UBM Electronics

Learn today. Design tomorrow.

ESD

EMBEDDED SYSTEMS DESIGN

The Official Publication of The Embedded Systems Conferences and Embedded.com

VOLUME 24, NUMBER 4

MAY 2011

Secrets of safety-critical C coding

16

Unleash your inner CPU, 28

Cracking unreadable code with style, 34

Introducing ZYNQ, the new element in processing.

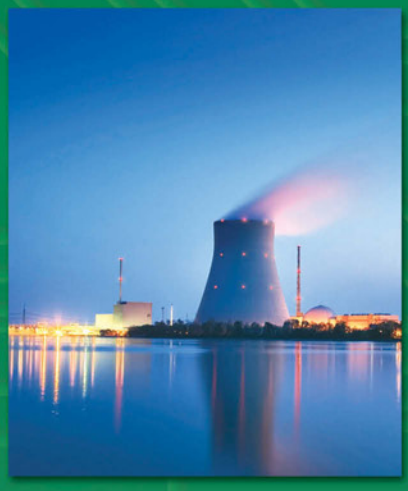
Finally, the processor comes together with the FPGA in a fully extensible processing platform called Zynq.™ More intuitive to program in the way you already know. Fully customizable to your requirements. Faster to implement and get to market. As a software engineer, if you know ARM® Cortex,™ you already know Zynq. And if you know Xilinx, you already know this is innovation you can count on.

Visit us at www.xilinx.com

XILINX ZYNQ™

The INTEGRITY® RTOS

Certified and Deployed Technology



The INTEGRITY RTOS is deployed and certified to:

Railway: ***EN 50128 SWSIL 4***, certified: 2010

Security: ***EAL6+ High Robustness***, certified: 2008

Medical: ***FDA Class III***, approved: 2007

Industrial: ***IEC 61508 SIL 3***, certified: 2006

Avionics: ***DO-178B Level A***, certified: 2002

Green Hills® SOFTWARE

www.ghs.com

Learn today. Design tomorrow.

ESD

EMBEDDED SYSTEMS DESIGN

VOLUME 24, NUMBER 4
MAY 2011

16

Cover Feature:

Seventeen steps to safer C code

BY THOMAS HONOLD

Here are 17 tips for writing safety-critical C code using methods adapted from C++ and Ada.

28 Why your embedded controller may not need a CPU

BY MARK AINSWORTH

Mark Ainsworth of Cypress Semiconductor explains how you can free up your CPU by making smart peripheral devices from a combination of programmable logic devices and datapaths.

34 Adopting C programming conventions

BY JEAN J. LABROSSE

Gone is the Wild West era of programming. Today production programmers have to follow the house style guide to get the job done efficiently. This article from the author's Embedded Systems Conference course shows you some civilizing techniques.

EMBEDDED SYSTEMS DESIGN (ISSN 1558-2493) print; (ISSN 1558-2507 PDF-electronic) is published 10 times a year as follows: Jan/Feb, March, April, May, June, July/August, Sept., Oct., Nov., Dec. by the EE Times Group, 600 Harrison Street, 5th floor, San Francisco, CA 94107, (415) 947-6000. Please direct advertising and editorial inquiries to this address. SUBSCRIPTION RATE for the United States is $55 for 10 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: Send all changes to EMBEDDED SYSTEMS DESIGN, EE Times/ESD, PO Box #3609, Northbrook, IL 60065-3257, embedsys@omeda.com. For customer service, telephone toll-free (847) 559-7597. Please allow four to six weeks for change of address to take effect. Periodicals postage paid at San Francisco, CA and additional mailing offices.

COLUMNS

programming pointers 11

Insights into member initialization

BY DAN SAKS

Often when it seems that C++ is generating bigger, slower code than C, it may be that C++ is just distributing generated code differently.

break points 46

A rumble, a wave, and iPads dry up

BY JACK G. GANSSLE

The disaster in Japan makes you think about the fundamental chemistry behind all our modern smart devices.

DEPARTMENTS

#include 5

Languages and customs

BY RON WILSON

The multilingual culture of embedded programming was replaced by a stultifying sameness. But from the ruins, a retrograde movement has begun to stir.

parity bit 7

IN PERSON

ESC Silicon Valley—May 2–5, 2011
http://esc-sv.techinsightsevents.com/

ESC Brazil—May 24–25, 2011
www.escbrazil.com.br/

ESC Chicago—June 6–8, 2011
http://esc-chicago.techinsightsevents.com/

ESC India—July 20–22, 2011
www.esc-india.com/

ESC Boston—September 26–29, 2011
http://esc-boston.techinsightsevents.com/

ONLINE

www.embedded.com

INDUSTRIAL

AEROSPACE

SYSTEM ON A CHIP

MEDICAL

AVIATION

CONSUMER

THREADX: WHEN IT REALLY COUNTS

When Your Company's Success, And Your Job, Are On The Line - You Can Count On Express Logic's ThreadX® RTOS

Express Logic has completed 14 years of successful business operation, and our flagship product, ThreadX, has been used in over 800 million electronic devices and systems, ranging from printers to smartphones, from single-chip SoCs to multiprocessors. Time and time again, when leading manufacturers put their company on the line, when their engineering team chooses an RTOS for their next critical product, they choose ThreadX.

THREADX

Our ThreadX RTOS is rock-solid, thoroughly field-proven, and represents not only the safe choice, but the most cost-effective choice when your company's product simply must succeed. Its royalty-free licensing model helps keep your BOM low, and its proven dependability helps keep your support costs down as well. ThreadX repeatedly tops the time-to-market results reported by embedded developers like you. All the while, Express Logic is there to assist you with enhancements, training, and responsive telephone support.

Join leading organizations like HP, Apple, Marvell, Philips, NASA, and many more who have chosen ThreadX for use in over 800 million of their products – because their products are too important to rely on anything but the best. Rely on ThreadX, when it really counts!

Contact Express Logic to find out more about our ThreadX RTOS, FileX® file system, NetX™ Dual IPv4/IPv6 TCP/IP stack, USBX™ USB Host/Device/OTG stack, and our new PrismX™ graphics toolkit for embedded GUI development. Also ask about our TraceX® real-time event trace and analysis tool, and StackX™, our patent-pending stack size analysis tool that makes stack overflows a thing of the past. And if you're developing safety-critical products for aviation, industrial or medical applications, ask about our new Certification Pack™ for ThreadX.

expresslogic

For a free evaluation copy, visit www.rtos.com • 1-888-THREADX

EMBEDDED SYSTEMS DESIGN

Editorial Director
Ron Wilson
(415) 947-6317
ron.wilson@ubm.com

Managing Editor
Susan Rambo
(415) 947-6675
susan.rambo@ubm.com

Acquisitions/Newsletter Editor, ESD and Embedded.com
Bernard Cole
(928) 525-9087
bccole@acm.org

Contributing Editors
Michael Barr
Jack W. Crenshaw
Jack G. Ganssle
Dan Saks

Art Director
Debee Rommel
debee.rommel@ubm.com

Production Director
Donna Ambrosino
dambrosino@ubm-us.com

Article submissions
After reading our writer's guidelines, send article submissions to Bernard Cole at
bccole@acm.org

Subscriptions/RSS Feeds/Newsletters
www.eetimes.com/electronics-subscriptions

Subscriptions Customer Service (Print)
Embedded Systems Design
PO Box # 3609
Northbrook, IL 60065- 3257
embedsys@omeda.com
(847) 559-7597

Article Reprints, E-prints, and Permissions
Mike Lander
Wright's Reprints
(877) 652-5295 (toll free)
(281) 419-5725 ext.105
Fax: (281) 419-5712
www.wrightsreprints.com/reprints/index.cfm?magid=2210

Publisher
David Blaza
(415) 947-6929
david.blaza@ubm.com

Associate Publisher/ Sales North America
Bob Dumas
(516) 562-5742
bob.dumas@ubm.com

UBM Electronics

Corporate—UBM Electronics

Paul Miller	*Chief Executive Officer*
David Blaza	*Vice President*
Karen Field	*Senior Vice President, Content*
Felicia Hamerman	*Vice President, Marketing*
Brent Pearson	*Chief Information Officer*
Jean-Marie Enjuto	*Vice President, Finance*
Amandeep Sandhu	*Director of Audience Engagement & Analytics*
Barbara Couchois	*Vice President, Partner Services & Operations*

Corporate—UBM LLC

Marie Myers	*Senior Vice President, Manufacturing*
Pat Nohilly	*Senior Vice President, Strategic Development and Business Administration*

#include

BY Ron Wilson

Languages and customs

There was a time when we were multilingual. Assembly language was the only alternative to machine code, and each microprocessor and microcontroller family had its own language, reflecting its own distinct instruction set architecture. There were similarities, often artifacts of the limitations that small transistor budgets imposed on silicon architects. And there were differences, arising from those architects' determination to create a better instruction set from their meager materials.

Master programmers fought against the complexity of assembly-language programming, and against the entropy that complexity engendered. Receiving no help from language structures, which only reflected the underlying chip hardware, they turned to styles, practices, and customs. They adapted notions such as structured programming and strong type-checking from the big-computer world—not as elements of some new language, but as manual practices. Above all, they relied on documentation to make their code comprehensible to those who would come after them.

Eventually, high-level-language compilers began to appear for MPUs: PLM from Intel, and then formal languages—Pascal and even Ada. These latter two languages from the big-machine world brought, inherent in themselves, ideas: strong type-checking, structure, some degree of readability, and the underpinnings of formal proofs. The foundations were coming together upon which we could build a reliable software development methodology.

Then something went wrong. Like barbarian hordes from the North, doctrinaire disciples of Unix swept out of the universities, bearing with them the sacred language C. It became incorrect to criticize the Unix cult of individual expression, or the fundamental rightness of exhibitionistic coding, or to raise questions about the language of the new secular gospel. Pascal and Ada retreated before the onslaught, first to remote fastnesses, and eventually to near oblivion. With them went the multilingual culture of embedded programming, replaced by a stultifying sameness. With them too went the idea that a language could enforce good programming practice, driven out by a language that reveled in conciseness and actively encouraged practices we knew to be bad.

Eventually, from the ruins a retrograde movement has begun to stir—a turn toward the days when, unassisted by their languages, master programmers imposed type, structure, and even provability on their code through their own practices. As our cover story author this month, Thomas Honold, argues, C may not encourage good practices, but it can't suppress them. In another article, Jean Labrosse uses C to express some universal techniques adaptable to any code production line.

And what of the future? A new generation of programmers is emerging for whom C is ancient, irrelevant history. Many have used only Java. At the same time, and not coincidentally, development platforms such as Android are appearing, enabling Java programmers to patch together embedded systems without resort to less-abstract languages. Will this new wave again engulf all that we have learned about creating reliable, maintainable systems? Will quality in embedded code sink to the level of Web code? Or will the experience of several generations in developing mission-critical systems, meeting requirements, and simply writing good code survive? Though languages change, will customs endure?

Ron Wilson is the editorial director of design publications at UBM Electronics. You may reach him at ron.wilson@ubm.com.

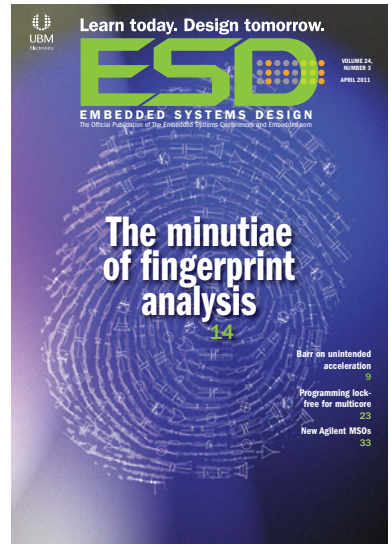
Debating NASA's report on unintended acceleration

The title of this article is irresponsible *("Unintended acceleration and other embedded software bugs," Michael Barr, April 2011, www.eetimes.com/4214602)*. No bug in Toyota / Denso's module has been found yet that causes unintended acceleration, while at least three mechanical and/or user errors have been identified (sticky pedals, floor mats, hitting the wrong pedal). *—JDT*

So this [investigation by NASA and NHTSA] was considered to have been an adequate "safety review" and yet no one even thought to attempt to verify whether these "CPUs" as actually implemented in the ASIC even faithfully executed the instruction set of the V850E1 (as would have been required by RTCA DO-254 if the code had been written for avionics instead of an automobile)? This is all just a sham and a whitewash designed to fatten the wallets of the attorneys and appease the car-buying public. It has nothing to do with an orderly safety review. *—JeffL_#2*

The other reason for simple tools for critical code is that the device needs to work even if some powerful radar chirps a worse-case code on any register (visible, invisible like TLA for cache, or virtual). Practically this means separate hardware to compare results with.

When my life depends on safety devices, I want reliably EMP-resistant, 1,000-year solar-storm–resistant, radar- and jammer-resistant, typo-resistant, sensor-plugging–resistant, final-actuator error-resistant safety systems. When we know how IC scaling affects EMP and other RF assaults on our safety systems, then we'll know how to write stable standards. Until then, we need to keep thinking and making [systems] as simple as possible. *—FluidCamp*



We might go on and on about code safety, but even in the worst offenders, the most dangerous part in a car is the lump of meat behind the steering wheel.

Sure you want these things, but are you prepared to pay for them? Every car could have a titanium alloy roll cage and racing bucket seats with a five-point harness. Would you pay for that?

We might go on and on about the safety of the code, but even in the worst offenders the most dangerous part in a car is the lump of meat behind the steering wheel. *—cdhmanning*

Multiprocessing hype or hope

Wow, finally someone is talking about the real issues and not just the hype *("Multiprocessing in your future," Ron Wilson, April 2011, www.eetimes.com/4214767)*. My whole career was involved with systems and debug. The whole notion of multicore, multithreading, RTOS, interrupts, shared data, mutexes, etc., creates an infinite number of states in the SoC and debug complexity gets to be overwhelming. I understand that multicore shares a lot of hardware resources and that it's too expensive to have multiple independent conventional CPUs.

Looking at high-level languages, I found that it's possible to design a general purpose hardware accelerator that is very small, very fast, and programmable using if/else, for, while, call statements. Unfortunately everyone seems so caught up in the "multi" hype that they don't think it's credible. *—KarlS*

Interface is very important and very time consuming but that's not my focus on the multiprocessors. I add another micro whenever there is any time crunch, set that micro up to do the preprocessing, then have the main processor (sometimes called the housekeeper) just ask for the results—usually a "true or false" answer. *—ccrican1*

Recent literature tends to focus on the challenges and hassles of multicore but rarely dwell on the advantages. My own experience with (usually wildly heterogenous) multicore has been that the overall job was simplified, with a cleaner design, and shockingly easier debugging. The time spent up-front on system partitioning was more than repaid by rapid development of reliable high-quality code. *—vapats*

A new trick for watchdog timers

As always, Jack Ganssle's article on watchdog timers (*"Watchdogs redux" March 2011, www.eetimes.com/4213592*) was interesting. In particular, I thought the discussion of the ST processor doing an interrupt call before resetting the processor on a WDT timeout was interesting.

Leading Embedded Development Tools

A full featured development solution for ARM Powered® Linux and Android platforms.

ARM®

1-800-348-8051
www.arm.com/ds5

NASA'S REPORT: A DISCUSSION ON C

I'm skeptical about The Power Of Ten [mentioned in *"Unintended acceleration and other embedded software bugs," Michael Barr, April 2011, www.eetimes.com/4214602*]. Some of it is self contradictory:

- **Rule 1: Simple control structures. Don't use goto and recursion.** Sometimes the simplest flow control is to use gotos and recursion. Used correctly, both of these can be used to generate very clean code that is far easier to read and verify than convoluted flow control that excludes them.
- **Rule 2: Always use a fixed upper bound in loops.** That means we have to write code like:

```
for(i = 0; i less than n && i less than MAX_N; i++){...}
```

which is confusing and cluttering. It would be better to use an assert:

```
assert(n less-than-or-equal MAX_N);
for(i = 0; i less than n; i++){ }
```

Read more comments online at www.eetimes.com/4214602

The Zune bug used as an example is not justification enough. The Zune bug is terrible code. —*cdhmanning*

And the rationale for rule 6 of the 10 (not such a bad rule in general) makes the claim that "Clearly if an object is not in scope, its value cannot be referenced or corrupted." This statement is obviously false in the case of C. It all makes me wonder about the inherent difficulty of this process. The dismay of the reviewers when presented with hundreds of thousands of lines of C code is quite understandable. The language is so semantically impoverished that it's intrinsically resistant to analysis, so it's hard to see how anyone can reasonably establish the correctness of something that large without immense effort. —*willc2010*

I've been doing embedded software for decades and am growing increasingly concerned about quality. Having reviewed literally millions of lines of code, quality ranges from excellent to horrific, with most in the barely acceptable range. One single function I reviewed was over 1,000 lines long. Numerous times, large chunks of code were cut and pasted with little or no change, rather than making a function call.

Nothing replaces good, old common sense. C provides the opportunity to create really obfuscated code. Resist the temptation. Follow the KISS principle. And, above all, remind yourself that other people read your code far more often than you write it. —*Fuzzball*

The problem with writing applications in C (especially real-time ones) is that you're at the mercy of the compiler and probably the RTOS. Gotchas can occur that you have no control over and can't anticipate. What's wrong with interrupts and assembly language? I've been doing that for 40+ years. —*Jerry.Brittingham*

While Jerry might have a point about RTOS vs. bare metal, I don't believe his C vs. assembler argument stacks up. These days compilers go through so many validations that compiler bugs are very rare. The biggest source of software errors is the programmer. C makes it easier to see what is going on and thus makes it harder to hide bugs in C code vs. assembler.

Perhaps a larger issue is how the hell has critical software gotten so complex? Why does an engine controller need hundreds of thousands of lines of code? Surely critical code like that can be done in a thousand or two lines tops. —*cdhmanning*

I don't think that the issue is so much compiler bugs as unexpected behavior. Even if one has a bug-free C/C++ compiler, it's often either ambiguous or at least very unclear what the correct behavior should be. This is made worse by the language's predilection for construing plausible interpretations and applying implicit conversions, as if the goal were to do something, rather than do the right thing. On top of that is the fundamental weakness of the type system, primitive semantics and so on, but those issues are mostly relevant in comparisons with higher-level languages.

It is true that the interpretation of C/C++ programs is highly dependent on their execution environment (RTOS, threading library, etc.). That is the inevitable consequence of the bolt-on approach to issues like tasking that has always characterized that language family (compare with Ada, where the tasking semantics are defined by the language itself and the compiler vendor has to make the implementation comply, whether the target is an OS or a bare board).

However, I also don't think that the solution in general is to code in assembler. Even a C compiler takes care of a mass of mundane details about parameter passing, expression evaluation, and so on (albeit with some 'gotchas'), and implementations for different targets are at least more-or-less similar to each other. The problem with C is not that it is too high-level, but that it is not high-level enough. It doesn't allow the programmer to express precision of meaning. It is more an assembler substitute than a high-level language. As such it demands a great deal of checking and testing.

I agree that the size of the code quoted in some of these articles seems excessive on the face of it. Does a car really require more software than an airliner? —willc2010

From the very beginning of my work with microprocessors (using the Motorola MC6802), I've seen a similarity between hardware interrupts and a reset. Both cause the system to vector to an address, but the interrupt pushes the current PC on the stack. Why could a system reset not also do that? Startup code could, after determining the reset was caused by the WDT, pull the offending address from the stack and make it available for debug.

In a recent project on a PIC32, I wrote my own WDT function that, along with resetting the hardware WDT, reset another timer. That timer was set to generate an interrupt a little before the WDT timed out. If that timer ever times out, the ISR logs the PC, pulled from the stack, then goes into a loop waiting for the WDT to time out and do a hardware reset of the system. On reset, the system also logs the cause of the reset (from the RCON register) to further help in troubleshooting the system.

Years ago, I had to drive for several hours to push a reset button at a remote radio station transmitter site. Since then, *everything* includes a WDT. —*Harold Hallikainen*

Send your comments to the editor, Ron Wilson, at ron.wilson@ubm.com or enter them online under an article.

HUGE discounts

on select RIGOL Beyond Measure equipment - for a limited time only!

