care about the weak spots in C because they need to avoid them. There is no point in stressing an already weak part of the language. Implementors care about the weak spots because they need to craft carefully around them. There is little point in trying to make them too sound, however. A program that depends upon a good implementation of a weak feature will only break when moved to another system.

I know of nothing more complex, or harder to standardize, than a programming language. A general purpose language has a broad range of uses. Each use requires an extremely precise description of the language. Each use has a constituency with strongly held views about how the language should behave. It is hard to be at once ecumenical, precise, and accommodating for diverse applications.

One of the least understood aspects of making a programming language standard is the need for compromise. A user with a limited viewpoint sees only that the language is weakened for his or her particular application. How to fix it is patently clear. Should someone else object to the fix, it must be because that person has only a limited viewpoint.

The hardest to defend are the compromises that accommodate implementors at the expense of users. In a competitive marketplace, users are accustomed to making outrageous demands and having them met. A standard cannot be so brazen.

It's one thing to expect all PC compilers, say, to provide a certain service. Chasing a market measured in tens of millions of machines is a real incentive for implementors to work late into the night. To demand that a compiler for an eight-bit embedded microcontroller offer the same service, however, is quite another thing. Make too many silly demands and you encourage nonstandard subsets. Or you starve an already lean market.

Conflicts can occur between different implementors as well, or between different users. In all cases, the standard must compromise as best it can. If it cannot compromise creatively, it can only weaken the language. Where the language is weak, all users must beware. The chance of writing erroneous or nonportable code is high. Implementors, too, must step cautiously in these areas. It is too easy to step outside the bounds of conformance when the bounds are fuzzy.

Making a language both powerful and portable at the same time is difficult. C succeeds quite well in many areas, but sometimes just barely.

For all its successes, the latest C standard still contains a generous helping of unfortunate compromises. My goal in this essay is to explore a number of the significant weak spots. I attempt to explain why they are weak, how they escaped better fixes, and why you should tread lightly around them.

Pointers and addresses

Computers vary widely in how they address storage. It is remarkable that

C permits the aggressive manipulation of addresses, yet remains efficient and portable. Nevertheless, here is an area that is fraught with dangers. Until fairly recently, I might have labelled pointer arithmetic in C an area of weak compromise. The C that grew up on the PDP-11 and VAX was facing serious portability problems:

- Early C assumed that pointers had the same representation as type `int`. More and more machines had pointers that are incommensurate with integers. You had to be much more careful about declaring function arguments and return values properly
- Early C assumed further that pointers looked like unsigned integers that counted bytes in storage. More and more machines had pointers that are composites of two or more fields. You had to be much more careful about converting between pointer and integer representations

Fortunately, many C programmers were willing to program more carefully. In the area of pointers, they got sophisticated in a hurry. Perhaps it was the surge in popularity of the IBM PC, with its bizarre 8086 memory addressing. Perhaps it was the need to port Unix-based tools to multiple platforms to reach an adequate market. Whatever the reason, C programmers soon left behind their wild and wooly ways with pointers.

The situation has almost reversed itself. C started out on a machine with 16-bit pointers and 16-bit ints. That remains a valid implementation, but it has become an uncommon one. Today, a 16-bit computer is more likely to be found in a rather small embedded application. Serious computers for applications programming, or even many embedded systems, now need larger pointers, to address millions of bytes of memory. Try to move an application to such a small computer and it will likely break in a dozen places.

If the purpose of standards conformance is to maximize portability, then here is arguably a weak spot in the standard. What's the use of promising some form of portability if the promise isn't kept?

The question is, of course, rhetorical. Small embedded systems and large applications represent diverse user communities. Both want to use Standard C. Both want to buy C translators with the cachet of standards conformance. Both deserve to have their interests represented in the C standard.

If you're looking for portability across implementations, however, it's not enough simply to ask for conformance to the C standard. You must also look for implementations from the same subculture as the original host. It would be nice if a standard could define these subcultures with some degree of precision. The C standards committee tried in some areas, and mostly failed. In other areas, the committee didn't even try.

If you accept that pointers have inscrutable representations, your code will be largely portable. If you accept that a machine can simply be too small to host certain applications, you will be disappointed far less often. That leaves only one area where I feel that the standard still contains a weak compromise on pointers.

If you subtract two pointers, you get a signed integer result. The result counts the number of bytes between the places designated by the pointers. Naturally, both pointers had better point within the same data object. They should, in fact, designate elements from the same array. (Remember that any data object can be treated as an array of some character type.) The result can be positive, zero, or negative.