DS1052E
50MHz 2-Channel Digital Oscilloscope
Was $519 **Now only $399**

DG1022
20MHz Function/Arbitrary Waveform Generator
Was $695 **Now only $499**

DS1052D
50MHz Mixed-signal Scope - 2ch + 16 Digital Channels
Was $1,095 **Now only $899**

DS1102D
100MHz Mixed-signal Scope - 2ch +16 Digital Channels
Was $1,495 **Now only $1,099**

DG1022 + DS1052E
Basic Signal Analysis Package **Was $1,094 Now only $799**
20MHz Function/AWG + 50MHz 2-Channel Digital Oscilloscope

DG2041A + DS1102CA
Standalone Signal Analysis Package **Was $2,290 Now only $1,799**
40MHz Function/AWG + 100MHz 2-Channel Digital Oscilloscope

Hurry! This sale is only for a limited time! Call us at 888-772-3544 or go to www.saelig.com today!

Saelig.com unique electronics

By Dan Saks

Insights into member initialization

Among the most common reasons that C programmers offer to explain why they're disinclined to use C++ is that C++ does too much behind the scenes. A closely-related complaint is that C++ compilers generate too much code for seemingly simple expressions. If you look online at the reader comments on my columns over the last few years,[1] you'll see remarks to that effect now and then.

Most of these complaints don't hold up well under scrutiny. Often, the alleged excess code simply isn't there. For example, function overloading and friendship are strictly translation-time facilities. They don't incur any run-time costs.

At other times, excess code appears only when targeting some processors and not others. For example, some processors are better than others at calling virtual functions. Even then, the code for calling a virtual function in C++ is usually about the same as calling a function through a pointer in C.

When the complaints do have merit, it's often that C++ isn't necessarily generating bigger and slower programs than C. It may be that C++ just distributes the generated code differently. It generates more code in some places and less in others. I believe that once you understand why C++ does what it does, the resulting code not only ceases to be surprising, but even becomes predictable. Such is the case with constructors.

Often when it seems that C++ is generating bigger and slower code than C, it may be that C++ is actually just distributing generated code differently.

A *constructor* is a special class member function that provides guaranteed initialization for objects of its class type. Since the beginning of the year, I've been explaining what constructors are in C++ and what kind of code they generate.[2,3] This month, I'll continue by explaining the interesting behavior of constructors for classes with members that have constructors of their own. As I often do, I'll illustrate the behavior using equivalent C code.

CLASS OBJECTS AS MEMBERS

Just as a C structure can have members that are themselves structure objects, a C++ class can have members that are themselves class objects. For example, let's look at a class for entries in some kind of symbol table, where each entry stores a name and some associated information.

To keep this simple, let's just say an entry has a name, an id, and a value. The name is the textual spelling of the entry's name. The id is an unsigned integer value that uniquely identifies each entry. The value is a sequence of one or more signed integer values associated with the name. The `entry` class definition looks in part like:

```
class entry
    {
    ~~~
private:
    string name;
    unsigned id;
    sequence value;
    };
```

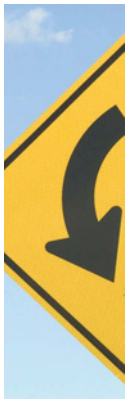
Dan Saks is president of Saks & Associates, a C/C++ training and consulting company. For more information about Dan Saks, visit his website at www.dansaks.com. Dan also welcomes your feedback: e-mail him at dan@dansaks.com.

Here, `string` is a class representing a variable-length string of characters. It might be the string class from the Standard C++ Library, or it might be a class custom built for this application. The `sequence` class represents a sequence of signed integer values. It might be a typedef name that's an alias for a Standard Library class template instantiation, such as:

```
typedef vector<int> sequence;
```

Then again, it might be a custom built class.

Now let's examine the behavior of various constructors for this `entry` class.



If the compiler generates a default constructor for class entry, that default constructor calls the default constructor for each member of class type.

GENERATED DEFAULT CONSTRUCTORS

As I explained in my first article on constructors, a definition for a class object can specify a constructor argument list, as in:

```
entry e (n, v);
```

This defines `e` as an entry `object`. In this case, the compiler generates code that initializes `e` by calling a constructor that accepts `n` and `v` as arguments. If the `entry` class declares no such constructor, the compiler will blurt out nasty things.

In limited cases, the compiler may generate a constructor. For example, a definition for an object with no argument list, as in:

```
entry e;
```

invokes a particular constructor called the *default constructor*. The default constructor is special in that the compiler may generate it, *but only if the class has no explicitly declared constructors at all.*

If the compiler generates a default constructor for class `entry`, that default constructor calls the default constructor for each member of class type. In this case, the default `string` constructor would be called for member `name`, and the default `sequence` constructor would be called for member `value`. A C function that performs the same initialization as the generated default `entry` constructor might look like:

```
void construct_entry(entry *_this)
     {
     string_construct(&_this->name);
     sequence_construct(&_this->value);
     }
```

This function doesn't initialize the `entry`'s `id` member, which has a non-class type and thus can't have a constructor. Generated default constructors leave such members uninitialized.

Most compilers don't generate code for a default constructor unless the program actually uses that constructor. Calls to a generated default constructor may be expanded inline.

USER-DEFINED DEFAULT CONSTRUCTORS

The generated default constructor doesn't construct `entry` objects properly because it doesn't initialize the `id` member. Uninitialized objects have indeterminate values.

Each `entry` should have a unique id. An easy way to implement unique ids is to obtain them from a counter that increments at each constructor call. In C++, that counter can and probably should be a private static data member, declared as:

```
class entry
     {
     ~~~
private:
     static unsigned counter;
     string name;
     unsigned id;
     sequence value;
     };
```

In C, the counter might be a global object or a local static object.

In C++, a default constructor that provides an appropriate `id` value might look like:

```
entry::entry()
     {
     id = ++counter;
     }
```

On the surface, it looks like this constructor doesn't initialize the `name` and `value` members, but it actually does. It applies a default constructor to each member, just as a generated default constructor would. That's why they're called "default" constructors—they're the ones the

DEVELOPING THE NEXT GENERATION OF

INNOVATION

KEEPING UP WITH THE PACE OF INNOVATION IN THE NEW ECONOMY REQUIRES **VISION, SKILL, AGILITY** AND **EXPERTISE**. THOSE WHO WILL BRING THE NEXT GENERATION OF **GREAT PRODUCTS** TO MARKET FACE CHALLENGES RANGING FROM AGGRESSIVE DEVELOPMENT SCHEDULES TO POSSESSING THE KNOWLEDGE AND MANPOWER TO PERFORM AND EXECUTE. WITH OVER **25 YEARS OF TECHNOLOGY DEVELOPMENT** EXCELLENCE, STRATOS CAN HELP.

Whether you need a full system development team or are looking for a specialized area of expertise, call on us for:

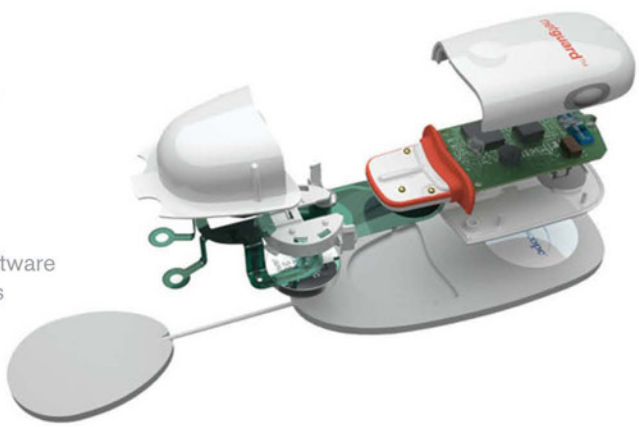
- Embedded software and systems development
- Hardware/software interfaces, embedded OS, DSP, sensors, controls, low-power wireless
- Embedded user interface development
- Medical device development with FDA-compliant design controls
- Strong concept design and architecture capabilities
- Industrial design and human factors engineering
- Full-capability integrated product development: hardware, software, mechanical, quality, project management

STRATOS

product development

"Stratos brought experience in patient monitoring, wireless software development, wearable human factors, miniaturized electronics and a total systems approach to our product development."

Director of NetGuard Business, Datascope



Learn today. Design tomorrow.

ESC Silicon Valley • May 2 - 5, 2011

VISIT US AT **BOOTH #2432** AT ESC SILICON VALLEY TO LEARN MORE ABOUT OUR INNOVATIVE APPROACH TO PRODUCT DEVELOPMENT AND ENTER TO **WIN AN APPLE iPAD™ 2**.

stratos.com

program calls by default. A C function that performs the same initialization as the default `entry` constructor defined just above might look like:

```
void construct_entry(entry *_this)
     {
     string_construct(&_this->name);
     sequence_construct(&_this->value);
     _this->id = ++counter;
     }
```

This user-defined default constructor still might not be very useful. The default constructors for `string` and `sequence` probably create empty objects. If so, the default constructor for `entry` produces an object with no name and no value. You might not want such objects floating around in the application.

If you want to ensure that every entry has a non-empty name and value, then you can define an entry constructor that requires arguments for the name and value.

NON-DEFAULT CONSTRUCTORS

If you want to ensure that every `entry` has a non-empty name and value, then you can define an `entry` constructor that requires arguments for the name and value. You might declare that constructor as:

```
class entry
     {
public:
     entry(string const &n, int v);
     ~~~
     };
```

The corresponding constructor definition might look like:

```
entry::entry(string const &n, int v)
     {
     name = n;
     value.push_back(v);
     id = ++counter;
     }
```

The first statement in the constructor body assigns parameter `n` to `entry` member `name` using an assignment operator defined in the `string` class. (It actually uses a particular assignment oper-

Listing 1 A C function that performs the same work as the `entry` constructor.

```
void construct_entry_nv(entry *_this, const string *n, int v)
     {
     string_construct(&_this->name);
     sequence_construct(&_this->value);
     string_copy(&_this->name, n);
     sequence_push_back(&_this->value, v);
     _this->id = ++counter;
     }
```

ator known as the *copy assignment*. It's in my queue of things to discuss eventually. I'm also aware that the argument passed for parameter `n` could be an empty string, so this constructor doesn't ensure that the `name` will be non-empty. That's curable, but I don't want to get sidetracked on that now.)

The second statement appends the value of parameter `v` to the end of the sequence stored in `entry` member `value`. The Standard C++ Library containers use the name `push_back` for this operation, so I do, too.

When they see C++ compilers generating code like this, they might feel their complaints are justified. If this were the end of the story, I'd agree. But it's not.

Strictly speaking, the `sequence`'s `push_back` is not an initialization. It modifies the value of a previously constructed `sequence` object. That is, `push_back` operates on the assumption that `sequence` already has an initial value. Calling `push_back` appends one more value to whatever's already there.

Similarly, the `string`'s assignment operator is not an initialization. It replaces the value of a previously constructed `string`. It will likely fail if the `string` isn't already initialized.

Remember, `entry`'s members `name` and `value` have class types. Those classes have constructors. Constructors provide guaranteed initialization, meaning that each object that has a type with a constructor must be initialized by calling one of those constructors before any operations may be performed. This is true for objects even when they're members of other objects.

C++ preserves the guarantee by inserting default constructor calls for `entry`'s members into the `entry` constructor itself. Specifically, the compiler generates a call that applies the default `string` constructor to `entry`'s member `name`, and another call that applies the default `sequence` constructor to member `value`. A C function that performs the same work as the `entry` constructor might look like the code in **Listing 1**.

In effect, this constructor initializes the `entry`'s `name` member to be empty, only to immediately replace that value with something else. Wouldn't the code be shorter and faster if it simply initialized the name member with a copy of `n`? Similarly, the `entry` constructor initializes the `value` member to be an empty sequence, only to immediately append one value. Wouldn't it be better to just initialize the `sequence` member to hold a copy of that single value?

MEMBER INITIALIZERS

Some C programmers are disinclined to use C++ because they think it does too much behind the scenes. When they see C++ compilers generating code like that in Listing 1, they might feel their complaints are justified. If this were the end of the story, I'd agree. But it's not.

C++ extends constructors with an additional facility called member initializers. Member initializers avoid the inefficiency of unnecessary calls to default constructors by initializing members directly. Member initializers will be the subject of my next column. ■

ENDNOTES:

1. Programming Pointers colums are available at *www.eetimes.com/electronics-blogs/27/Programming-Pointers*
2. Saks, Dan, "Demystifying constructors," *Embedded Systems Design*, January/February 2011, p. 9. *www.eetimes.com/4212701*
3. Saks, Dan. "Constructors and object definitions," Embedded.com, March 2011. *www.eetimes.com/4213712*

The author gives 17 tips for writing safety-critical C code using methods adapted from C++ and Ada.

Seventeen steps to safer C code

BY THOMAS HONOLD

In embedded systems design, many of us tend to write our software in C the way our "grandfathers" did, which was appropriate before we had to worry about ubiquitous connectivity and its security implications. Today, the programming methods of the past must be adapted to a world in which safety-critical design is required not only in military/aerospace applications but in ordinary commercial applications as well. The C language is definitely not type safe, and only by applying many good practices and self-imposed rules can it be made a viable choice for safety-critical software development.

I learned these rules and best practices working for companies that were moving to programming paradigms more amenable to safety-critical software development. For example, at one company we were developing Internet banking and chip-card terminal applications, using C++ on Windows PCs. At the time, we believed we were doing object-oriented programming, but I now believe that what we were writing was really C with C++ syntax.

Having seen it often enough, I theorize that embedded systems developers naturally fall into this type of hybrid coding when migrating from procedural C programming to the object-oriented C++ paradigm.

Later I moved on to what was, at the time, a new domain of software engineering: safety-critical embedded systems design. My first project required me to learn Ada. At the end of the project, I understood that new hardware in the embedded systems area also means new software and firmware, and I learned from Ada what type safety really means.

NEWS 2011

LAUTERBACH DEVELOPMENT TOOLS

DEBUGGER, REAL-TIME TRACE, LOGIC ANALYZER

Always a Few Steps Ahead

Lauterbach has been developing tools for the embedded industry for over 30 years advocating this slogan. For most new debug technologies Lauterbach is the world leader and trend setter.

This has allowed us to gain the recognition of all the big semiconductor manufacturers. For many years, those involved in developing and implementing new technologies have favored collaboration with Lauterbach. This collaboration has inspired many ground breaking ideas to be transformed into advanced products.

In addition, Lauterbach is very customer focused. The desires and suggestions of our TRACE32 users provide a valuable contribution to our product development. In many cases, suggestions are put into practice immediately and are then included in the next released version of our debugger.

From this vantage point what trends does Lauterbach currently see? What technologies are soon to emerge in the market?

Android Debugging

Android debugging is certainly an important topic. Applications for mobile phones are increasingly being written architecture-independent for virtual machines (VM). Google's Android and its Dalvik VM are quite prevalent. Complex errors that will only appear with the interplay of application, virtual machine, operating system and the underlying hardware have to be debugged. To do this it is necessary to have transparency through all of the software layers, from the Java application down to the Linux hardware drivers.

At the request of some mobile phone manufacturers, Lauterbach started developing an API for *VM Debugging Awareness* in the middle of 2010. Android is used here as a reference platform. The aim is to provide an open interface that allows providers of open-source and closed-source VMs to adapt their products for debugging with TRACE32. For information on *VM Debug-* »

CONTENTS

New Supported Processors	4
Tracing for Virtual Targets in Fast Models	5
API for VM Debugging Awareness	6
Extensions and New RTOS Versions	8
Serial Trace Port Usage Growing	9
Higher Transmission Rate for RTS	10
Energy Profiling with the CombiProbe	11
SMP Profiling	12

ging Awareness and the current state of development, see the article “API for VM Debugging Awareness” on page 6.

Energy Profiling

Energy measurement for embedded systems has come more into focus with the increasing emphasis on global warming and “green” electronics systems. Every technical journal now contains many articles on battery-driven equipment and low-power microcontrollers. Prizes for innovation are increasingly being awarded for new technologies in this field.

However, in the mobile phone market standby and operating times have always been an important topic. For years, extensive energy reduction measures have been implemented in this area. But these measures only make sense if the software that controls an embedded system consistently uses all the energy-saving features of the hardware.

Since the beginning of 2006, Lauterbach tools have supported measuring arrangements that allow the simple comparison and analysis of the interplay between software and power consumption in an embedded system. This technology has also been available for the TRACE32 CombiProbe since mid 2010. For more information on “Energy Profiling with the CombiProbe”, see page 11.

Multicore Debugging

Although multicore chips have been used in embedded systems for ten years and Lauterbach has had debuggers for them since 2001, this is still a highly dynamic topic. The current calls for greater visibility into the internal system operation are ensuring the integration of new trace cells within the debug infrastructure of the chips.

Originally, trace information was only generated for the individual cores, whereas today there are many other trace sources:

a) Trace sources that make transfers on chip-internal buses visible:

- ARM CoreSight with the AMBA AHB Trace Macrocell (HTM)
- MCDS with the System Peripheral Bus (SPB) and the Local Memory Bus (LMB) for the TriCore from Infineon
- RAM Trace Port for chips from Texas Instruments
- DMA and FlexRay trace for NEXUS Power Architecture

b) Trace sources that generate trace information for chip-internal IP (Intellectual Property), such as special interrupt traces.

c) Trace sources that permit the output of software-generated trace information, such as:

- Instrumentation Trace Macrocell (ITM) for ARM CoreSight
- System Trace Macrocell (STM) for ARM CoreSight

The continuous development of the TRACE32 debugger ensures it is aware of these new trace sources and can provide easy configuration and a comprehensive analysis of the information provided.

Serial Trace Ports

Due to the extra trace data provided by this visibility into the internal chip processes, complex multicore chips and high-performance processors require more and more bandwidth and thus even faster trace ports.

In response chip manufacturers have developed serial trace ports as an important innovation in the last few years. Hard-disk manufacturers, who have been using serial interfaces for high-speed data exchange with the PC for years, used this technology for the first time in 2008 to export trace information via ARM’s High Speed Serial Trace Port (HSSTP). At the same time, Lauterbach launched trace tools for this technology.

In the meantime, there are other processor families with serial trace interfaces. For current developments in this area, see the article “Serial Trace Port Usage Growing” on page 9.

Bigger Trace Memory

Fast trace interfaces with their high data rates inevitably require more trace memory. Without this, it is impossible to capture a sufficiently large program section for troubleshooting and the analysis of the time behaviour for an embedded system. »

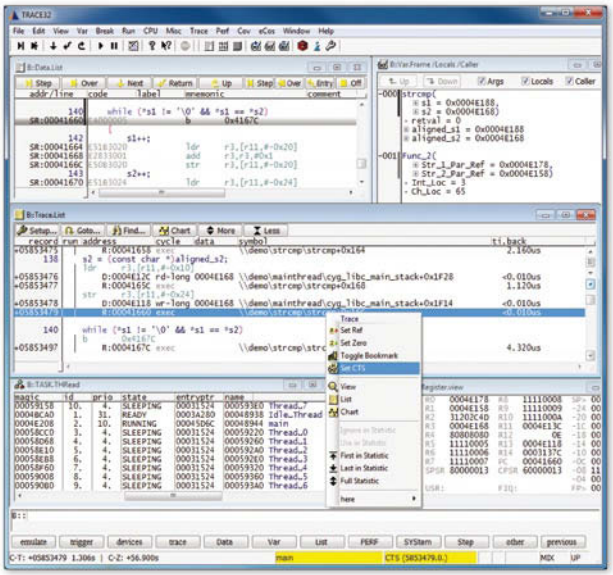


Fig. 1: Even demanding 4-GB trace memory analysis for Trace-based Debugging can be performed quickly.

However, providing more and more trace memory only makes sense if the necessary infrastructure for fast processing of the trace information is available. This applies particularly to demanding trace analysis functions such as *Trace-based Debugging* (see Fig. 1). The increasing capacity of SDRAM chips, fast PCs, and GB Ethernet interfaces enabled Lauterbach to launch the trace tool PowerTrace II with 4 GB memory in 2007.

In mid-2008, Lauterbach started developing a new method of trace recording and analysis, *Real-time Streaming*. This development was driven by customers' demands for long-term code coverage analysis, for comprehensive system runtime analyses and for a much longer trace recording time to locate infrequent errors.

The new feature of *Real-time Streaming* is that the trace data is transferred to the host while it is being recorded. The trace information is then analyzed on the host as soon as it is received. Optionally, the trace information can also be saved to the hard-disk while it is being analyzed.

Real-time Streaming works only if all processing steps for the trace data run at optimal speed. This applies to transfer and analysis, as well as the systematic search for trace information in a file saved to the hard-disk.

Conventional tracing also profits from many of the new speed optimizations. For example, there are plans to implement the trace-data compression (developed for *Real-time Streaming*) also for conventional tracing. For details on the trace-data compression, see page 10.

Trace-Based Debugging

Trace-based Debugging (also known as CTS = Context Tracking System) allows re-debuggging of a traced program section. TRACE32 makes this possible as it can reconstruct the state of the target system for each individual trace record in its PowerView GUI. This reconstruction includes the register and memory contents, variable states, source and task listing, stack-frame, and much more.

After choosing a starting point for Trace-based Debugging, all of the debug commands can be used. These commands are executed by TRACE32 based upon the reconstruction from the trace recording. Many users of Trace-based Debugging appreciate the fact that they can also step backwards or return to the function start.

Trace-based Debugging also provides a series of other useful functions:

- Trace display in high-level language with all local variables
- Runtime analyses and function call tree
- Reconstruction of the trace gaps that can occur if more trace data is being generated than can be exported via the trace port

www.lauterbach.com/cts.html

Outlook

In addition to the current trends, there are a large number of new developments in debug technology. When you browse through our 2011 newsletter, you will probably discover one or two of these that might help with your project. We will be demonstrating several of them live at the upcoming ESC Silicon Valley, May 2-5, in San Jose, and also at many other shows in the US throughout the year.

Visit us: Booth # 1922

New Supported Processors

New derivatives

Actel	**LA-7844 (Cortex-M)** • A2F060, A2F200, A2F500
AppliedMicro	**LA-7723 (PPC400)** • APM80186, APM821x1 • APM86290 **LA-7752 (PPC44x)** • PPC460SX
ARM	**LA-7843 (Cortex-A/R)** • Cortex-A15 • Cortex-A15 MPCore **LA-7844 (Cortex-M)** • Cortex-M4 • SC000, SC300
Atmel	**LA-7844 (Cortex-M)** • AT91SAM3S, AT91SAM3N **LA-3779 (AVR32)** • AT32UC3A/B/C/D/L
Broadcom	**LA-7760 (MIPS32)** • BCM3549/35230/4748 • BCM5354/5358/5331X • BCM6816/6328/6369 • BCM7407/7413/7420
Cavium	**LA-7761 (MIPS64)** • CN63XX
Ceva	**LA-3711 (CEVA-X)** • CEVA-X1643, CEVA-XC
Cortus	**LA-3778 (APS)** • APS3/B/BS/S
Cypress	**LA-7844 (Cortex-M)** • PSoC5
Faraday	**LA-7742 (ARM9)** • FA726TE
Freescale	**LA-7736 (MCS12X)** • MCS9S12GC/GN/Q **LA-7732 (ColdFire)** • MCF5301x, MCF5441x **LA-7845 (StarCore)** • MSC8156 **LA-7742 (ARM9)** • i.MX28 **LA-7843 (Cortex-A/R)** • i.MX53 **LA-7844 (Cortex-M)** • Kinetis
Freescale (Cont.)	**LA-7753 (MPC55xx/56xx)** • MPC5602D/P • MPC564XA/B/C/S • MPC567XF/R **LA-7729 (PowerQUICC II)** • MPC830X **LA-7764 (PowerQUICC III)** • P10xx, P20xx, P40xx • P3041 (2H/2011) • P5010, P5020 (2H/2011)
Fujitsu	**LA-7844 (Cortex-M)** • FM3
Infineon	**LA-7756 (TriCore)** • TC1182, TC1184 • TC1782, TC1782ED • TC1784, TC1784ED • TC1791, TC1791ED • TC1793, TC1793ED • TC1798, TC1798ED **LA-7759 (XC2000/C166S V2)** • XC22xxH/I/L/U • XC23xxC/D/E/S • XC27x2/x3/x7/x8 • XE16xFH/FU/FL
Intel®	**LA-3776 (Atom™/x86)** • E6xx, Z6xx, N470 • Core i3/i5/i7, Core2 Duo
Lantiq	**LA-7760 (MIPS32)** • XWAY xRX200
LSI	**LA-7765 (ARM11)** • StarPro2612, StarPro2716 **LA-7845 (StarCore)** • StarPro2612, StarPro2716
Marvell	**LA-7742 (ARM9)** • 88F6282, 88F6283, 88F6321 • 88F6322, 88F6323 **LA-7765 (ARM11)** • 88AP510-V6 **LA-7843 (Cortex-A/R)** • 88AP510-V7
MIPS	**LA-7760 (MIPS32)** • MIPS M14K, MIPS M14KC
Netlogic	**LA-7761 (MIPS64)** • XLR, XLS
NXP	**LA-7844 (Cortex-M)** • LPC11xx • EM773

New derivatives	
Ralink	**LA-7760 (MIPS32)** • RT3052, RT3662
Renesas	**LA-3777 (78K0R/RL78)** • 78K0R/Hx3/Lx3/Ix3 • 78F804x, 78F805x • RL78/G12, RL78/G13 **LA-3786 (RX)** • RX610/6108/621/62N/630
STMicro-electronics	**LA-7753 (MPC55xx/56xx)** • SPC560D/P, SPC56APxx • SPC564Axx, SPC56ELxx **LA-7844 (Cortex-M)** • STM32F100, STM32L15x
ST-Ericsson	**LA-7843 (Cortex-A/R)** • DB5500, DB8500
Tensilica	**LA-3760 (Xtensa)** • LX3
Texas Instruments	**LA-3713 (MSP430)** • MSP430xG461x • MSP430x20x1/x2/x3 **LA-7742 (ARM9)** • AM1707/1808/1810 **LA-7843 (Cortex-A/R)** • OMAP36xx **LA-7838 (TMS320C6x00)** • OMAP36xx
Toshiba	**LA-7742 (ARM9)** • TMPA900, TMPA910 **LA-7844 (Cortex-M)** • TMPM330, TMPM370
Trident	**LA-7760 (MIPS32)** • HiDTV PRO-QX
Wintegra	**LA-7760 (MIPS32)** • WinPath3, WinPath3-SL
Zoran	**LA-7760 (MIPS32)** • COACH 12

Tracing for Virtual Targets in Fast Models

Lauterbach has supported tracing for ARM Fast Models since November 2010.

To avoid having to wait for the first hardware prototypes before starting software development, software models of the hardware are often used. With Fast Models, ARM offers its customers a software package for programming models for ARM-based designs.

Since 2008, Lauterbach has supported the debugging of Fast Models over the CADI interface. It has now introduced support for the Model Trace Interface, which was introduced for Fast Models with Version 5.1. To prepare the trace information appropriately and buffer it in the virtual target, debugger manufacturers can load a separate trace plug-in. Fig. 2 shows an overview of the interplay of TRACE32 and Fast Models.

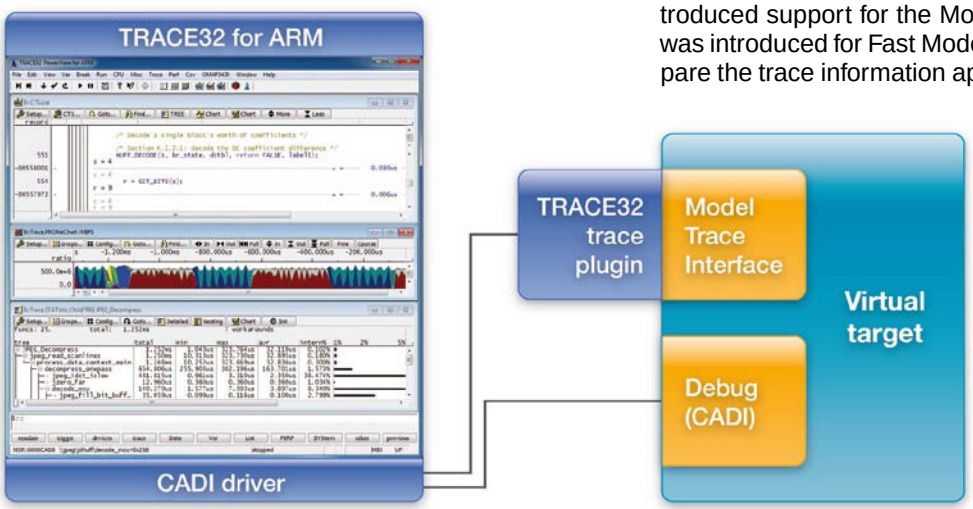


Fig. 2: TRACE32 supports the debugging and tracing of virtual targets.

For detailed information on debugging virtual targets, see:

www.lauterbach.com/frontend.html

API for VM Debugging Awareness

Since 2006, Lauterbach has supported the debugging of Java applications for the Java Virtual Machines J2ME CLDC, J2ME CDC and Kaffe. Since virtual machines are increasing in popularity, the number of providers is growing. Nowadays not all of these virtual machines are open-source. To enable VM providers and their customers to adapt debugging flexibly for their VM, Lauterbach has been working on a solution since mid-2010.

The Android Dalvik Virtual Machine implemented for ARM cores is used as a reference for the development of a VM API for stop-mode debugging.

Two Debug Worlds

For developers, Android is an open-source software stack consisting of the following components (see Fig. 3):

- A Linux kernel with its hardware drivers.
- Android Runtime with Dalvik Virtual Machine and a series of libraries: classic Java core libraries, Android-specific libraries, and libraries written in C/C++.
- Applications programmed in Java and their supporting Application Framework.

Software for Android is written in various languages:

- The Linux kernel, some libraries, and the Dalvik Virtual Machine are coded in C, C++, or Assembler.
- VM applications and their supporting Application Framework are programmed in Java.

Each block of code is tested in its own separate debug world.

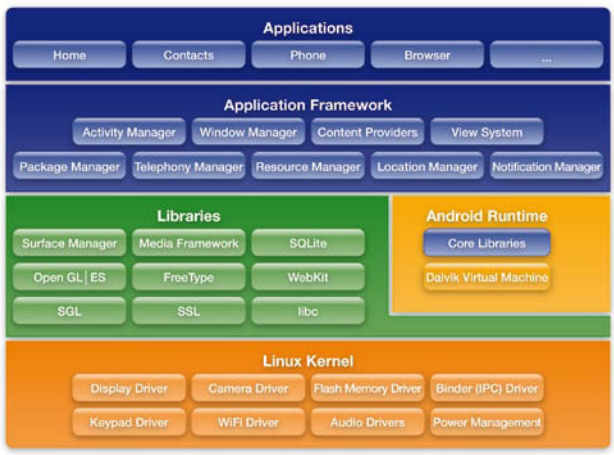


Fig. 3: The open-source Android software stack.

Debugging C/C++ and Assembler Code

The Android part coded in C/C++ and Assembler can be debugged on the target hardware over the JTAG interface in stop-mode. In stop-mode debugging, the TRACE32 debugger communicates directly with the processor of the Android hardware platform (see Fig. 4).

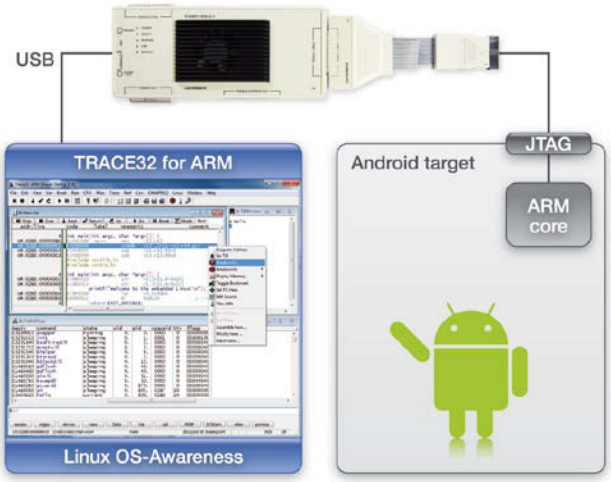


Fig. 4: In stop-mode debugging, the debugger communicates directly with the processor on the Android hardware platform.

A characteristic of stop-mode debugging is that when the processor is stopped for debugging, the whole Android system stops.

Stop-mode debugging has some big advantages:

- It needs only a functioning JTAG communication between the debugger and the processor.
- It needs no debug server on the target and is therefore very suitable for testing release software.
- It permits testing under real-time conditions and therefore enables efficient troubleshooting for problems that only occur in such conditions. »

At present, stop-mode debugging does not support the debugging of VM applications such as on the Dalvik VM. Therefore transparent debugging through all of the software layers is not yet possible.

Debugging Java Code

Java code for Android is usually tested with the Android Development Tools (ADT) integrated into Eclipse. The adb server – adb stands for Android Debug Bridge – on the host communicates over USB or Ethernet with the adb daemon on the target (Fig. 5).

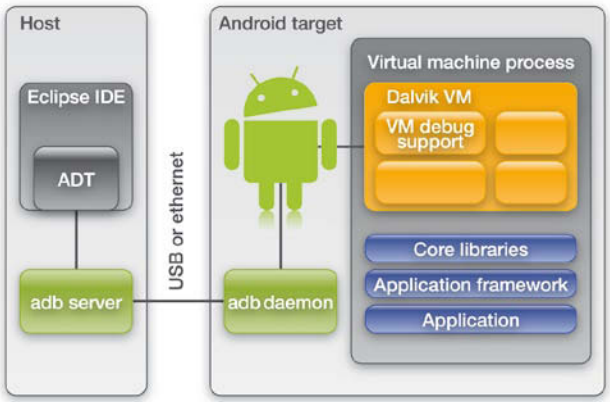


Fig. 5: The Android Development Tools (ADT) integrated in Eclipse for debugging Java code.

Prerequisites for debugging with ADT are VM applications specially compiled for debugging and Android debug support (adb daemon) running on the hardware platform.

Debugging Java code with ADT is comfortable. However, there are a few cases in which ADT cannot help you. These are:

- Errors that first occur with the release code.
- Errors that first occur when the Java application interacts with a service offered in C/C++ or a Linux hardware driver.
- Debugging following a communication breakdown between adb server and adb daemon.

VM Aware Stop-Mode Debugging

To enable thorough testing of an Android system from the Java application down to the Linux hardware driver under real-time conditions, Lauterbach is currently adding VM debugging awareness to its stop-mode debugging.

The JTAG debugger communicates directly with the processor on the Android hardware platform. The debugger can therefore access all system information after the processor stops. The “fine art” for the debugger is now to find the correct information and make it easy to understand for the user, abstracted from bits and bytes.

One abstraction level has given TRACE32 users the option of debugging operating system software even over several virtual address spaces. Another abstraction level, up to now independent of operating-system debugging, is Java debugging.

To debug applications running on VMs in systems like Android, where the VMs themselves are instantiated within the operating-system processes, operating-system debugging and Java debugging now have to be combined. To implement this new complexity, Lauterbach is developing a new, open, and easy-to-expand solution.

The Open Solution

In the future, stop-mode debugging from Lauterbach will support the following abstraction levels:

- High-level language debugging
- Target OS debugging awareness
- VM debugging awareness

High-level language debugging is a fixed component of the TRACE32 software and is configured for a program with the loading of the symbol and debug information. »

Dalvik Virtual Machine

Dalvik is the name of the virtual machine used in Android. The Dalvik Virtual Machine is a software model of a processor that executes byte code derived from Java. Virtual machines permit the writing of processor-independent software. If you switch to a new hardware platform, you only have to port the virtual machine.

Software compiled for a VM runs automatically on any platform to which this VM is ported.

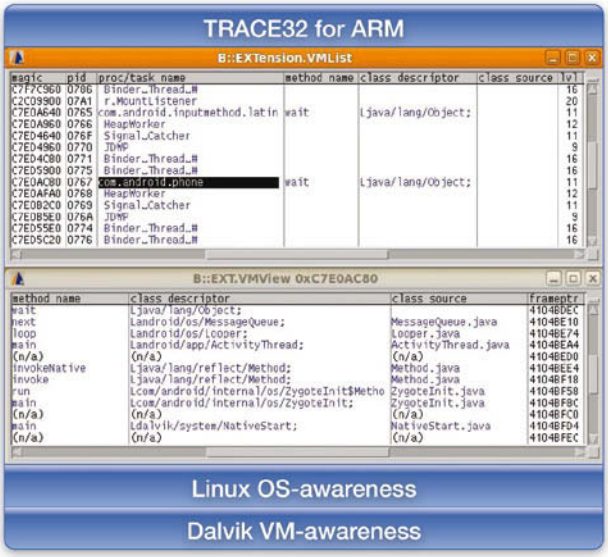


Fig. 6: For the reference implementation, Linux OS-awareness and Dalvik VM-awareness have to be loaded in TRACE32.

Target-OS debugging awareness must always be configured by the TRACE32 user. There are example configurations available for all common operating systems. The RTOS API provides an option to be customized for proprietary operating systems.

VM debugging awareness is a fixed component of the TRACE32 software for J2ME CLDC, J2ME CDC and Kaffe. All other virtual machines have to be adapted individually with the VM API. A ready-to-use configuration is available for the very popular Android Dalvik VM.

The open solution, both for the operating system and for the virtual machine, enables providers of closed-source VMs to write a TRACE32 VM awareness for their product and offer it to their customers.

The Reference Implementation

To be able to debug thoroughly on an ARM-based Android target from the Java applications right down to the Linux hardware drivers, TRACE32 requires the following extensions (see Fig. 6):

- A Linux OS-awareness as provided by Lauterbach since 1998.
- A Dalvik VM-awareness, which can be downloaded from the Lauterbach homepage. This just has to be configured for the platform used.

www.lauterbach.com/vmandroid.html

It is now possible to identify and list all Java applications now being run (EXTension.VMList in Fig. 6) and to analyze and view the VM stack for a selected Java application (EXTension.VMView in Fig. 6).

The next step planned is to display the source code currently being run by the VM. The aim of the development is of course stop-mode debugging for VM applications with all the functions of a modern debugger.

New Supported RTOS	
DSP/BIOS for ARM	Q2/2011
OSEK/ORTI SMP	Q2/2011
T-Kernel for ARM	available
Windows Embedded Compact 7 for ARM	available
µC/OS-III for ARM	available

Extensions and New RTOS Versions

- TRACE32 scripts were adapted for Timesys embedded Linux.
- OSEK/ORTI now ensures that NEXUS ownership trace messages are generated for task changes. This enables TRACE32 to make task-aware runtime measurements for the MPC55xx/MPC56xx, even if NEXUS generates no data trace messages.

The following version adaptations have been made or are planned:

- OSEck 4.0
- QNX 6.5.0
- Symbian^3 for ARM
- Symbian^4 planned for Q1/2011
- Windows CE6 for Atom™

Serial Trace Port Usage Growing

Faster, higher, stronger! Not only is this the motto of many sports – it has even been raised to a core principle in microelectronics. Ever faster clock speeds and a greater parallelization of processing steps have given us an astonishingly constant increase in processing speed for decades. It is no wonder that designers have also followed this motto for the transmission of trace information.

The trace interface, over which the processors deliver the detailed information on the operation of their inner processes, has struggled to keep up with the growing flood of information. For many developers of embedded systems it would be unthinkable to undertake a development without this important information, so all sorts of efforts have been made to increase the data throughput of the trace interface. For many years the increase in clock frequency and a greater bus-width at the trace port were an effective way of increasing data volumes.

However, these measures have their price. Not only does a wider trace port take up highly coveted package pins but poor signal quality at higher clock frequencies requires compensation on all signals from the trace bus. Thanks to the sophisticated algorithms of its AutoFocus technology, Lauterbach is able to ensure error-free recording of high-frequency trace signals.

As processor architectures continue to gain in speed and complexity through parallelization, the trace interfaces are starting to use a high-speed data transfer method that has been in use in other areas for a long time. A high-speed serial transmission is used in SATA, Fibre Channel, PCI Express, and USB3.0 (SuperSpeed USB). The extremely high data rates more than compensate for the disadvantage of only a few differential data lines.

The integration of high-speed serial interfaces on the chip is expensive and can initially cause problems. As just one example, the I/O pads have to be operated at a much higher speed. But with the increasing experience in the implementation of serial interfaces in the gigahertz range the knowledge gained can be used to solve many of the problems arising with the serial trace ports.

In 2008, ARM implemented this technology with its High Speed Serial Trace Port – HSSTP for short. This

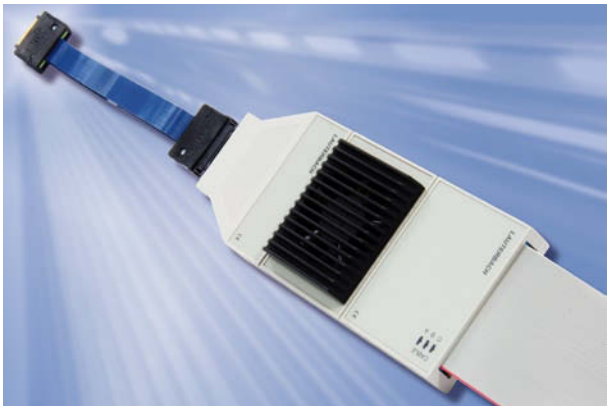


Fig. 7: Following firmware and software adaptations, a universal hardware supports the most varied protocols of serial trace interfaces.

was quickly followed by AMCC with the Titan, Freescale with the QorIQ processors P4040 and P4080, as well as Marvell with the SETM3.