The result can also overflow, at least on some machines. Overflow is most likely to occur on 16-bit computers, since they are cramped to begin with. It can occur on a machine of any size, however. All the implementor has to do is choose the type of `ptrdiff_t` properly (or improperly).

If you can declare a data object sufficiently large, you can generate an overflow when subtracting two pointers. Say, for example, that `ptrdiff_t` is a synonym for type `int` and `size_t` is a synonym for type `unsigned int`. If the implementation lets you declare an object with more than `INT_MAX` bytes, you can get in trouble. The size of the object is well defined, but not the difference between any two pointers within it.

Nothing prevents an implementation from choosing a larger signed type for `ptrdiff_t`. Assuming one exists, that is. And assuming that old C code doesn't break because it assumes a certain type for the difference between two pointers. Sadly, at least one of these assumptions often fails.

In the days when 16-bit computers were more common, I habitually tested code for this class of failures. Contrive a data object that fills more than half of memory and something invariably broke. It was a rare programmer who was always alert to this weak spot in C.

This problem has really come to haunt us in recent years. We already have more and more applications that strain the bounds of memory even on 32-bit machines. We have 64-bit integers and truly interesting combinations of sizes for the types `int`, `long`, `long long`, and `size_t`. But at the same time, programmers who are alert to the dangers of large objects are probably becoming more rare.

Integer sizes and shapes

If machines with small pointers are becoming rare, machines with small `int`s persist. When C89 was first approved, the most popular chips, such as the Intel 8086 and the Motorola 68000, favored 16-bit integers. They could work with 32-bit integers, to be sure, but at a penalty. The penalty is always in time and frequently in code size.

Here is one of the most interesting compromises in the C standard. The language has recognized from the outset that each computer has a preferred computational type. With its 16-bit roots, C has always tolerated `int`s as small as 16 bits. The C committee decreed, after only a brief debate, that `int` should never be smaller. It can certainly get much larger.

Today, great gobs of C code originate on machines with 32-bit `int`s. I would be astonished if many of those gobs survive porting to machines with 16-bit `int`s, at least not without some careful attention and a few judicious changes. (Porting to the newer architectures with 64-bit `int`s is still another interesting problem, though generally less challenging.) You could argue that portability is not well served by tolerating machines with `int`s smaller than 16 bits.

I was one of the first people to push the limits of portability by writing commercial software in C. My goal was to write as much code as possible to run unchanged on half a dozen different architectures. Those architectures ranged from eight-bit Intel 8080s to 32-bit IBM System/370s. Certainly at the time—the mid 1980s—no other enterprise was nearly that ambitious.

One of my earliest discoveries about C was most ironic. I found that the type `int` was the greatest impediment to portability. Wherever it appeared in code was a likely source of difficulty in moving code among machines. It was so bad, I learned never to declare a data object with type `int`. I also learned to contain places where the type crept into expressions willy nilly.

The irony, of course, is that `int` is a pervasive type. It is the type to which character and short integers gravitate when they appear in expressions. It pops up whenever you write a Boolean expression. Until C9X finally disallowed implicit typing, it has been the default type when you fail to mention an explicit type.

The elasticity of type `int` is one of those fundamental aspects of C that is both a strength and a weakness. Many rules in C help an implementation generate good code by adapting to the peculiarities of each machine. But those same rules put an extra burden on programmers. Unless you learn to be wary of elastic types, you can easily sacrifice much of the portability that supposedly comes free with using C.

Integers can also be elastic in another dimension. The C standard tries to be ecumenical about how an implementation encodes integers. It doesn't allow every form that has ever been wired into hardware, but it does allow a certain variety. My guess is that many C programs will fail if moved to some of these variations.

The one thing you can depend on is that integers have a weighted binary encoding. That means that a positive integer always has a least significant bit that contributes zero or one to the value. It has a next least significant bit that adds zero or two, and so on. The value zero always has all bits zero.

Not all binary encodings are weighted, by the way. Gray code has the peculiar property that adjacent values have codes that differ in one bit position. N bits can represent 2^N distinct values, as usual, but not in the usual way. C does not permit Gray code for representing integer types.

Where the variety comes is in representing negative values. I know of three encodings. They are called twos complement, ones complement, and signed magnitude. In all cases, a negative number has its most significant bit set.