Lauterbach had designed a hardware interface for the serial trace in 2008. A universal preprocessor was developed on the basis of the Aurora protocol. Only the firmware and software have to be changed to record any of the alternative protocols. This means that our system is already prepared for further variants of serial trace protocols.

Supported Serial Trace Ports		
AMCC	APM83290 Program flow	2009
ARM-HSSTP	ETMv3, PTM, CoreSight ETMv3, CoreSight PTM Program flow, Data flow and Context-ID	2008
Freescale	NEXUS QorIQ P4040 and P4080 Branch Trace and Ownership Trace Messages, Data Write Messages	2010
Marvell-SETM3	CoreSight ETMv3 Program flow, Data flow and Context-ID	2009

Higher Transmission Rate for Real-Time Streaming

"Real-time Streaming" means transferring trace data to the host whilst it is being recorded and analyzing it there immediately. This requires the transmission of large volumes of data from the trace tool to the host, especially for CPU-intensive applications and multicore systems. To make TRACE32 fit for these application scenarios, the trace data is compressed by the trace tool, PowerTrace II, before being transferred to the host. This feature has been supported by the TRACE32 software since December 2010.

Real-time Streaming is currently implemented for the ARM trace protocols ETMv3 and PTM.

Hardware Compression

The maximum transmission rate to the host is still the bottle-neck for *Real-time Streaming*. Even with a peer-to-peer GB Ethernet interface between the trace tool and the host, the maximum is currently only about 500 MBit/s net. This maximum transmission rate has to be sufficient to transfer all data at the trace port without loss to the host.

To be able to estimate the actual data volume to be transmitted, it is important to know the conditions of *Real-time Streaming:*

1. The main applications for *Real-time Streaming* are code coverage and run-time measurements. For both functions, it is sufficient if only the program trace information is exported. To get a very accurate run-time measurement, cycle-accurate tracing can be enabled.

2. For a realistic estimate of the necessary data rate, you just have to consider the average load at the trace port. Peak loads at the trace port are intercepted by PowerTrace II, which can be considered as a large FIFO (up to 4 GB). Fig. 8 shows an overview of the average/maximum load at the trace port for Cortex cores. The application running on the Cortex core ultimately determines the actual load.

By implementing FPGA-based hardware compression in PowerTrace II, the transmission rate to the host was raised to 3.2 GBit/s.

Pure Long-Time Trace

If trace data is analyzed and also saved to the hard-disk during *Real-time Streaming*, Lauterbach considers this a *Long-time Trace*.

To provide long-time tracing for other trace protocols such as Nexus, Lauterbach is now offering pure streaming onto the hard-disk without simultaneous analysis. This means that trace recording of up to 1 tera-frames is possible for a 64-bit host operating system.

For detailed information on *Real-time Streaming* and *Long-time Trace*, go to the Lauterbach homepage at: **www.lauterbach.com/tracesinks.html**

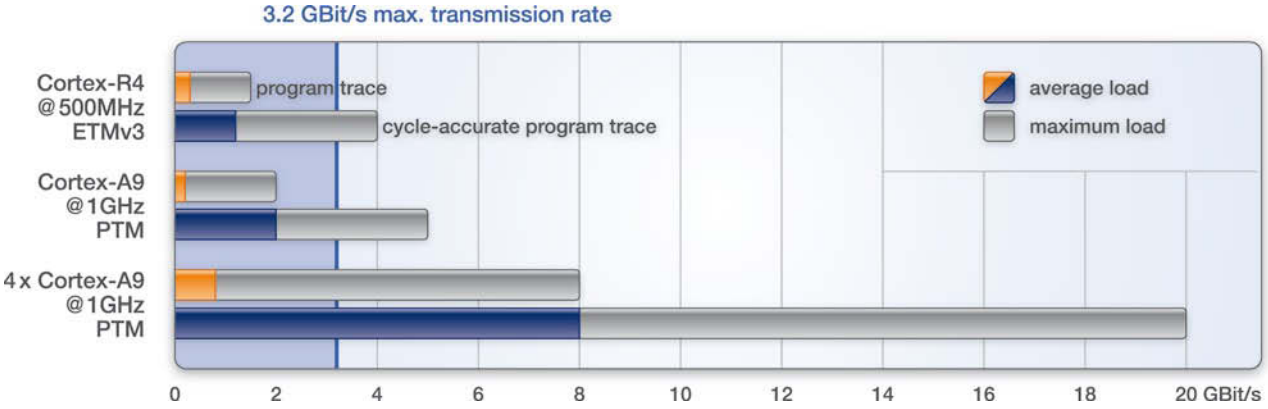


Fig. 8: A transmission rate of 3.2 GBit/s is usually enough to transfer program trace information to the host while it is being recorded.

Energy Profiling with the CombiProbe

The TRACE32 CombiProbe can now also be used for measuring the energy used by applications.

The following analyses are possible:

- The current/voltage profile at up to three measurement points can be displayed directly linked to the code running on the processor.
- The energy consumption of the entire system can be analysed for the individual functions.

Which part of a program uses the most energy? What influence does a program modification have on the energy requirements of an embedded system? These are the questions that can now be dealt with by the CombiProbe.

To determine the energy consumption for every point of the program, the following measurement data has to be collected:

- The program flow being exported via the trace port of the processor.
- The current and voltage profile measured at suitable measurement points on the target hardware.

The current and voltage development for up to three power domains can now be identified by connecting a *TRACE32 Analog Probe* to the CombiProbe.

CombiProbe

The CombiProbe is a debug cable that also contains a 128 MB trace memory. The CombiProbe was specially developed for processors with a 4-bit trace port. Program flow recording is currently supported for the following trace protocols:

- ARM-ETMv3 in continuous mode (ARM)
- IFLOW Trace for PIC32 (Microchip)
- MCDS Trace for X-GOLD102 and X-GOLD110 (Infineon)

www.lauterbach.com/cobstm.html

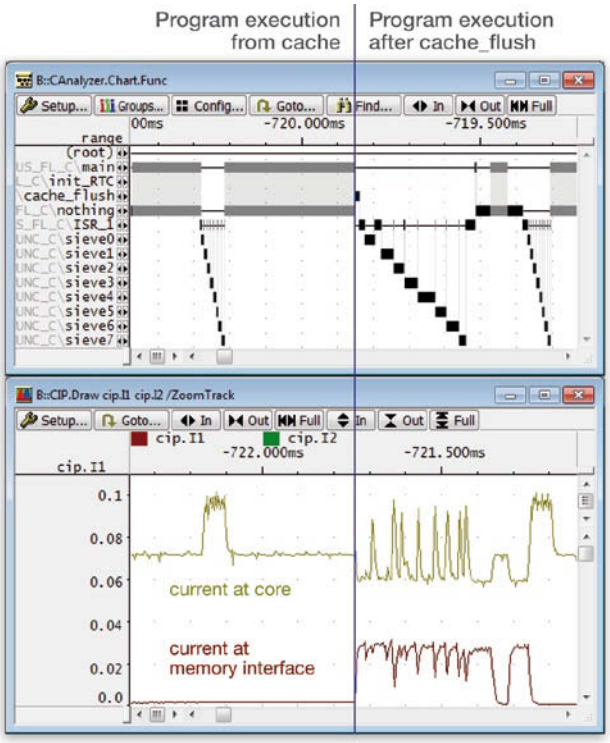


Fig. 9: A program section not running from the cache needs more time and uses more current.

Since all measurement data is time-stamped by the global timer of the CombiProbe, you can quickly and easily see the direct connection between executed program code and the power consumption as well as the voltage profile of the system.

Fig. 9 shows that a program section running from external memory instead of cache not only needs much more processing time but also uses more power at the external memory.

Fig. 10 shows the energy consumption as a statistical analysis.

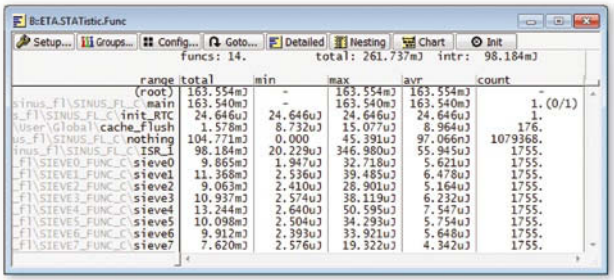


Fig. 10: The minimum, maximum, and average energy consumption of individual functions.

SMP Profiling

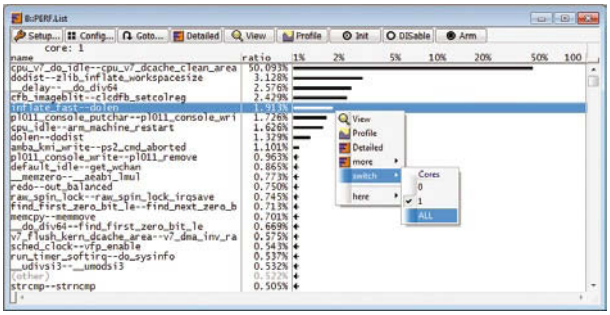


Fig. 11: Sample-based profiling shows the ratio of individual code sections compared to the overall run-time as a percentage. The result can be displayed both for the individual cores of the SMP system (here for core 1) as well as for the total of all cores.

Sample-based profiling was completely reworked in 2010. Important innovations include a new operating concept for measuring, a self-calibrating sampling rate and an extension for SMP systems.

SMP Profiling for Functions

From October 2010, profiling data for SMP systems can now be collected. To create function level profiling, TRACE32 cyclically reads the program counters of the individual cores and saves them in a database. The profiling can then be shown for the individual cores and also as a total for all cores.

Since many chips provide the capability of reading the program counter whilst the processor is executing, this measurement can be made in real-time for the following architectures:

- **ARM / Cortex:** ARM11 MPCore, Cortex-A5 MPCore, Cortex-A9 MPCore, Cortex-A15 MPCore

Sample-Based Profiling

For Sample-based Profiling, the program counter or the variable containing the ID of the current task is periodically read. On the basis of this information, the ratio of a function or task compared to the overall run-time is shown as a percentage.

Symmetrical Multiprocessing (SMP)

A multicore chip consisting of identical cores can be configured as an SMP system. An SMP operating system distributes the pending processes (tasks) dynamically to individual cores at program run-time (but not before). For debugging SMP systems, a single TRACE32 instance is opened from which all cores are monitored.

- **MIPS32:** MIPS34K, MIPS1004K
- **MIPS64:** Broadcom BCM7420

If the chip's debug logic does not permit non-intrusive reading of this information, the individual cores will have to be briefly stopped periodically to obtain this information.

SMP Profiling for Tasks

To create a task profile, the task ID for the individual cores has to be read cyclically from the memory. Many chips allow the physical memory to be read at program run-time. If the on-chip debug logic supports this feature, the measurement can be made in real-time:

- **ARM / Cortex:** ARM11 MPCore, Cortex-A5 MPCore, Cortex-A9 MPCore, Cortex-A15 MPCore
- **Power Architecture:** MPC8641D, MPC8572, QorIQ

Otherwise, the individual cores have to be briefly stopped to read the data required.

KEEP US INFORMED

If your address has changed or if you no longer want to be on our mailing list, please send us an e-mail to info_us@lauterbach.com.

WORLDWIDE BRANCHES

- **USA**
- Germany
- France
- UK
- Italy
- China
- Japan

Represented by experienced partners in all other countries

Three things you must do each time you start writing your code: initialize variables before use, do not ignore compiler warnings, check return values.

From this and other experiences, I've come up with a set of 17 tips summarizing the lessons I've learned as a software engineer in the embedded systems environment, particularly as they relate to C programming, as a way to help others avoid the same potholes I encountered. In the process, I had a lot of help, particularly with the many tips on safe C in articles at EmbeddedGurus.com and Embedded.com.

TIP #1—FOLLOW THE RULES YOU'VE READ A HUNDRED TIMES

There are three things you must do each time you start writing your code. You've read these rules many times before and resolved to do them the next time you started code development. This time, do them—they will help you to avoid many long hours of debugging:

- Initialize variables before use.
- Do not ignore compiler warnings.
- Check return values.

Accessing objects before they have a defined state can lead to strange effects. Not only does avoiding these effects require that you make sure you've set all your `int`s and `float`s to a defined state, you also have to make sure that your complex type functions, such as `typedefed structs`, are initialized first.

For instance, declare an object like the one in **Listing 1** in the header. In Listing 1, the object has the typical `init` flag and two function pointers for reading and updating data.

Then, in the C module, initialize the object to a level for first usage. In this case, the `init` flag was set to false. The variable is static here, since I have only one instance of it:

```
static symbol_model_t
g_symbol_model =
{
  false,/* is initialized */
  NULL, /* update function */
  NULL  /* read function */
};
```

Later on during construction of

Listing 1 Declaring an object in the header, initializing it with an init flag, and describing it with two function pointers.

```
typedef struct
{
    bool is_initialized;
    symb_model_error_t ( *update ) ( display_data_t const * const hud_data );
    symb_model_error_t ( *read )   ( display_data_t * const hud_data );
} symbol_model_t;
```

Listing 2 Check if objects were not initialized.

```
symbol_model_t * const symbol_model_instance ( void )
{
     if ( g_symbol_model.is_initialized == false )
     {
          g_symbol_model.update = symbol_model_update;
          g_symbol_model.read = symbol_model_read;

          g_symbol_model.is_initialized = true;
     }

     return &g_symbol_model;
}
```

In programming, it's safer to assume that the failure is not the exception in a string of successes but exactly the opposite.

the object, you can check if things were not initialized, such as shown in **Listing 2**.

Even though they finally compile the code, modern compilers are always complaining about strange constructs. Do not ignore these complaints. More often than not, they are right. Also do not ignore return values since they indicate the first time something has gone wrong. If you ignore such warnings, you'll have a ticking time bomb in your system that will explode at a later point.

If you follow these procedures, you'll have more time left at the end of the project. After all, the end of the project is the point at which money and time are running out and people are overstressed, especially if the product isn't working and is shipping late. Now you'll have time to help them set things right.

TIP #2—USE ENUMS AS ERROR TYPES

Every module should have a specific error return type that explains what the problem is in detail, at the time it occurs. Often you receive error codes like "-1" or "an error occurred." If there is a run-time error detected and you know exactly what it is, document this for your later reference and for those who maintain the software. For example, consider the code in **Listing 3**.

Such an error type already has been decoded and the cause determined. The last entry, called `<XYZ>_LAST_ERROR`, makes it possible to iterate over the content of the `enum`. That means you only have to know what the first element in the chain is. No matter how many more errors you add between the first and the last, all you have to do is check the range or iterate. Also, this last `enum` value gives you the total number of entries. More on this later.

TIP #3—EXPECT TO FAIL

Failures happen. Often. So plan for it and use it to your advantage. It's good practice to set the default return value of an operation to something like `UNNOWN_ERROR`. Only in the case of a

Name
Dr. Dennis Hong

Job Title
Associate Professor of Mechanical Engineering, Virginia Tech

Area of Expertise
Robotics

LabVIEW Helped Me
Convey and respond to vast amounts of data in real time

Latest Project
Design and prototype a car that can be driven by the blind in just 4 months

NI LabVIEW

LabVIEW makes me better because

CODE REUSE

saves time and effort

>> Find out how LabVIEW can make you better at **ni.com/labview/better**

800 453 6202

NATIONAL INSTRUMENTS™

WHAT IS TYPE SAFETY

I generally define type safety as Wikipedia defines it: *"In computer science, type safety is the extent to which a programming language discourages or prevents type errors. A type error is erroneous or undesirable program behaviour caused by a discrepancy between differing data types."**

If you define a type, this means nothing more than saying there are a certain number of bits that represent a predefined data type. For instance, `uint32_t number_of_bytes` defines 32 bits, which hold an unsigned scalar value ranging from 0 to 4.294.967.295. If you would assign a negative value to the `number_of_bytes` variable, a type-safe language would raise an exception during run time or a compile-time error at compile time. The Ada language does this, but C does not. In C, you could also assign a floating-point value like 3.456 to the variable, which makes some compilers complain at compile time and produces undefined behavior during run time.

*Wikipedia entry on type safety, from April 20, 2011. http://en.wikipedia.org/wiki/Type_safety

Listing 3 The last entry <XYZ>_LAST_ERROR makes it possible to iterate over the content of the enum. No matter how many more errors you add between the first and the last, all you have to do is check the range or iterate.

```
typedef enum
{
     SYMB_MODEL_SUCCESS = 0,
     SYMB_MODEL_UNKNOWN_ERROR,
     SYMB_MODEL_NOT_INITIALIZED_ERROR,
     SYMB_MODEL_INITIALIZATION_ERROR,
     SYMB_MODEL_SYSTEM_UNAVAILABLE,
     SYMB_MODEL_WRONG_INPUT_VALUE,

     /* the last entry is just for looping over the enum */
     SYMB_MODEL_LAST_ERROR
} symb_model_error_t;
```

This pessimistic approach is safer than expecting all things to go well.

good result should you set it to SUCCESS, for instance, as in **Listing 4**.

This pessimistic approach is safer than expecting all things to go well and setting the default to _SUCCESS. In programming, it's safer to assume that the failure is not the exception in a string of successes but exactly the opposite: The good case is the only exception in a string of more commonly occurring error cases.

TIP #4—CHECK INPUT VALUES: NEVER TRUST A STRANGER

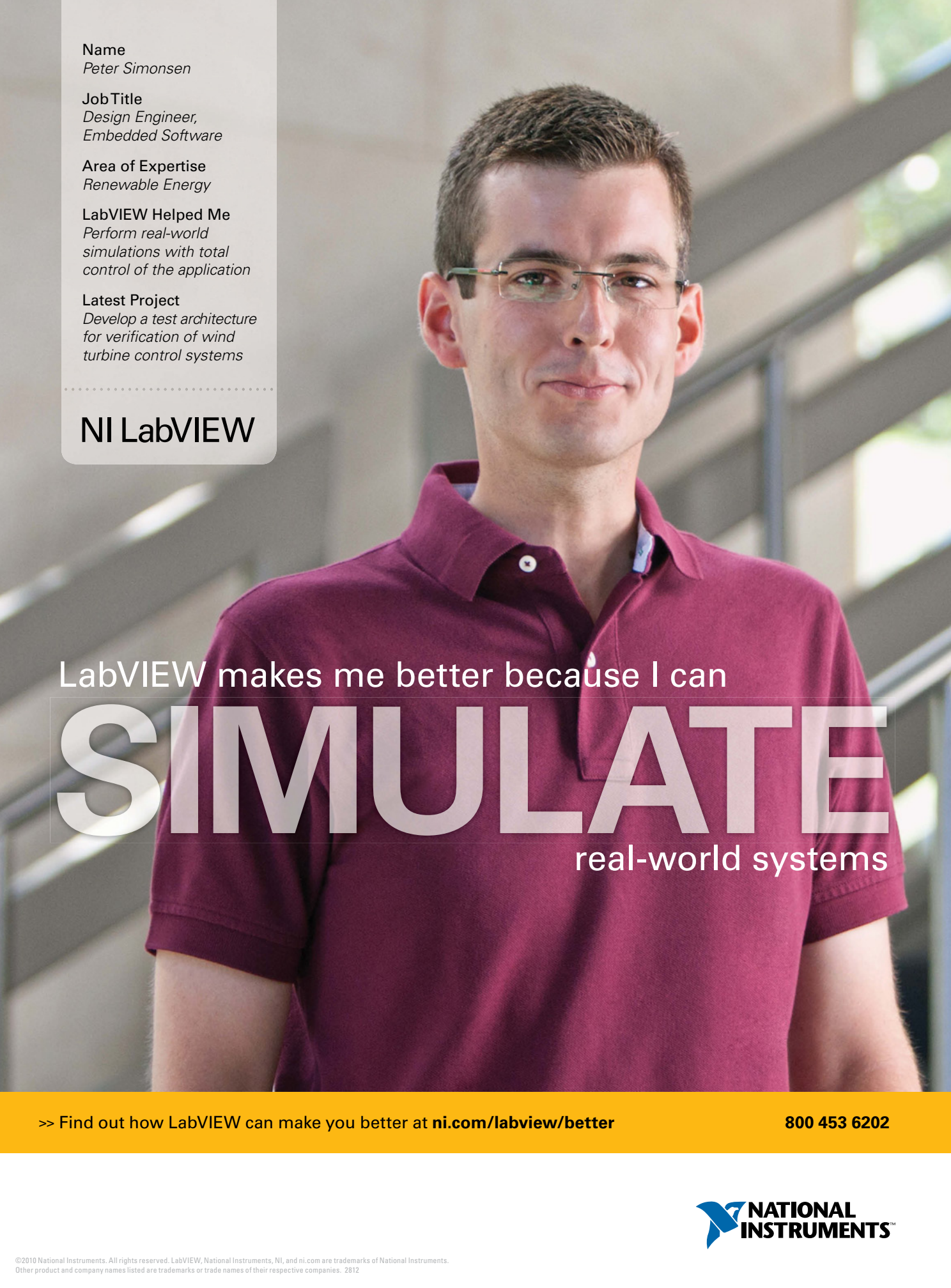
If your modules expect input data from other modules, you should never trust a stranger. That is, at the outmost layer of your software architecture, check all input values for consistency. The check has to be at the outmost layer since it must be detected as soon as possible. Otherwise you could, for instance, dereference an invalid pointer given to you at one of your lower layers. The result: The crash dump reports that it was your software's problem, but later, after many hours of debugging, you find out that someone has given you invalid input.

Here is an example using an enum error type mapped to a string represen-

Listing 4 Only in the case of a good result should you set the default return value of an operation to SUCCESS.

```
symb_model_error_t symbol_model_read (display_data_t * const data )
{
    symb_model_error_t status = SYMB_MODEL_UNKNOWN_ERROR;
    if ( data != NULL )
    {
        memcpy ( data, &g_display_data, sizeof ( display_data_t ) );
        status = SYMB_MODEL_SUCCESS;
    }
    else
    {
        status = SYMB_MODEL_WRONG_INPUT_VALUE;
    }

    return status;
}
```



>> Find out how LabVIEW can make you better at **ni.com/labview/better** **800 453 6202**

NATIONAL INSTRUMENTS™

Listing 5 An example using an `enum` error type mapped to a string representation for trace output to a console window.

```
typedef enum
{
     SYMBOL_VIEW_SCREEN_OFF = 0,
     SYMBOL_VIEW_SCREEN_PILOT,
     SYMBOL_VIEW_SCREEN_NO_DATA,

     /* the last entry is just for looping over the enum */
     SYMBOL_VIEW_SCREEN_LAST
} symb_view_screen_t;
```

Listing 6 The lookup table representing the strings is defined.

```
static const char *
symb_view_screen_string_map[SYMBOL_VIEW_SCREEN_LAST] =
{
     “SYMBOL_VIEW_SCREEN_OFF”,
     “SYMBOL_VIEW_SCREEN_PILOT”,
     “SYMBOL_VIEW_SCREEN_NO_DATA”
};
```

tation for trace output to a console window, shown in **Listing 5**.

The lookup table representing the strings is defined as shown in **Listing 6**. Safe access to the string map that does not allow any out of bounds access is shown in **Listing 7**.

Unfortunately `enums` in C are integers. That means you could hand over any value of integer to the interface accessing the array, an error that can be avoided.

By the way, if you define the lookup table with the `_LAST` enum as a size parameter, it will have the right size and keep you from indexing out of bounds. Also getting the string out of the string array is a very simple offset addressing operation, which is really fast in C.

So, that was range checking. You should also check for `NULL` pointers if someone gives you an address value. You cannot check pointers for anything other than the `NULL` value, but this is better than nothing.

TIP #5—WRITE ONCE, READ MANY TIMES

When we read other people's code, we're thankful for any good line of comment or more readable code; most of the time, however, the original coder hasn't been so kind to us. If the variables are called *i*, *j*, and *k*, you'll soon have a mental break down. Often the longest variable name is *pbuf*. What can happen when code is difficult to decipher is that even though the next programmer should only slightly change the software, he or she says, "I can't understand this hacker's code. It will be faster to rewrite it." The rewrite results in extra work and possibly new bugs.

So what can you do? First of all, if you write code, write it to be as readable as a newspaper. Well-written code requires only a few lines of comments. Also consider that although code is nothing for compilers, it needs to be readable by human beings.

Don't be lazy at typing new variable names and, if required, add the unit to the name. For example, do not call parameters `Size`, `Length`, `Temperature`, or `Angle`. Instead, since all those parameters have a unit, call them:

- `number_of_bytes`
- `length_in_meters`
- `temperature_in_celsius`
- `angle_in_radians`

Listing 7 Safe access to the string map that does not allow any out of bounds access.

```
const char * symbol_view_screen_name ( const symb_view_screen_t screen )
{
     const char * screen_name = “”;

     /* check array index */
     if ( ( screen >= SYMBOL_VIEW_SCREEN_OFF ) && ( screen < SYMBOL_VIEW_SCREEN_LAST ) )
     {
          screen_name = symb_view_screen_string_map[screen];
     }

     return screen_name;
}
```

Name
Dr. Laurel Watts

Job Title
Principal Software Engineer

Area of Expertise
Chemical Engineering

LabVIEW Helped Me
Control multiple instruments operating in harsh conditions

Latest Project
Engineer the ultimate storm chaser

NI LabVIEW

LabVIEW makes me better because the

INTEGRATION

with hardware is so seamless

>> Find out how LabVIEW can make you better at **ni.com/labview/better**

800 453 6202

NATIONAL INSTRUMENTS™

Listing 8 Memory map of an embedded system.

```
/*--------------------
   Memory map of the project XYZ
--------------------

     0000_0000  +----------------+
                |          APSW Code RAM          |
                |      16 MByte = I-/D-BAT0       |
                |       I-/D-Cache enabled        |
     00FF_FFFF  |         User/Supervisor         |
     0100_0000  +----------------+
                |             BSP RAM             |
                |        16 MByte = D-BAT1        |
                |         D-Cache enabled         |
     01FF_FFFF  |           Supervisor            |
     0200_0000  +----------------+
                |           General RAM           |
                |             96 MByte            |
                |    D-BAT2[16MB]+D-BAT3[64MB]    |
                |         D-Cache Enabled         |
     07FF_FFFF  |         User/Supervisor         |
     0800_0000  +----------------+
                |                                 |
                |            Reserved             |
     1FFF_FFFF  |                                 |
     2000_0000  +----------------+
                |      Host bridge Registers      |
                |    516 kByte = D-BAT4[1MB]      |
                |        Cache Inhibited          |
     200F_FFFF  |           Supervisor            |
     2010_0000  +----------------+
                |                                 |
                |            Reserved             |
     7FFF_FFFF  |                                 |
     8000_0000  +----------------+
                |          PCI Interface          |
                |       256 MByte = D-BAT5        |
                |        Cache Inhibited          |
     8FFF_FFFF  |        User/Supervisor          |
     9000_0000  +----------------+
                |                                 |
                |            Reserved             |
     FBFF_FFFF  |                                 |
     FC00_0000  +----------------+
                |           Boot Flash            |
                |        64 MByte = D-BAT6        |
                |    1 MByte (FFF...) = I-BAT6    |
     FFF0_0000  + - -  I-/D-Cache enabled  - -    +
                |           Supervisor            |
     FFFF_FFFF  +----------------+
*/
```

There are famous examples of errors coming from wrong unit conversions, such as the loss of a Mars climate orbiter (see *http://mars.jpl.nasa.gov/msp98/news/mco990930.html*). Not using units in your application programming interface's definitions can also cause major design failures. See *How To Design A Good API and Why it Matters* (Joshua Bloch's Google TechTalks video from 2007) at *www.youtube.com/watch?v=aAb7hSCtvGw*.

If you've written code that requires some renaming, I recommend the use of the open-source Eclipse Development Environment (*www.eclipse.org*). It has a great feature called Refactoring that renames any kind of object everywhere in the code. For instance if you want to change a function parameter's name from `number_of_bytes` to `number_of_floats`, just mark it, press ALT-SHIFT-R, and change the name.

Documenting the source code is helpful not only for your future reference but for those who come after you. For instance, if you're working on an embedded system, you need to have a memory map indicating where all the memory-mapped devices can be found. **Listing 8** shows an example of a memory map.

It's useful to have diagrams of all the software layers in your application as well as diagrams of the overall software architecture, preparing them in a format that allows you to simply cut and paste them to a word processing program. Remember that if you write it down, you don't have to keep it in mind.

TIP #6—WHEN IN DOUBT, LEAVE IT OUT

I've already mentioned the API design tutorial from Joshua Bloch, a guru in the Java community. He brings up a good point in his API-design tutorial on YouTube (URL mentioned earlier).

And that is: If you design an API that is nothing other than the external interface of your modules, consider the need of an operation. If you are not sure anyone will ever need an operation, leave it out. If someone does use your API and you later remove an operation, you'll break his code. So Josh says, "When in doubt, leave it out. You can always add, but you can never remove."

TIP #7—USE THE RIGHT TOOLS

Everyone has a favorite editor, debugger, and compiler. But sometimes it's worth looking for something new since "the better is the enemy of the good." Here is what I use (many of which you may already use):

- **Eclipse:** Has a good editor, is good at refactoring code, also good for prototyping architectures on the PC (for example, with Cygwin on Windows). *www.eclipse.org*.
- **Astyle:** Artistic Style 2.01 is a great code formatter that can be configured in many ways to beautify the code. *http://astyle.sourceforge.net/*.
- **Cygwin:** For PC-based prototypes and for architectural studies, you can use the GNU tool chain of cygwin. Make sure you install the make, binutils, and gcc from the development package. *www.cygwin.com/*.
- **GNU tool suite:** Many embedded systems tool chains use this set of tools. Even if you don't have hardware at the beginning of the project (your hardware developers may not have finished their work), you can start writing prototypes for your architecture. Eclipse together with Cygwin using the GNU tools is worth trying. *www.gnu.org*.
- **Tortoise SVN:** This is a nice add-on for Windows Explorer to access the subversion versioning system. *http://tortoisesvn.tigris.org/*.

All these software packages are available for free.

TIP #8—DEFINE THE SOFTWARE REQUIREMENTS FIRST

Defining the requirements for the software you write is the first step for a successful product. I mean the software requirements for the final product, not those for the quick hacked throwaway prototype you're working on as a first step to the final product. And this, of course, requires defining the goal to be reached.

If you don't define the requirements, you can't test your final software properly—you'll have nothing to use to define a useful test case. In other words, how can you determine if you've finished the development? Here's a helpful multiple-choice software quiz along these lines:

How can you determine if you have finished the development?

1. There is no more money left;
2. There is no more time left; or
3. All the requirements are implemented *and* tested successfully.

To properly finish development (you should all know which answer is the correct one above), I define the following on every project:

1. **Requirements for the OK case**—that is, what is required to fulfill the main functionality.
2. **Requirements for the ERROR case**, important for the safety-critical design since you also need to define what has to be done if things go wrong. Remember that the good case is often the exception in a string of more commonly occurring error cases.
3. **Tests that check if the above defined requirements are implemented correctly.**

If you do testing on the code base directly (white-box testing), you may tend to test what the code does and not what it's supposed to do. Here's where requirement-based testing can improve your end product: You're forced to do black-box testing.

TIP #9—DURING BOOT PHASE, DUMP ALL AVAILABLE VERSIONS

If you're the one who implements the boot loader on new hardware, you would normally do the following:

- Initialize the hardware according to the required memory map.
- Execute a hardware self test.
- Start booting the application.

Nothing new here: That is what your PC typically does every time you boot up.

But in embedded systems development in the era of FPGAs and CPLDs, the hardware is as modifiable and subject to change as the software. Dump all programmable logic devices version registers onto a console window or file before starting the application. This step is important since hardware developers nowadays use programmable devices to quickly change the behavior of their hardware, with the result that VHDL code can be changed as fast as software code can be changed.

To circumvent such messages as "this error is only on your system" or "we cannot reproduce the problem," you should dump at least all the version registers of the hardware devices. Also your software should say what version it is. For a real product, there must be a matrix telling you which hardware works with what software version.

If you do this, you'll find out that often people are operating illegal combinations that can cause some super-strange effects. Such versioning information is very helpful for your product hotline or test staff, as well as to production people who can use it to check if what they've produced is the right configuration.

TIP #10—USE A SOFTWARE VERSION STRING FOR EVERY RELEASE

If you've finished development of a particular stage in a project in order to do tests on it or to release a software version, be sure you take the following steps in exactly the order written:

1. Update the version string and date.
2. Check the software version in to your versioning system.
3. Update the version string right after check-in for the next version.
4. Test the software.
5. Fix the bugs.
6. Continue developing the next version.

The most important step is 3.

I know of several instances where developers have given their software to testers or customers without incrementing the software version string. The result: Several software versions were out there with the same version string. It can take days before you realize this in-

Listing 9 An example of a wheel reinvented. Don't be that bore who reinvents the int and boolean wheels. Use standards instead.

```
typedef enum
{
    MY_GREAT_BOOLEN_FALSE    = 0,
    MY_GREAT_BOOLEN_TRUE
} my_great_boolean_t;

my_great_boolean_t is_wheel_re_invented = MY_GREAT_BOOLEN_TRUE;
```

consistency and resolve it. (By the way, in the era of programmable devices, this also applies to the hardware engineers. So, if you meet some of them in the breakroom, remind them).

TIP #11—DESIGN FOR REUSE: USE STANDARDS

Don't try to reinvent the wheel, believing your wheel will be better than all the millions that have already been invented. I've read so many times things like **Listing 9**.

Since the C99 language standard has defined the `stdbool.h` and `stdint.h` headers, things have become portable and there is absolutely no need to define your own `int` or `boolean types`.

TIP #12—EXPOSE ONLY WHAT IS NEEDED

When I read other programmers' code, I wonder if they've ever heard about "information hiding." I find many externally declared variables that can be accessed from several modules. The practice is both pointless and sometimes dangerous.

Module internal operations and variables are often not declared `static`, which allows them to be accessible from other modules. This accessibility results in a design that is not modular because when operations and variables are not declared `static`, they're interdependent and not modular (since one thing cannot live without the other).

Also C doesn't have any syntax for anything like namespaces, common in other object-oriented languages. Or to be more precise, C knows only one, the global namespace. This means that all nonstatic operations or variables are visible globally unless you hide them. This global visibility could result—and often does—in a name clash detected at linking the software. As long as you have all the source code for the project, you can easily resolve this issue. If you have only a library in binary format and some function headers, the situation is more complicated.

Don't reinvent the wheel, believing your wheel will be better than all the millions already invented.

Another related topic: Parameters have to be declared as `const` if the implementer of the interface doesn't want this object to be changed. The difference between C and C++ is that in C++ `const` means constant, whereas C defines constant to be interpreted as read-only. To illustrate this, **Listing 10** shows a declaration of a read operation reading data into a specified buffer called `display_data`, which is at a constant address.

A write operation that is creating constant data located at a constant buffer address requires `const` two times, shown in **Listing 11**.

If you later try to modify constant objects, your compiler will correct you.

You should let your compiler help you develop your software in a safe way. What you need to do is to provide the compiler the information on how the objects are to be treated.

TIP #13—MAKE SURE YOU'VE USED "VOLATILE" CORRECTLY

In embedded software development you sometimes have to do things that your host-based colleagues are often not concerned about. One of those things is declaring variables to be volatile, which keeps the compiler from optimizing read or write operations for this variable. How to do this is well described by Michael Barr's blog posting "Firmware-Specific Bug #3: Missing Volatile Keyword" found at *http://embeddedgurus.com/barr-code/2010/02/firmware-specific-bug-3-missing-volatile-keyword/*

TIP #14—DON'T START WITH OPTIMIZATION AS THE GOAL

Some developers are intent on writing "fast code," even though they cannot define what "fast" means in the context of their application. It sounds good as an objective, but what I've seen is that often under the cover of writing fast code, they want to move beyond the existing sys-

Listing 10 A declaration of a read operation reading data into a specified buffer called `display_data`, which is at a constant address.

```
display_error_t display_data_read ( display_data_t * const display_data );
```

Listing 11 A write operation that is creating constant data located at a constant buffer address requires const two times.

```
display_error_t display_data_write ( const display_data_t * const display_data );
```

tem definition and move to a nonexisting architecture.

To account for this and other system-redefining goals, you should think seriously about developing a flexible architecture capable of adapting to various exigencies. First, this means developing a set of software requirements for the product being planned. Then you should assume that once developed, your software architecture will no doubt be extended, so consider how you can design it to be flexible enough to incorporate new features into the existing platform without scrapping the code base you've already developed.

If you're concerned about performance, wait until your project's first integration phase, at which point you can determine how fast the system is. This doesn't have to be the last milestone in the project. As proof of concept, you can plan to produce several interim "proof of concept" implementations and measure performance. Working from that known value, consider what you need to do to achieve the necessary performance goals.

If you then detect that your code is not fast enough, you have to check which parts are responsible for the main time consumption. Then you profile the software and determine what loops and routines are consuming all the time.

The rule with optimization is that you first have to know where you are before considering what you need to optimize in order to get where you want to be. And don't forget that the software must still be maintainable.

TIP #15—DON'T WRITE COMPLEX CODE

Complex code is error-prone code. I think most of you already know this. But the question is, what does complex really mean in the context of your particular design?

A good metric for defining this is the McCabe or cyclomatic complexity algorithm. See also this well-written article by Jack Ganssle ("Taming software complexity," Embedded.com, 2008) at *www.eetimes.com/4007519*.

My opinion is that if you write safety-critical code, the best rules are:

- Max. cyclomatic complexity per function: 10
- In a few exceptions, such as use of switch-case constructs: 15

Many good tools exist for measuring the complexity of your code. "Understand for C," at *www.scitools.com* is a good tool, but many others are available.

You shouldn't just think about writing code during the initial software development phase. You need to think about all the stages of the code's life.

You need to think about all the stages of the code's life. Do not just think about your own needs: many others are coming after you.

The code of a product will be changed and extended many times during the product's life cycle. And the people who have to do this are pretty often not the ones who have written the code initially. In other words, do not just think about your own needs: many others are coming after you.

TIP #16—USE A STATIC CODE CHECKER

If you write safety-critical code, you surely have a coding guideline. Even if your guideline contains only 10 rules, you must have a tool to help you check those rules. If your team doesn't have a tool for checking, you can be sure that things won't be checked.

Many tools, such as PC-Lint, are available to accomplish this task. You should check your code at every significant milestone in your project to be sure that the code quality is good.

Remember, software testing is a multistage process, of which static code checking is one part. The other stages include:

- Functional tests.
- Requirements-based tests.
- Coverage tests (such as MC/DC).

All these tests have one general purpose: to reduce the number of bugs in your code.

No software code base exists on this planet that can be considered to be error-free. But there are many good software code-base products that do what they're supposed to do. Unfortunately, there are also many that do not.

TIP #17—MYTHS AND SAGAS

Many myths and sagas persist in the world of safety-critical embedded systems. One of the most common is that dynamic memory allocation is forbidden. This myth, however, is only half of the truth. Every application has an initialization phase. This phase is followed by the operational one. It's no risk at all to do dynamic memory allocation during the initialization phase. But to avoid memory fragmentation, during the operational phase, you aren't allowed to change those allocations.

AVOID THE POTHOLES

With this article, I've identified some potholes on the road to safety-critical software development and how you can avoid them. When you come to the part of your job where you tell a computer what you want it to do, I hope these tips will be helpful. At that time, remember this one sentence summary I came across once about our common job domain: "Computers have the strange habit of doing what you say, not what you mean." ■

Thomas Honold is a software architecture designer, specializing in safety-critical DO-178B software development in the defence/aerospace industry. He has a master in electronic engineering and has worked 15 years on software architectures and design for banking software, Internet banking, chip-card readers, avionics, and bootloader driver software.

Do you really need that CPU in your microcontroller? Here's a way to free up your CPU using a combination of programmable logic devices and datapaths. Mark Ainsworth of Cypress Semiconductor explains how.

Why your embedded controller may not need a CPU

BY MARK AINSWORTH

In most microcontroller architectures, a "smart" CPU is surrounded by a set of relatively "dumb" peripherals. The peripherals have limited functions; usually they just convert data from one form to another. For example, an I²C peripheral basically converts data between serial and parallel formats, while an ADC converts signals between analog and digital. The CPU has to perform all of the work to process the data and actually do something useful with it. This, plus close management of the peripherals, can result in a lot of complexity in the CPU's firmware and may require a fast and powerful CPU to execute that firmware within real-time timing constraints. This in turn can lead to more obscure bugs and thus to more complex and expensive debugging equipment, and so on.