Twos complement gives the most significant of N bits the value $-2^{(N-1)}$. An unsigned integer would give this bit the value $+2^{(N-1)}$. To negate a number, complement all the bits and add one. (Ignore a resultant carry off the end.) Only the value $-2^{(N-1)}$ overflows when negated.

Ones complement is similar to twos complement, with negative values differing slightly. To negate a number, simply complement all the bits. No values overflow when negated, but zero comes in two flavors (all one bits and all zero bits).

Signed magnitude multiplies the value by -1 if the most significant bit is set. To negate a number, simply complement the sign bit. As with ones complement, you avoid overflow on negation at the cost of having a distinct code for -0.

The vast majority of modern computers use twos complement arithmetic, for integers at least. Floating-point numbers typically use signed magnitude, but that is much less of an issue. Floating-point operations are more intricate, and more stylized, than integer operations. Even -0 causes few floating-point surprises, provided comparison operators work sensibly.

The C committee decided to permit all three encodings for integer arithmetic. Twos complement is expensive to simulate if a computer doesn't do it naturally. And there are some important processors that can host C well in all other respects. It would be unacceptable to hamstring them, or to rule them out, on the matter of arithmetic alone.

Nevertheless, you can expect numerous and subtle surprises moving code to a machine that does not use twos complement arithmetic for integers. Some can come, as with floating point, if an implementation is shoddy about its treatment of -0. Most, however, come from the bitwise operators.

Most programmers learn the power of bitwise operations sooner or later. They let you manipulate integers as individual bits or groups of bits. You use bitwise `AND` to mask or clear specific bits. You use bitwise `OR` to merge or set specific bits. You use bitwise `XOR` to complement specific bits.

Stick with positive values for all operands and you're in good shape. Remember that C requires a weighted binary representation for all integers. That means zero and positive values always have the same bit patterns. Your only uncertainty is the number of bits. Even there, you have rigorous lower bounds.

Once an operand gets its sign bit set, however, you are asking for trouble. Mix it with an unsigned operand and it may well get converted to unsigned itself before the bitwise operation is performed. Twos complement values suffer no change of bit pattern. The other two forms get curdled in various ways. The low-order bits may not be what you expect.

Smart programmers have learned the dangers of right-shifting signed operands containing negative values. Implementations have altogether too much latitude in deciding what to do, even with the low-order bits. But even smart programmers tend to be cavalier about bitwise operations. Expect trouble if you have to move code away from a twos complement implementation.

Messing with the stack

Another notoriously spongy area concerns how arguments get passed on a function call. Most of the time, you don't really care. It's only when you need to advance a pointer from one argument to the next that storage layout matters. Even here, you should only care when you write code that processes a varying number of arguments. The C standard encourages a fairly disciplined approach, but even this approach has plenty of soft spots.

In the earliest days of C, this was a non-problem. Everyone knew how the PDP-11 compiler passed arguments. It laid them out in a solid brick on the stack. Successive arguments occupied increasing addresses. Arguments had only a handdful of distinct representations—a character type became an `int`, for instance, and a `float` became a `double`. Learning the possibilities and writing code that walked any argument list was easy.

As C moved to other environments, various problems arose. In some cases, the implementor chose to match an existing calling sequence that let you mix C code and, say, Fortran, to advantage. In other cases, the machine itself was unsympathetic to the PDP-11 way of doing things.

Perhaps it was easier to put arguments on the stack in reverse order. Or perhaps it was necessary to leave various size holes between arguments. Or arguments got scattered through registers. In some cases, arguments had to be accessed through a chain of pointers.

Several of us C pioneers settled on the obvious solution very early on. We each introduced a set of macros for accessing arguments and stepping through argument lists. Naturally, the various solutions differed in detail. But they were similar enough in spirit to make it possible to port code among the different schemes.

The C committee was able to pick and choose among the best aspects of the various approaches. Since the Berkeley `<varargs.h>` was best known, it served as the basic prior art. We

introduced enough changes to warrant renaming the standard header `<stdarg.h>`. That is the machinery you should of course use today for walking varying length argument lists in Standard C.

The approach is basically flawed, however. Few operations require a more intimate knowledge of implementation details than walking an argument list. Prior art let you do so with macros, and the C committee was committed to preserving prior art wherever possible. But that prior art barely worked.

Consider: you have to prime the macros with the address of the rightmost declared parameter in the function definition. Some implementations require this information to determine where the varying arguments begin. That rules out functions with no fixed arguments. A perfectly sensible possibility is ruled out by an implementation kludge.