But what if the peripherals were complex enough, flexible enough, and ultimately "smart" enough to effectively relieve the CPU of many of its tasks? A complex design could then be re-

Example PLD with 12 inputs, eight product terms, and four macrocells.

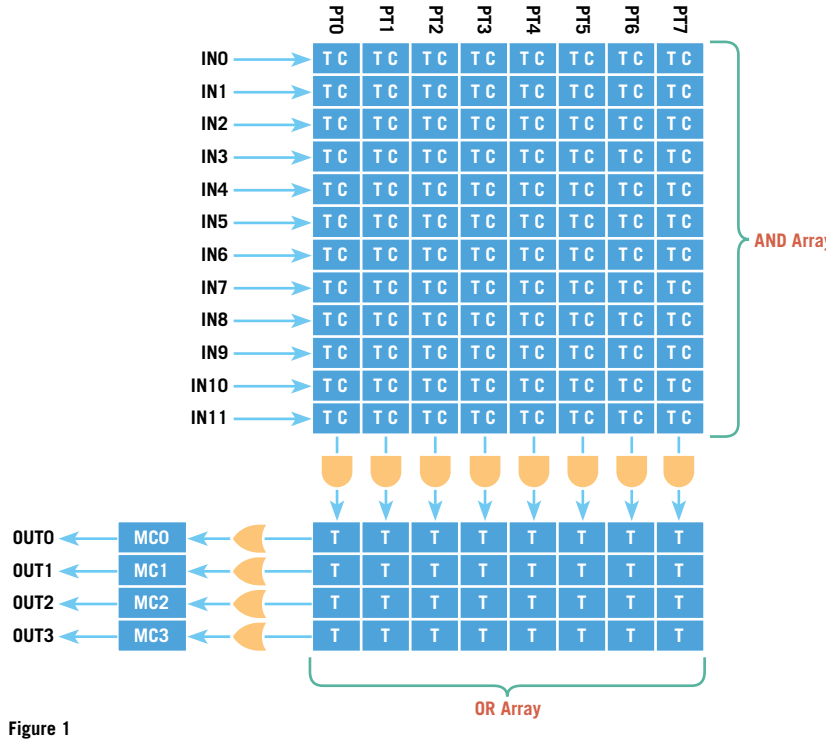


Figure 1

structured as a group of simple designs distributed among the CPU and the peripherals. The CPU would ultimately have fewer tasks and perhaps fewer interrupts to handle, in turn making bugs easier to find and fix. The overall design would become more robust, and portions of the design more easily reused. Finally, a CPU with less to do may be run at a slower speed to save power, or that available bandwidth could be used for the additional tasks that the marketing department dreams up for the next-generation product. However, the peripherals would still need to be designed in a cost-effective manner or the overall microcontroller might become too expensive. This article shows how a set of smart, flexible, low-cost, custom digital peripherals can be designed into a microcontroller and configured to help implement a robust distributed system design.

SMART LOGIC OPTIONS—PLD OR DATAPATH?

There are two general ways to construct a smart configurable peripheral. The first is to use a programmable logic device (PLD). As shown in **Figure 1**, a PLD has a sum-of-products logic gate array driving a number of macrocells. The "T" and "C" notations indicate that each product term can generate either a true or complement (inverted) output, so that both positive and negative logic can be supported.

Figure 1 shows a simple example of a PLD. PLDs can have hundreds of macrocells with up to 16 product terms driving each macrocell. The AND and OR gates within the product terms can be interconnected to form highly-flexible

A simple example of an ALU-based datapath, with accumulators, function select, clock, and carry and shift chaining signals.

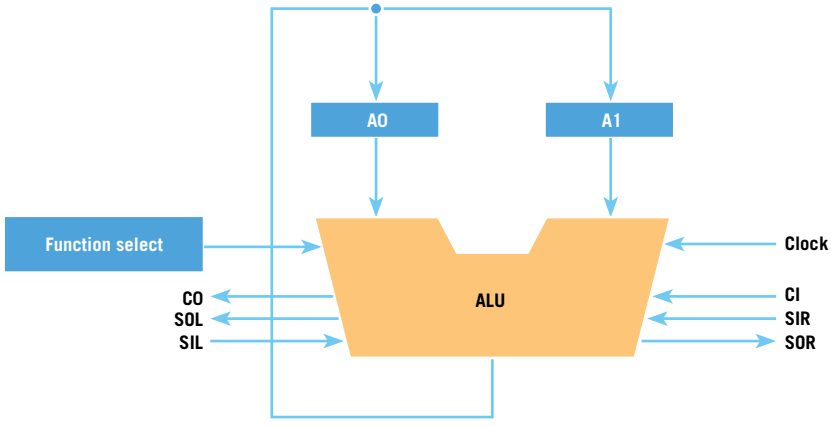


Figure 2

custom logic functions. The macrocells are typically clocked, and their outputs can be fed back into the product term array. This allows state machines to be created.

Large-scale PLDs can be used to form complex logic functions, even full CPUs, and thus can certainly be used to make smart digital peripherals. However, a lot of gates may be needed to implement even simple logic functions like counters or adders, and it can become expensive to scale up a PLD-based solution for more complex functions. At some point, it makes more sense to just use an actual CPU.

A very simple form of such a CPU is a datapath based on an arithmetic logic unit (ALU), also known as a *nano-processor*. A datapath implements just a few common functions but does so more efficiently than an implementation using PLDs. **Figure 2** shows a simple datapath with an ALU. A typical ALU can do a variety of operations, usually on 8-bit operands: count up (increment), count down (decrement), add, subtract, logical AND, logical OR, logical XOR, shift left, and shift right. There are two 8-bit accumulators that can act as either input data registers or storage for ALU output. A single operation takes place on the edge of an input clock signal. A function select register is used to control:

- What operation takes place.
- The source register(s) for that operation.
- The destination register for the output.

Which method is better for creating smart, low-cost digital peripherals—PLDs or datapaths? Separately neither, but together they work well.

Depending on the specific design of the datapath, it is possible to do a series of complex operations, as shown in **Table 1**.

The function select block can actually be a small SRAM, preloaded with the desired function select bits, and the SRAM's address lines can be used to select which operation is to be done. Finally, multiple datapaths can be chained together with carry and shift signals so that operations can be done on multibyte operands.

Since a datapath does only a few specific functions, it's possible to optimize its design so that it is inexpensive to build. However, a datapath is not nearly as flexible as a PLD for implementing complex logic. So which method is better for creating smart, flexible, low-cost digital peripherals—PLDs or datapaths? The answer is that separately neither one works well but together they can work very well. Let's take a look at a practical example of how this is done.

Examples of datapath functions implemented by function select bits.

Function select bits	Function
Operation: increment Source: A0 Destination: A1	A1 = A0 + 1
Operation: subtract A0 – A1 Sources: A0, A1 Destination: A0	A0 = A0 – A1
Operation: right shift Source: A0 Destination: A0	A0 = A0 >> 1

Table 1

UNIVERSAL DIGITAL BLOCKS

An example of a system utilizing both PLD and datapath components is Cypress Semiconductor's PSoC 3 and PSoC 5 ICs. Each system contains up to 24 general-purpose digital logic subsystems called Universal Digital Blocks (UDBs) constructed as shown in **Figure 3**. A UDB contains two PLDs of the type shown in Figure 1. It also con-

tains a datapath as well as status and control registers. There are two chaining paths, one for the PLDs and one for the datapath. Finally, a routing channel exists to connect signals between each of the UDB's sub-blocks as well as between UDBs. Configuration of the PLD, datapath, and routing is done by writing to UDB configuration registers (not shown).

The UDB's PLD design was described in Figure 1. As shown in **Figure 4**, the UDB datapath is similar to the basic datapath concept shown in Figure 2 but is more sophisticated—it has more registers and more features:

- The 8-bit ALU can do all seven basic functions—increment, decrement, add, subtract, AND, OR,

Block diagram of a Universal Digital Block (UDB).

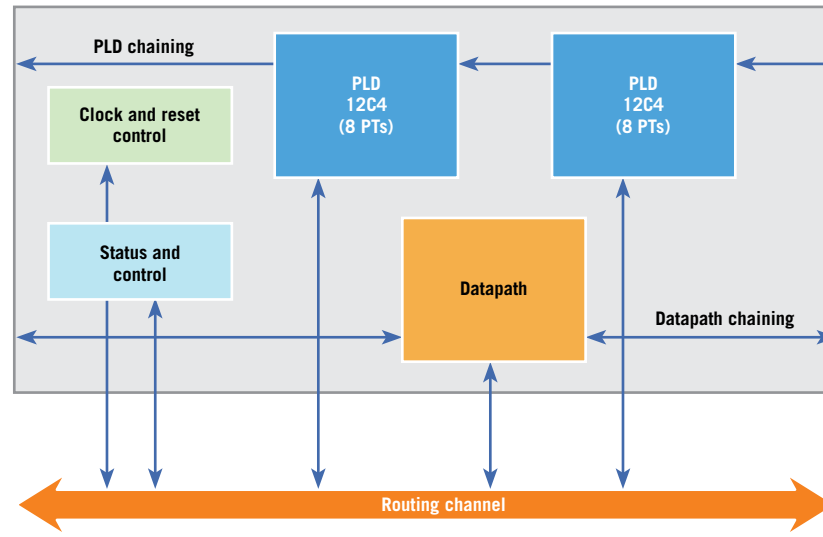


Figure 3

Block diagram of UDB datapath.

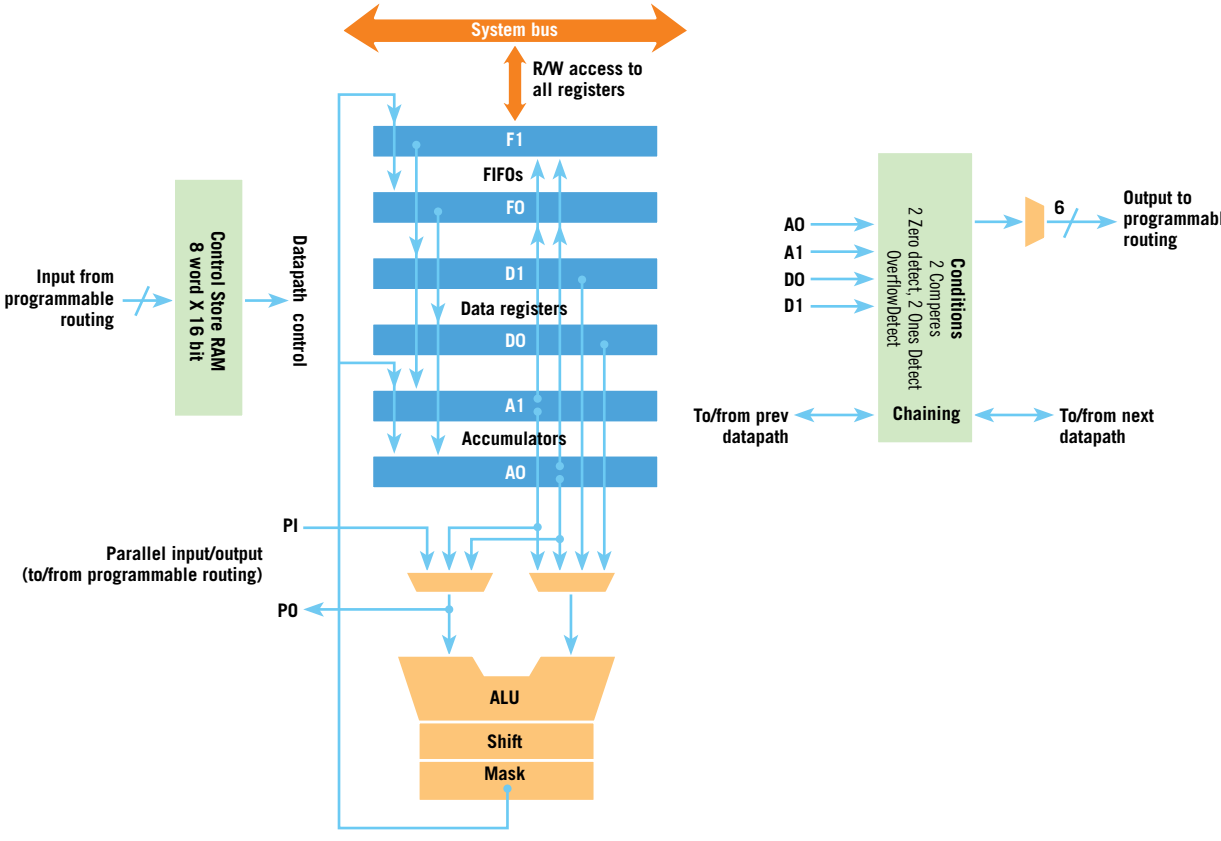


Figure 4

UDB datapath configured as a counter with reload.

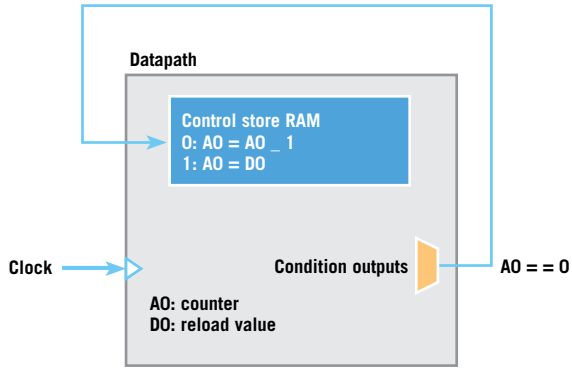


Figure 5

XOR—and has separate shift and bit-mask blocks for post-processing the ALU result. (An eighth ALU function, *pass*, just passes a value through the ALU to the shifter and bit-mask blocks.) The shift block can do shift left, shift right, nibble-swap, and pass. The mask block does a bitwise AND with the contents of a separate mask register (not shown).

- Operations can be done using two accumulators (A0, A1) and two data registers (D0, D1). Two FIFO registers (F0, F1) are available for passing data between the datapath and the CPU. The FIFOs are up to 4-bytes deep. This structure enables simple multitasking; at different times separate operations can be done on subsets of the registers.

This structure enables simple multitasking; at different times separate operations can be done on subsets of the registers.

Block diagram of a traffic-light controller using UDB PLDs and datapaths.

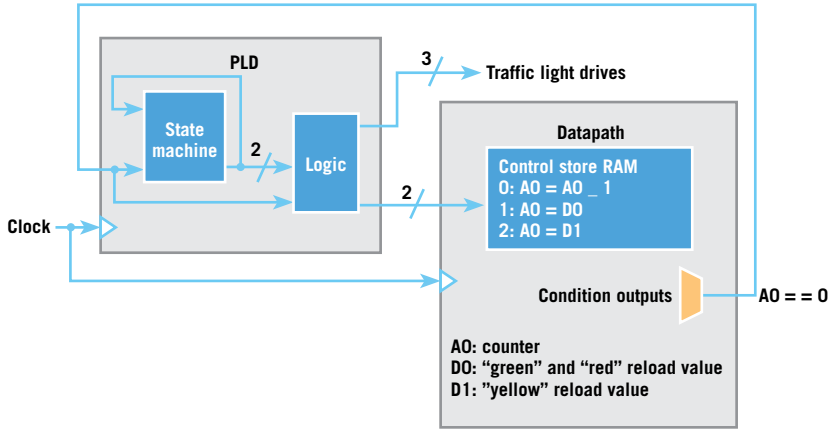


Figure 6

ables simple multitasking; at different times separate operations can be done on subsets of the registers. So for example A0, D0, and F0 could be used for one task and A1, D1, and F1 could be used for a different task.

- A broad set of status conditions—compare, zero detect, all ones detect, and overflow detect—can be applied to the accumulators and data registers and routed elsewhere in the device.

FLEXIBLE ROUTING

Although the UDBs have a lot of features in both the PLD and datapath subsystems, what makes them especially useful is the extensive digital routing that is also offered. Signals can be routed among the PLDs and datapaths throughout the entire set of UDBs, and elsewhere in the device, to form a complex fabric called the Digital System Interconnect (DSI).

EXAMPLES

In a basic example, we can use one UDB datapath to create an 8-bit counter with reload capability. To do this we connect one status condition back to a control store SRAM address line, as shown in **Figure 5**.

In this design, A0 is the counter register and D0 is the reload register. We need two functions, one to decrement the counter and one to reload the counter from the period register; these functions are preloaded in the Control Store RAM.

The logic is as follows. When A0 is not zero, the condition output will be low and the decrement operation at address 0 will be executed. When A0 is zero, the condition output will be high and the reload operation at address 1 will be executed.

All operations take place on the rising edge of the clock input, allowing the number of clock edges to be counted. The clock input can be routed from a variety of sources. The condition output can be routed through-

out the DSI, including to DMA and interrupt request inputs. Using datapath chaining and the mask block, the size of this counter can be any number of bits, and is not limited to a multiple of eight bits.

The counter shown in Figure 5 is a down counter. It can easily be converted to an up counter by using a different condition output (A0 == D0) and different functions in the control store SRAM: A0 = A0 + 1, and A0 = A0 XOR A0. Exclusive-or'ing any value with itself yields a zero result.

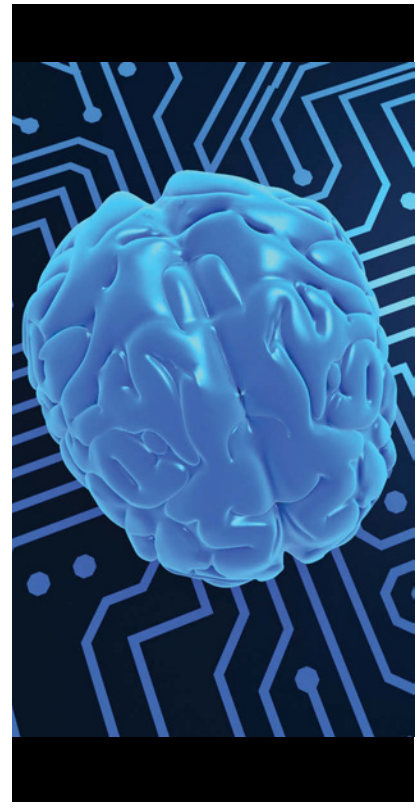
This simple design can be expanded, with the use of PLDs, to create a more complex application. To illustrate this, consider a traffic-light controller. A traffic-light controller cycles through three states—green, yellow, and red—so a state machine is required. Each state lasts for a certain amount of time before changing to the next state, so a counter is also required. For simplicity, assume that the "green" time is the same as the "red" time but that the "yellow" time is different.

Only one datapath is needed (assuming an 8-bit count value) to implement this timing structure, and three of the datapath registers are used. A0 is the count register, D0 contains the counter reload value for the "green" and "red" states, and D1 contains the counter reload value for the "yellow" state. The block diagram is shown in **Figure 6**.

The operations to be saved in the Control Store RAM are:

```
A0 = A0 – 1  // count
A0 = D0      // reload "green" or "red"
             // count value
A0 = D1      // reload "yellow" value
```

The state machine is implemented in the PLD. The datapath condition output is fed back to the PLD to indicate that it's time to change state. The PLD also has logic that, based on the current state and the signal fed back from the datapath, controls which datapath operation to perform and which traffic light to activate.



By offloading tasks to smart support peripherals, the CPU is freed up to do other, maybe more lucrative tasks.

BEYOND THE BASICS

A traffic-light controller is a simple application of a type that is commonly programmed using a CPU. However, we have seen that, except for initialization code, this function can be completely dissociated from the CPU and in fact has been encapsulated as a smart configurable peripheral. The functionality can be easily expanded to support additional requirements such as turn signals, pedestrian WALK signals, vehicle detect sensors, and transit/emergency transponders.

WHAT'S A CPU TO DO?

By using an efficient combination of PLDs and datapaths, you can create smart, flexible, low-cost peripherals that take the load off the CPU. However, if so much functionality can be offloaded to peripherals, what's left for the CPU to do? In many cases, not much—in some cases after system initialization, the CPU can be turned off! However, a more realistic solution is to use the CPU to do what CPUs do best, such as:

- Complex calculations.
- String and text processing.
- Database management.
- Communications management.
- System management.

For example, in our traffic-light application, the CPU could be used to:

- Detect when a vehicle goes through a red light,
- Use the camera to photograph the license plate,
- Extract the license plate's text from the photo,
- Look up the owner in the state database, and
- Send a ticket to the owner.

By offloading tasks to smart support peripherals, the CPU is freed up to do other, maybe more lucrative tasks. ■

Mark Ainsworth is a applications engineer principal at Cypress Semiconductor. He has a BS in computer engineering from Syracuse University and an MSEE from the University of Washington.

Learn today. Design tomorrow.

ESC

If you like going it alone or bucking convention, programming embedded systems may not be for you. Rarely is it a solo job anymore. Brilliant as we are, we embedded systems developers need to code using production-oriented mindset, which means following the same house style guide and rules to get the job done right.

Adopting C programming conventions

BY JEAN J. LABROSSE

This paper (from the Embedded Systems Conference class of the same name) discusses some of the problems found in a lot of code today and suggests how they can be avoided. There are many ways you and your embedded development team can improve code quality and in the process, become significantly more productive. Techniques are presented to organize project directories, naming files, laying out code, naming variables, functions, and more. Examples are presented for C but most of the concepts apply to other languages.

Today's competitive world forces us to introduce products at an increasingly faster rate due to one simple fact of business life: having a product out first may mean acquiring a major share of the market. One way to help make this possible is to assure that the mechanics of writing code become second nature. All project members should clearly understand where each file resides on the company's file server, what each file should be named, what style to use, and how to name variables and functions.

The topic of coding conventions is controversial because we all have our own ways of doing things. One way is not necessarily better than the other. However, it's important that all team members adopt a single set of rules and that these rules are followed religiously and consistently by all participants. The worse thing that you can do is to leave

each programmer to do his or her own thing. Such an undisciplined activity will certainly lead to chaos. When you consider that close to half of the development effort of a software-based system comes after its release, why not make the sometimes unpleasant task of supporting code less painful?

In this paper, I'll share some of the conventions I've been using for years and I hope that you'll find some of them useful for your own organization. I urge you to document your own conventions because it makes life easier for everyone, especially when it comes to supporting someone else's code.

! It's important that all team members adopt a single set of rules and follow them religiously and consistently. The worse thing you can do is to leave each programmer do his or her own thing.

DIRECTORY STRUCTURES

One of the first rules to establish is how files are organized. Do you place all the source files in a single directory or do you create different directories for different pieces? I like to use a structure similar to that shown in **Table 1**.

Each product (such as `ProdName`) has its own directory under `PRODUCTS\`. If a product requires more than one microprocessors then each has its own directory under `ProdName\`. All products that contain a microprocessor has a `SOFTWARE\` directory. The `SOURCE\` directory contains all the source files that are specific to the product. If you have highly modular code and strive to reuse as much code as possible from product to product, the `SOURCE\` directory should generally only contain about 10% to 20% of the software which makes the product unique. The remaining 80% to 90% of the code should be located in the `\SOFTWARE` directory (discussed later). The `DOC\` directory contains documentation files specific to the software aspects of the product (such as specifications, state diagrams, flow diagrams, and software description). The `TEST\` directory contains product build files (such as batch files, make files, and IDE project) for creating a test version of the product. A test version will build the product using the source files located in the `SOURCE\` directory of the product, any reusable code (building blocks), and any test-specific source code you may want to include to verify the proper operation of the application. The latter files generally reside in the `TEST\` directory because they don't belong in the final product. The `PROD\` directory contains build files for retrieving any released versions of your product. The other directories under `ProdName\` are provided to show you where other disciplines within your organization can store their product-related files. In fact, this scheme makes it easy to backup or archive all the files related to a given product whether they're related to software or not.

The `\SOFTWARE\` directory is where you could store all reusable, non-product specific files. I call these *building blocks,* and each contains its own documentation.

SOURCE FILES

It's well known that coding is the only aspect of programming that must be done to have a product. All the documentation in the world is useless if it doesn't reflect what the code does. C has been called a write-only language because once you have written the code, it's difficult to read

An example directory structure for source files.

Directory contents			Directory name
PRODUCTS			(PRODUCTS)
ProductName			
	Manual		(MANUALS)
	Software		
		Documentation	(DOC)
		Source	(SOURCE)
		Object Code	(OBJ)
		Listings	(LST)
		Test	(TEST)
		Product Build	(PROD)
	Hardware		
		Design and Analysis	(DESIGN)
		Schematics	(SCH)
		PCB layouts	(PCB)
		BOM (Parts List)	(BOM)
	Mechanical		
SOFTWARE (reusable components)			(SOFTWARE)
ModuleName			
	Source		(SOURCE)
	Documentation		(DOC)
	Releases		(RELEASES)

Table 1

2011 EE Times ACE AWARDS

UBM Electronics Celebrates the Industry and its Innovation through the EE Times ACE Awards and EDN Innovators Awards

2011 EE TIMES ACE AWARDS

The Gallery of Innovation

Lifetime Achievement Award Honoree

Irwin Mark Jacobs

Co-Founder, Former Chairman, Qualcomm

The Annual Creativity in Electronics (ACE) Awards celebrates the creators of technology who demonstrate leadership and innovation in the global electronics industry and shape the world we live in.

Finalists and winners will be recognized by EE Times editors, a distinguished judging panel and the global electronics industry.

Tuesday, May 3rd 2011
5:30pm-7:00pm
Fairmont San Jose
Club Regent Room

Tickets available:
www.eetimes-ace.com

Good Luck Finalists!

NOMINATION BASED:

DESIGN TEAM OF THE YEAR:
Advanced Micro Devices
IBM
Intel Corporation
Nordic Semiconductor
PrimeSense

INNOVATOR OF THE YEAR:
Adapteva Inc.–Andreas Olofsson
Cirrus Logic–John Melanson
Intel–Mario Paniccia
NuPGA Corporation–Zvi Or-Bach
Wave Systems Corp.
–Robert Thibadeau

EXECUTIVE OF THE YEAR:
ARM–Warren East
Cirrus Logic–Jason Rhode
GLOBALFOUNDRIES
–Douglas Grose
Maxim Integrated Products
–Tunc Doluca
Spansion–John Kispert

STARTUP OF THE YEAR:
Dynamics Inc.
GLOBALFOUNDRIES
Lunera Lighting Inc.
Pixtronix
Semitech Semiconductor Pte Ltd

COMPANY OF THE YEAR:
Altera Corporation
ARM
Maxim Integrated Products
NetLogic Microsystems, Inc.
TriQuint Semiconductor, Inc.

MOST PROMISING NEW TECHNOLOGY:
Hillcrest Labs
InvenSense, Inc.
InVisage Technologies Inc.
Lyric Semiconductor
Tilera

ENERGY TECHNOLOGY AWARD:
Efficient Power Conversion Corporation
Freescale Semiconductor
SolarBridge Technologies
SPD Control Systems Corporation
Freescale Semiconductor

IEEE AWARD CATEGORIES:

IEEE SPECTRUM TECHNOLOGY IN THE SERVICE OF SOCIETY & THE IEEE SPECTRUM EMERGING TECHNOLOGY AWARD
Laster Technologies (Smart Spectacles)
Seabed Rig (Robotic Oil Driller)
IBM (Watson)
ClariPhy Communications (Digital Processor)
Willow Garage (Personal Robotics)

EE TIMES SELECTED CATEGORIES:

MOST POPULAR PRODUCT OF THE YEAR (7 CATEGORIES)
-Digital logic
-Analog-mixed signal
-Memories
-Interconnects
-Electro-mechanical
-Software/IP
-Test & Measurement

LIFETIME ACHIEVEMENT AWARD
EDITOR'S CHOICE AWARD
CONTRIBUTOR OF THE YEAR
MOST ENGAGED MEMER OF THE EE TIMES COMMUNITY OF THE YEAR
BEST STUDENT DESIGN/ DESIGN CHALLENGE WITH PUBLIC SCHOOLS
EE LIFE CONTRIBUTOR OF THE YEAR
STUDENT OF THE YEAR

UBM Electronics

GOLD SPONSORS

AVNET Embedded

MOUSER ELECTRONICS a tti company

ASSOCIATION MEDIA SPONSOR

ieee spectrum INSIDE TECHNOLOGY

and understand what it does. I believe that any language can be made write-only because this depends more on your attitude than the language you use. I always say that, if you hate writing (or typing) then you're in the wrong business. Many programmers write for themselves and are not concerned with the life of a product. Another sign that you're in the wrong business is if you believe that your job is done once the code works. A major portion of a product's life consist of maintenance. In fact, in most cases, maintenance accounts for the majority of a product's development cost and thus, you should write source code to facilitate maintenance.

One of the easiest ways I found to accomplish this goal is to adopt a clean and consistent coding style. Which style you decide to use doesn't matter, as long as you and others in your organization follow a common guide. To that end, somebody should document the style used in your organization and have everyone follow it religiously. A few years ago, I read an article in the *Hewlett-Packard Journal* that stated that the product development manager decided to have the team adopt a common coding style (a potentially dangerous management decision!).[1] The team members were initially reluctant about having to conform to the style guide, but at the completion of the project everybody was impressed by the productivity gains. Team members were able to help each other because they didn't have to adjust themselves to other members' coding styles to find bugs.

Source files should be grouped by modules with each module having its own directory. Taking C as an example, a directory may contain as few as two files (a `.C` and a `.H` file) or as many files as needed to perform the functions of the module. In some cases, lookup tables are placed in the `.C` file along with the module's code while in other cases, large tables are located in their own files. All files within a module share a common file name prefix. For example, a display module may consist of three files: `DISP.C` contains code and variables, `DISP.H` contains function prototypes and `DISP_TC.C` contains tables (the `_TC` means **T**ables written in **C**). If you have multiple types of display modules (such as LCD and LED), each can be located in its own directory. For example, the LCD module can be placed in `\SOFTWARE\LCD` while the LED module in `\SOFTWARE\LED`. Also, the name of the files can be the same in both cases. In fact, you should try to provide the same functionality (such as function names and variable names) whether you have an LCD-based display or an LED-based display. Where the application code is concerned, it only knows that it has a display, and the product requirements dictate which type is used.

Team members were able to help each other because they didn't have to adjust themselves to other members' coding styles to find bugs.

I don't like to limit the width of source code to 80 characters. In fact, I still don't know why some programmers limit themselves to 80 columns. Deciding on the width of a source file can become controversial. I believe that you should *not* limit your source code to 80 columns even if you never print your source code. The reason is simple. I like to put executable statements on the left and comments on the right. This simple technique makes it easier to follow your code—your mind doesn't continuously have to "filter" the code from the comments if the comments are interleaved with your code (more on this later). The 80-column rule comes from old text-based monitors that allowed you to display only 80-characters wide and was useful for code reviews when code was printed. Today's large monitors can easily display 200-columns wide and quite frankly, I don't recall the last time I actually printed code.

A source file should contain the following sections and these sections should always be in the same order. Header files should never contain code except for macros.

1. File heading.
2. Revision history.
3. `#defines` and macros.
4. `#includes`.
5. Variables.
6. Function prototypes.
7. Functions.

The **file heading** section is a comment block that contains your company name, address, copyright notice, file name, author's name, and a description of the module.

The **revision history** section is also a comment block that describes what changes were made to the module over time. This section can automatically be filled in by your version control software when the module is released (assuming you're using version control software—hopefully you do). I have seen cases where the revision history section is actually at the end of the file so that the line numbers are not affected as much each time revision comments are inserted. Either way is acceptable.

`#defines` and macros are located in three different places. First, if `#defines` and macros are only applicable to the module, they're placed in the module's `.C` file (for example, equating the different states of a state machine). There is no point of extending the scope of something that is used locally. If used by multiple `.C` files in a given module, `#defines` and macros can be placed in the module's `.H` file to make them globally visible to the module. Finally, if the `#defines` are meant to be product specific, they're

placed in a file called APP.H (you can certainly use a different name) in the product's directory. For example, the size of a module's buffer can be specified in APP_CFG.H (application configuration) instead of the module itself. Also, you can #define a constant to enable/disable compilation of a feature in a module. This allows your code/data size to be reduced in case you don't need all the functionality of a module.

In other cases, conditional compilation is used to enable a version of an algorithm. For example, a CRC (cyclic redundancy check) module can contain two versions. The first version can be slow but require very little ROM while the other can be very fast but require the use of a 512-bytes table and thus consumes more ROM. In this case, setting the #define CRC_CFG_FAST_EN to 1 selects the faster version. Where the application is concerned, it doesn't know the difference. #defines and macros are always written using uppercase characters with an underscore character separating words. This agrees with the conventions established by Kernighan and Ritchie.[2]

I don't think that globals are bad. Having variables globally accessible may be beneficial when debugging and at run time to visualize what your product is doing.

Every .C file whether product specific or part of a reusable module contains a #include section that always consists of a single statement as follows:

```
#include "INCLUDES.H"
```

I like to use a single "master" include file called INCLUDES.H whose contents is defined in the product's directory. I use a single master header file because it prevents you from having to remember which header file goes with which source file, especially when new modules are added. When you design reusable modules, you will never have to remember which header files are needed with which module—they are always *all* included. If you add a new module, you simply include its header file in INCLUDES.H. The only inconvenience that I found is that it takes a little bit longer to compile each file but, this is barely perceptible with today's fast computers. Some people hate the idea of exposing all the header files to all the .C files because they believe programmers will then start accessing everything that the headers are exposing. I believe that can easily be controlled. Using a single include file is a matter of preference.

VARIABLES

Most of today's C compilers conform to the ANSI X3J11 standard, which allows up to 32 characters for identifiers. Descriptive variables can be formulated using this 32-character feature and the use of acronyms, abbreviations, and mnemonics (see section on acronyms, abbreviations, and mnemonics). Variable names should reflect what the variable is used for. I like to use a hierarchical method when creating a variable. For instance, the array KeyBufIn[] indicates that it is part of the keyboard module (Key), it is a buffer (Buf)—and specifically the input buffer (In). Uppercase characters are used to separate words in a variable, but this rule only applies to global variables. This is called *Camel Back*.

In C, you can have two types of global variables: *global to a file* (also called *local globals*) and *global to the rest of the world* (such as the product). I personally don't think that globals are bad and thus should be avoided. Having globals doesn't mean that you should not encapsulate their access through interface functions. Having variables globally accessible may be beneficial when debugging and at run time to visualize what your product is doing.

All global variables (variables seen by other modules) are placed in the .H file of the module and not in the .C file. In C, however, the .H file generally contains an extern statement so, the question is: "How do you extern a variable and allocate storage for it at the same time?" The answer is that you use conditional compilation through the C preprocessor as shown in the following statements that are placed at the beginning of the .H file:

```
#ifdef  xxx_GLOBALS
#define xxx_EXT
#else
#define xxx_EXT extern
#endif
```

Where, xxx is the name of the module. Each variable that needs to be declared global will be prefixed with xxx_EXT in the .H file. The module's .C file will contain the following declarations:

```
#define   xxx_GLOBALS
#includes "INCLUDES.H"
```

When the compiler processes the .C file, it forces xxx_EXT (found in the corresponding .H file) to "nothing" (because xxx_GLOBALS is defined) and thus each global variable will be allocated storage space. When the compiler processes the other .C files, xxx_GLOBALS will not be defined for that module and thus xxx_EXT will be set to extern allowing you to reference the global variables. To illustrate this concept, let's suppose you need to create the following global variables:

```
CPU_INT08U CtrlState;
CPU_FP32   CtrlLevel;
CPU_INT16U CtrlCtr;
```

CTRL.H would look like this:

```
#ifdef  CTRL_GLOBALS
#define CTRL_EXT
#else
#define CTRL_EXT extern
#endif

CTRL_EXT  CPU_INT08U CtrlState;
CTRL_EXT  CPU_FP32   CtrlLevel;
CTRL_EXT  CPU_INT16U CtrlCtr;
```

CTRL.C would look like this:

```
#define   CTRL_GLOBALS
#includes "INCLUDES.H"
```

The nice thing about this technique is that you don't have to declare global variables in the .C file and duplicate the statements with the addition of the extern attribute in the .H file. You not only save a lot of time but you also reduce the chances of introducing an error in the process. Once you use this technique, you'll never want to declare globals any other way.

Variable names should be declared on separate lines rather than combining them on a single line. Separate lines make it easy to provide a descriptive comment for each variable. You should also explicitly declare the data type of every variable instead of relying on the default, int.

By convention, all variables are prefixed with the module's name. This convention makes it quite easy to know where variables are declared when you're dealing with large applications. Furthermore, a file scope global should have the underscore character (in other words, '_') after the module name. For example, a local global variable in the file CTRL.C would be prefixed by Ctrl_. Because CTRL.C will most likely manipulate Ctrl variables you will be able to know whether the variables are declared at the top of CTRL.C (Ctrl_ prefix) or in CTRL.H (Ctrl prefix).

You not only save a lot of time but you also reduce the chances of introducing an error. Once you use this technique, you'll never want to declare globals any other way.

Formal arguments to a function and local variables within a function are declared in lowercase. The lowercase convention makes it obvious that such variables are local to a function because, also by convention, global variables will contain a mixture of upper- and lowercase characters. To make local variables or function arguments readable, you can use the underscore character (in other words, _). Within functions, certain variable names can be reserved to always have the same meaning. Some examples are given below but others can be used as long as consistency is maintained.

i, j, and k	for loop counters.
p1, p2 ... pn	for pointers.
c, c1 ... cn	for characters.
s, s1 ... sn	for strings.
ix, iy, and iz	for intermediate integer variables
fx, fy, and fz	for intermediate floating-point variables

Structures are typedef since this allows a single name to represent the structure. The structure type is declared using all uppercase characters with underscore characters used to separate words, shown in **Listing 1**.

I find it very useful to include the name of the structure in the suffix of a pointer as shown below. This allows the reader to know what structure the element being referenced belongs to.

```
p_line->>Color;
```

Listing 1

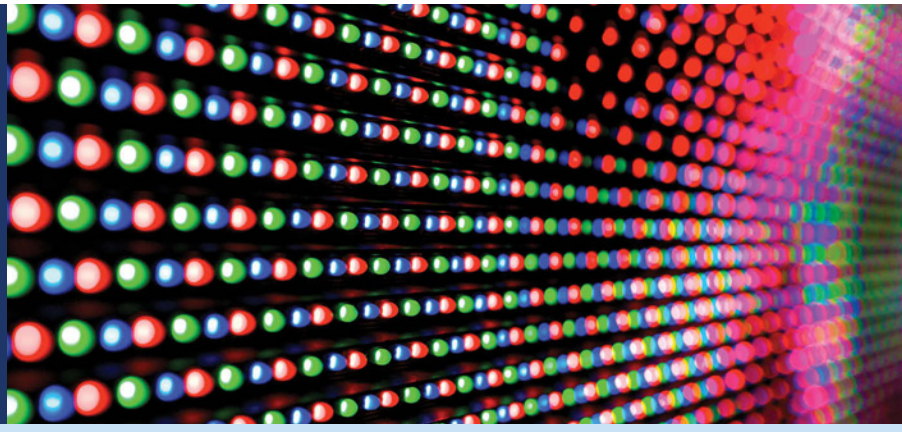
```
typedef struct {            /* Structure that defines a LINE */
    CPU_INT16U  StartX;     /* 'X' & 'Y' starting coordinate */
    CPU_INT16U  StartY;
    CPU_INT16U  EndX;       /* 'X' & 'Y' ending   coordinate */
    CPU_INT16U  EndY;
    CPU_INT16U  Color;      /* Line color                    */
} LINE;
```

designing with LEDs

Produced By: EDN

MAY 4, 2011
San Jose, CA

SEPTEMBER 27, 2011
Boston, MA



Unlock the new world of energy-efficient solid–state lighting at EDN's Designing with LEDs Workshop

Full day technical seminar and workshop

This hands-on event is created for design engineers and systems architects who need to learn and explore drive electronics and thermal management options for the newest generation of energy-efficient, high-brightness LEDs. Get practical training on how new HB LED devices, packages, control electronics and thermal devices combine to revolutionize lighting for consumer and medical devices, automotive, architectural, and signage applications. Spend the day with us and learn!

Keynote Address: Cary Eskow, Director, Lightspeed

Solid-state lighting enabled by high-brightness LEDs will creep into every facet of daily life within a few short years. Cary Eskow, renowned LED industry expert will identify how the latest technology advances will usher in a new light-enabled view of tomorrow.

TECHNICAL TRACKS:

- Power management
- Thermal management
- Optics and light measurement
- Beyond the light: Networking and Solar Power

HIGHLIGHTS:

- Plenary LED manufacturers' panel, featuring representatives from Cree, Lumileds, and Osram, will focus on the relationship between LED drive current and lifetime.
- Lightening talks, and ample networking opportunites.