You also have to know the type of an argument after it has suffered default conversions, in the absence of function prototype information. That's not impossible, but it is often tedious and occasionally tricky. Thanks to the value-preserving rules, for example, an argument of type `unsigned short` might become either type `int` or type `unsigned int`. If you think that's of no consequence, go reread the previous section.

Finally, you are constrained as to the types of arguments you can pass. Put simply, you must be able to write an asterisk after the type specifier for an argument to make the appropriate pointer type specifier. That is not always possible.

In summary, you had better not be ambitious in your use of the `<stdarg.h>` macros. Keep it simple or expect portability problems.

The last item I will discuss is probably the biggest kludge in Standard C. It is the machinery for performing nonlocal jumps. These are control transfers that do violence to the normal stack discipline of nested function calls and returns. They are implemented by the function macro `setjmp` and the function `longjmp`, defined in the standard header `<setjmp.h>`.

A call to `setjmp` memorizes information about the current function call environment in a data object provided by the caller. A later call to `longjmp` whangs the calling environment back to that memorized in the data object. The net effect is that control once again returns from `setjmp`. Any intervening function calls are forgotten.

It is very hard for a C translator to optimize much in the presence of calls to `setjmp` and `longjmp`. The last thing an implementor wants to worry about is having the stack wrenched about in the middle of a block of code. Nevertheless, that is exactly what happens.

As a sop to implementors, the C standard allows a scary bit of uncertainty. It leaves open what happens to dynamic data objects declared in a scope containing a `setjmp` call. The environment might snapshot one of these data objects when `setjmp` is called. In that case, the value gets rolled back by `longjmp`. Or a data object might be left out of the environment. In that case, the value remains unchanged.

You'd think that you could devise a simple rule for determining what happens. Register data objects are part of the saved environment, auto data objects are not. Sounds good, but it doesn't work. The translator may choose not to honor a register declaration. Or it may choose to promote an auto data object to a register.

The C committee realized that it could seriously limit such optimizations if it insisted that `setjmp` be more predictable. Or it could effectively require that all implementations recognize `setjmp`and `longjmp`as magic functions. (Any implementation can do so if it chooses. The committee wanted to avoid requiring such behavior.) So it opted for a kludge definition.

As a result, programmers had better be careful using these functions. You should confine any `setjmp` calls to very simple expressions. Place these expressions in the smallest possible functions. Use them only where you must.

Like `<stdarg.h>`, the machinery in `<setjmp.h>` more properly belongs in the C language proper. Since it isn't there, it barely works.

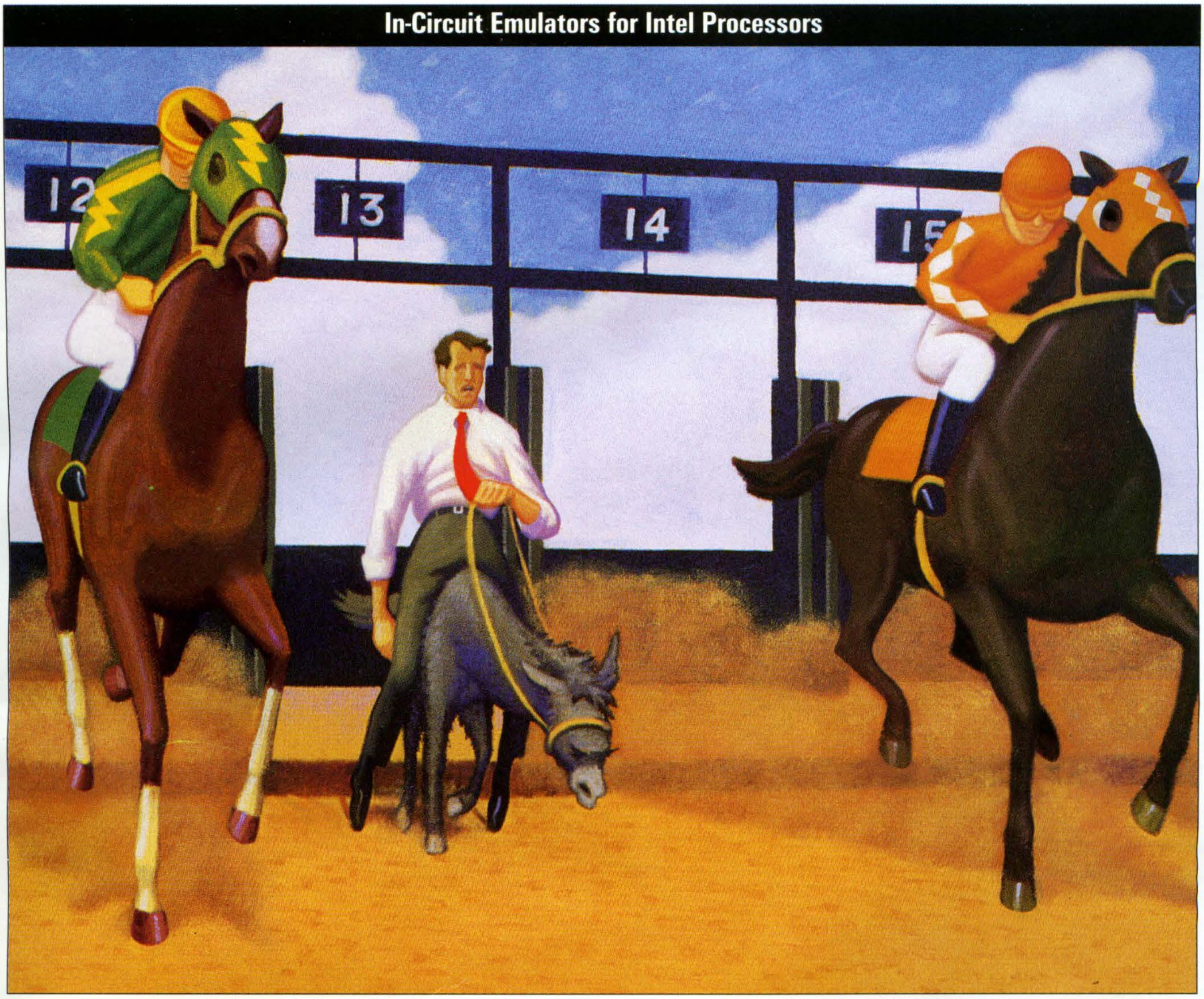
I could add more items to the list of things that barely work. The error reporting machinery built around `errno`, for example, is notoriously klunky. And signal handling has been so emasculated that it has almost no portable semantics. I consider these so clearly flawed, however, that even naïve C programmers soon learn to be wary.

I have confined my remarks to more subtle issues. These are areas that at least look like they might be properly thought out. That is part of their danger. Were they as well engineered as many people think, they would cause less trouble. Or were they more clearly flawed, they would invite less trouble. And if we had figured out how to fix them, over the past ten years, they might have gotten better in C9X. But none of them did.

None of the machinery described here is outright broken. If you use it right, it will work for you. Just don't push it. **esp**

P.J. Plauger is the author of the standard C++ library shipped with Microsoft Visual C++. His latest books are The Draft Standard C++ Library *and* Programming on Purpose *(three volumes), both published by Prentice Hall in Englewood Cliffs, NJ.*

EMBEDDED SYSTEMS PROGRAMMING (ISSN 1040-3272) is published monthly, with an additional issue published in September, by Miller Freeman Inc., 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 © 2000 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.



```
static int static_int0 = 0;
int bar = static_int0;
foo_charPtr = even_odd_str[bar];
else
```

```
Loading Command Language Extensions: C:\Pr
PG>
Loading Macro: C:\Program Files\American A
PG>
```

The Fastest BMD on the Planet...

HMI
HUNTSVILLE MICROSYSTEMS, INC.
Power
Stop
Run
Background Mode Debugger Plus US3
Serial
USB
3V DC

USB-BMD

- High-Speed USB Connection
- Supports entire device families
- Supports Flash Programming
- Extremely affordable
- Source-level debugger included
- 128 Software Breakpoints
- Supports Most Major Compilers
- Ideal for Production Environments
- Manipulate Registers
- Single-Step Source or Assembly
- Connect multiple BMDs to one PC

Background Mode Debugger for PowerPC, 683xx, 68HC12, ColdFire

At speeds topping 115 KB/second, this is now the fastest BMD on the market. Full source-level debugging features are now at your fingertips... at a fraction of the cost of traditional in-circuit emulators. Call us today.

USB
UNIVERSAL SERIAL BUS

...Just Got Faster

USB BMD

HMI has tools for

PowerPC
ColdFire
680x0
6830x
683xx
68HCxx
6809
8051
8085
8096
Z80
Z180

If you need more debugging power, we have a wide range of full in-circuit emulators with support for most major processors.

HMI-200

SPS-1000

SPS-2000

Huntsville Microsystems Incorporated

www.hmi.com | sales@hmi.com | 800-847-1998

HMI