Check our web site for the latest agenda and updates.
e.ubmelectronics.com/LED

Register Today and Save 10% with Promo Code EDNAD
e.ubmelectronics.com/LED

CORNERSTONE SPONSORS

GOLD SPONSORS

SILVER SPONSORS

Unless you have to use the C standard library, you should avoid using C's data types because they're inherently not portable.

To summarize, global variables should use the file/module name (or a portion of it) as a prefix and should make use of upper-/lowercase characters. File scope globals should have an underscore character following the module's prefix. Function arguments and local variables should use only lowercase characters. `#define` constants and macros are always written in uppercase with underscore characters separating words for sake of legibility.

ACRONYMS, ABBREVIATIONS, AND MNEMONICS

When creating names for variables and functions, it's often useful to use acronyms (such as OS, ISR, TCB), abbreviations (such as buf and doc), and mnemonics (such as clr and cmp). Their use allows an identifier to be descriptive while requiring fewer characters. Unfortunately, if the terms are not used consistently, they may add confusion. To ensure consistency, you should create a list of acronyms, abbreviations, and mnemonics that you will use in all your projects. I call this list the *Acronyms, Abbreviations, and Mnemonic Dictionary*. Once it is assigned, an acronym, abbreviation, or mnemonic is used throughout. As we need more terms, we simply add them to the list. Once everyone has agreed that `Buf` means buffer, all project members should use that instead of having some individuals use `Buffer` and others use `Bfr`. To further this concept, you should always use `Buf` even if your identifier can accommodate the full name. In other words, stick to `Buf` even if you can fully write the word `Buffer`.

There might be instances where one list for all products doesn't make sense. For instance, if you are an engineering firm working on a project for different clients and the products that you develop are totally unrelated, a different list for each project would be more appropriate; the vocabulary for the farming industry is not the same as the vocabulary for the defense industry. My rule is that if all products are similar, they use the same dictionary.

DATA TYPES

While we're on the subject of variables, you may have noticed that I don't use the standard C types in variable declarations. In fact, unless you have to use the C standard library, you should avoid using C's data types because they're inherently not portable. An `int` can either be 8, 16, 32, or even 64 bits. Similarly, a `float` is either a 32-bit, a 64-bit or 80-bit value depending on the target processor and compiler. To resolve the portability issue, we create a header file (I call it `CPU.H`) that defines the following data types:

```
typedef unsigned char  CPU_INT08U;
typedef signed   char  CPU_INT08S;
typedef unsigned int   CPU_INT16U;
typedef signed   int   CPU_INT16S;
typedef unsigned long  CPU_INT32U;
typedef signed   long  CPU_INT32S;
typedef float          CPU_FP32;
typedef double         CPU_FP64;
```

The current version of the C standard "resolved" the issue of data type sizes by introducing data types that specifies the resolution of each type. However, I still contend that an all UPPERCASE data type makes code much more readable. That being said, you could declare the new data types while still using the C99 types as the base types as shown below:

```
typedef uint8_t        CPU_INT08U;
typedef int8_t         CPU_INT08S;
typedef uint16_t       CPU_INT16U;
typedef int16_t        CPU_INT16S;
typedef uint32_t       CPU_INT32U;
typedef int32_t        CPU_INT32S;
typedef float          CPU_FP32;
typedef double         CPU_FP64;
```

By convention, `CPU_INT08S` is always used to declare 8-bit signed variables, similarly, `CPU_INT16U` declares 16-bit unsigned variables, and so forth. Your application code as well as the reusable modules can now assume the appropriate range for each variable in a portable fashion. If you then decide to port your code to a different target, you'll only need to look up the definition of various data-type sizes in your compiler literature and change the above definitions.

FUNCTIONS

Function naming follow the same convention as with global variables. Every function is prefixed with the module name; again, I use acronyms, abbreviations, and mnemonics. The first letter of each word is capitalized and local functions (file scope) have an underscore after the module name. I found that indenting four spaces works out well, but you should use whatever you are comfortable with.

Whatever you do, use spaces instead of tabs to indent your code. Tabs are interpreted differently on different editors and printers. Avoiding tab characters doesn't mean that you can't use the tab key on your keyboard. A good editor will give you the option to replace tabs with spaces (in this case, four spaces). If you have a lot of legacy code that contain tab characters, you can write a simple utility that scans

your source code for tab characters and replaces each one with the number of spaces you decided to adopt.

Functions that are only used within the file should be declared `static` to hide them from other functions in different files. Each local variable name should be declared on its own line, an action that allows the programmer to comment each one as needed. Actual code statements should start after adding two blank lines after local variable declarations. This makes the delineation between variables and executable statements clear. A function should be declared as follows:

```
void  CommRx (CPU_INT08U ch, CPU_INT08U c)
{
}
```

I even like the following style, which allows you to isolate the arguments onto its own line. When you have many arguments, it's a lot easier to see how many arguments there are, what their type is, and so on:

```
void
CommRx (CPU_INT08U  ch,
        CPU_INT08U  c)
{
}
```

You should note that I included a single space between the function name and the open parenthesis. This convention allows you to quickly locate where a function is actually declared when using your editor's search capability or even a `grep` utility. When you actually invoke the function, you should not include a space between the function name and the open parenthesis. This "feature" may not be necessary anymore if you use advanced editors that can jump to the function declaration when you "hover" over the name of the function or variable.

Your style guide should also specify how every C construct should be written. A space follows the keywords `if`, `for`, `while`, and `do`. The keyword `else` has the privilege of having one before and one after it if curly braces are used. We write `if (condition)` on its own line and the statement(s) to execute on the next following line(s):

```
if (y > 2) {
    z =  10;
    x = 100;
    p++;
} else {
    z =  5;
}
```

I always fully enclose statements within the `if (condition)` with curly braces even though the condition executes a single statement. This makes it convenient to add additional statements and prevents you from forgetting to add the curly braces when you add these extra statements. Also, the placement of curly braces follows the K&R style, but obviously you should adopt the style your organization is comfortable with.

Treat `switch` statements as you would any other conditional statement. Note that the `case` statements are lined up with the case label. The important point here is that `switch` statements must be easy to follow. Cases should also be separated from one another by a blank line.

```
switch (key) {
    case KEY_BS:
         if (cnt > 0) {
             p--;
             cnt--;
         }
         break;

    case KEY_CR:
         *p = NUL;
         break;

    case KEY_LINE_FEED:
         p++;
         break;

    default:
         *p++ = key;
         cnt++;
         break;
}
```

By convention, I use `for` loops when I know ahead of time the number of iterations the loop will perform. On the other hand, I use `while` and `do-while` loops when the number of iterations is only known at run time as shown below.

```
for (i = 0; i < MAX_ITER; i++) {
    *p2++ = *p1++;
    xx[i] = 0;
}


while (*p1 != 0) {
    *p2++ = *p1++;
    cnt++;
}


do {
    cnt--;
    *p2++ = *p1++;
} while (cnt > 0);
```

You should avoid multiple assignments on the same line.

```
x = y = z = 1;
```

I also like to break up a long statement into multiple lines (as long as it's one statement). Note how the '+' sign lines up with the equal signs. This makes the code much more readable.

```
x2   = x  * x;
x3   = x2 * x;
x4   = x3 * x;
x5   = x4 * x;
x6   = x5 * x;
temp = b0 * x
     + b1 * x2;
     + b2 * x3;
     + b3 * x4;
     + b4 * x5;
     + b5 * x6;
```

The following operators are written with no space around them:

`->>`	Structure pointer operator	`p->>m`
`.`	Structure member operator	`s.m`
`[]`	Array subscripting	`a[i]`

As previously mentioned, you declare a function with one space following its name and the open parenthesis while you invoke the function with no space after the function name. A space should be introduced after each comma to separate each actual argument in a function. Expressions within parentheses are written with no space after the opening parenthesis and no space before the closing parenthesis. Commas and semicolons should have one space after them.

```
strncat(t, s, n);
for (i = 0; i < n; i++)
```

The unary operators are written with no space between them and their operands as shown in **Listing 2**.

The binary operators are preceded and followed by one or more spaces, as is the ternary operator, in **Listing 3**.

The keywords `if`, `while`, `for`, `switch` and `return` are followed by one space.

For assignments, numbers are lined up in columns as if you were to add them. This allows you to quickly spot errors. The equal signs are also lined up. "Magic numbers" are shown here only for sake of illustration to show how the "weight" of the numbers should line up. Magic numbers must be avoided and, in fact, replaced with `#define` constants so they are more legible.

```
x        = 100.567;
temp     =  12.700;
var5     =   0.768;
variable =  12;
storage  = &array[0];
```

COMMENTS

Comments should be meaningful and help you and others understand how the code works. Don't just state the obvious or what a reasonable programmer would conclude by simply looking at the code.

Each function should be preceded with a comment block to describe what the function does, what arguments are passed, what the function returns, and any other notes about the function.

I find it very difficult to mentally separate code from comments when code and comments are interleaved. Because of this, I avoid using this practice. Comments should go to the right of the actual C code. When large comments are necessary, they're written in the function description header or in a comment block before the actual code. Comments are lined up as shown in **Listing 4**. The comment terminators (`*/`) does not need to be lined up, but for neatness I prefer to do so. It is not necessary to have one comment per line since a comment could apply to a few lines.

Listing 2

```
!p   ~b   ++i   --j   (long)m   *p   &x   sizeof(k)
```

Listing 3

```
c1 = c2      x + y      i += 2      n > 0 ? n : -n;
```

Listing 4

```
/*
*********************************************************************************************************
*                                          UPDATE THE TIME-OF-DAY
*
* Description : This function is called to update the time (i.e. hours, minutes and seconds) or a
*               software managed clock.
* Arguments   : None.
* Returns     : TRUE      if we have completed one day.
*               FALSE     otherwise
* Notes       : This function updates the global variables: ClkSec, ClkMin and ClkHr.
*********************************************************************************************************
*/

static CPU_BOOLEAN ClkUpdateTime (void)
{
    CPU_BOOLEAN newday;


    newday = FALSE;                 /* Assume that we haven't completed one whole day yet    */
    if (ClkSec >= 59) {             /* See if we have completed one minute yet               */
        ClkSec = 0;                 /* Yes, clear seconds                                    */
        if (ClkMin >= 59) {         /*    See if we have completed one hour yet              */
            ClkMin = 0;             /*    Yes, clear minutes                                 */
            if (ClkHr >= 23) {      /*         See if we have completed one day yet          */
                ClkHr  = 0;         /*         Yes, clear hours ...                          */
                newday = TRUE;      /*         ... change flag to indicate we have a new day */
            } else {
                ClkHr++;            /*         No,  increment hours                          */
            }
        } else {
            ClkMin++;               /*    No,  increment minutes                             */
        }
    } else {
        ClkSec++;                   /* No,  increment seconds                                */
    }
    return (newday);
}
```

FINAL WORDS

You may not agree with some of the conventions that I adopt but what's important is that you recognize that you'll increase productivity and increase the quality of your code by having your organization work from a common set of rules. Programmers will certainly resist this kind of change but, the long-term benefits will be worth the struggle. I have concluded over the years that I much rather fight to have things done correctly the first time than spend double the effort when a programmer moves on to greener pastures and leaves the rest of us holding the proverbial bag! ■

Jean Labrosse is founder, CEO, and president of Micrium. He is a regular speaker at the Embedded Systems Conferences and is the author of three books: *MicroC/OS-II, The Real-Time Kernel, Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C and MicroC/OS-III, The Real-Time Kernel.* Jean has also written numerous articles for magazines. He has an MSEE and has been designing embedded systems for many years.

ENDNOTES:

1. Long, David W. and Christopher P. Duff. "A Survey of Processes Used in the Development of Firmware for a Multiprocessor Embedded System." *Hewlett-Packard Journal*, October 1993, p.59-65.
2. Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 1988 ISBN 0-13-110362-8.

FURTHER READING:

3. Labrosse, Jean J. *µC/OS-III, The Real-Time Kernel.* Weston, FL Micrium Press, 2009, ISBN 978-0-9823375-3-0.
4. Maguire, Steve. *Writing solid code.* Microsoft Press, Redmond, WA 1993.
5. McConnell, Steve. *Code Complete.* Microsoft Press, Redmond, WA, 1993, ISBN 1-55615-484-4.
6. Straker, David. *C-Style, Standards and Guidelines.* Prentice Hall, 1992, ISBN 0-13-116898-3.
7. Barr, Michael. *Embedded C Coding Standard.* Netrino Institute, 2008, ISBN 978-1442164826.

2011 EE Times ACE AWARDS

UBM Electronics Celebrates the Industry and its Innovation through the EE Times ACE Awards and EDN Innovators Awards

2011 EE TIMES ACE AWARDS

The Gallery of Innovation

Lifetime Achievement Award Honoree

Irwin Mark Jacobs

Co-Founder, Former Chairman, Qualcomm

The Annual Creativity in Electronics (ACE) Awards celebrates the creators of technology who demonstrate leadership and innovation in the global electronics industry and shape the world we live in.

Finalists and winners will be recognized by EE Times editors, a distinguished judging panel and the global electronics industry.

Tuesday, May 3rd 2011
5:30pm-7:00pm
Fairmont San Jose
Club Regent Room

Tickets available:
www.eetimes-ace.com

Good Luck Finalists!

NOMINATION BASED:

DESIGN TEAM OF THE YEAR:
Advanced Micro Devices
IBM
Intel Corporation
Nordic Semiconductor
PrimeSense

INNOVATOR OF THE YEAR:
Adapteva Inc.–Andreas Olofsson
Cirrus Logic–John Melanson
Intel–Mario Paniccia
NuPGA Corporation–Zvi Or-Bach
Wave Systems Corp.
–Robert Thibadeau

EXECUTIVE OF THE YEAR:
ARM–Warren East
Cirrus Logic–Jason Rhode
GLOBALFOUNDRIES
–Douglas Grose
Maxim Integrated Products
–Tunc Doluca
Spansion–John Kispert

STARTUP OF THE YEAR:
Dynamics Inc.
GLOBALFOUNDRIES
Lunera Lighting Inc.
Pixtronix
Semitech Semiconductor Pte Ltd

COMPANY OF THE YEAR:
Altera Corporation
ARM
Maxim Integrated Products
NetLogic Microsystems, Inc.
TriQuint Semiconductor, Inc.

MOST PROMISING NEW TECHNOLOGY:
Hillcrest Labs
InvenSense, Inc.
InVisage Technologies Inc.
Lyric Semiconductor
Tilera

ENERGY TECHNOLOGY AWARD:
Efficient Power Conversion Corporation
Freescale Semiconductor
SolarBridge Technologies
SPD Control Systems Corporation
Freescale Semiconductor

IEEE AWARD CATEGORIES:

IEEE SPECTRUM TECHNOLOGY IN THE SERVICE OF SOCIETY & THE IEEE SPECTRUM EMERGING TECHNOLOGY AWARD
Laster Technologies (Smart Spectacles)
Seabed Rig (Robotic Oil Driller)
IBM (Watson)
ClariPhy Communications (Digital Processor)
Willow Garage (Personal Robotics)

EE TIMES SELECTED CATEGORIES:

MOST POPULAR PRODUCT OF THE YEAR (7 CATEGORIES)
-Digital logic
-Analog-mixed signal
-Memories
-Interconnects
-Electro-mechanical
-Software/IP
-Test & Measurement

LIFETIME ACHIEVEMENT AWARD
EDITOR'S CHOICE AWARD
CONTRIBUTOR OF THE YEAR
MOST ENGAGED MEMER OF THE EE TIMES COMMUNITY OF THE YEAR
BEST STUDENT DESIGN/ DESIGN CHALLENGE WITH PUBLIC SCHOOLS
EE LIFE CONTRIBUTOR OF THE YEAR
STUDENT OF THE YEAR

UBM Electronics

GOLD SPONSORS

AVNET Embedded

MOUSER ELECTRONICS a tti company

ASSOCIATION MEDIA SPONSOR

ieee spectrum INSIDE TECHNOLOGY

By Jack G. Ganssle

A rumble, a wave, and iPads dry up

Anything I write about the recent tragedy in Japan will surely be superseded by events before this goes to press. But as of this writing, iSuppli reports that Shin-Etsu Chemical Company has stopped operations at their Shirakawa plant. The Shirakawa facility, which is about 100 km by air from the stricken Fukushima I Nuclear Power Plant, produces 20% of the global supply of 300-mm silicon wafers.

A press release from Shin-Etsu dated March 26, two weeks after the disaster, states they have not even been able to conduct an inspection of the facility and remain concerned that rolling blackouts will hinder operations for some time to come.[1]

Various reports in the press shrilly claim that 300-mm wafers are used for "all" processors and memory. That is simply not true; many vendors like Microchip and Atmel are still using 200-mm wafers.

It's impossible to predict what this will mean for our electronics industry. IC Insights claims that manufacturing capacity of chips went from 57% utilization in the first quarter of 2009 to 93% a year later, so the fabs are running nearly at capacity.[2] Other sources suggest that most fabs keep only a month's worth of wafers on-hand. A shortage could have significant effects.

We all know how bunny-suited technicians in ultra-clean rooms turn wafers into ICs. But where do the wafers come from?

Photo: Stockbyte

The disaster in Japan makes you think about the fundamental chemistry that makes all our modern smart devices possible.

BIGGER IS BETTER

First, silicon wafers are the substrate on which semiconductor vendors build most of their chips. They look like highly-polished flat disks. In ancient times, say 1975, state-of-the-art wafers were 100 mm in diameter. Today it's 300 mm, netting an order of magnitude more chips per wafer. The reality is much better, since process shrinks have shrunk feature sizes a hundredfold from the 3 µm used in 1975.

More chips on a wafer means higher profits for the semiconductor companies so they have strong motivation to increase diameters while shrinking feature sizes. But the jump from 200- to 300-mm wafers was a disaster. Costs far exceeded anyone's expectations. The next increment is the truly enormous 450 mm. Vendors have been long gun-shy due to the 300-mm debacle but are starting to invest the vast sums that will be necessary. Some expect first production next year.[3]

Currently there are about a hundred 300-mm fabs in the world, each costing a whopping $3 to $4 billion a pop. You could actually buy a handful of F-22s for that. 450-mm fabs will run a staggering $6 to $10 billion each.

TSMC has two 300-mm fabs with a combined capacity three million wafers per year. That's a lot of silicon. One source figures a 300-mm wafer costs about $250, or four times as much as a 200-mm wafer.[4]

Fabs don't make their own wafers, instead buying them from a handful of suppliers like Shin-Etsu.

14 PROTONS

Weird Al thought it was all about the Pentiums, but the story of semiconductors is all about the silicon (though other elements are used for some applications). I cringe when the uninformed so often talk about the "silicone" in semiconductors. Silicone is a complex polymer found in augmented breasts, not in integrated circuits.

Silicon, atomic weight 14, is in Group 14 in the periodic table. Car-

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. Contact him at jack@ganssle.com.

bon is directly above and germanium, tin, and lead below. It's somehow appropriate that carbon, the stuff of life, shares Group 14 with silicon, the stuff of electronics and smart products. Silicon is indeed life in some critters, like diatoms and radiolarians, which build their skeletons from silicon molecules. Silicon has been proposed as an alternative "organic" molecule, and there is some speculation that the earliest living proto-life was based on element 14.

Silicon exists mostly in the form of silica in nature, which is more appropriately known as silicon dioxide, or SiO_2. Our windows are mostly of silica, and they're made by floating the glass on molten tin, another member of Group 14.

In electronics, we need pure silicon, not SiO_2. It starts with sand, which is silica, usually extracted from quarries rather than the Copacabana. Some sources suggest that the beaches of Australia supply sand to our industry, which sounds like a great reason for an electronics engineer to visit Oz.

To extract the silicon, the sand and carbon-rich coal coke or wood are heated to 2,000°C, which "reduces" (removes the oxygen) from the silica:

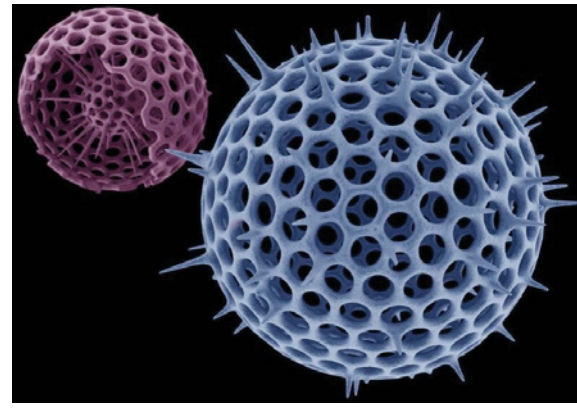
$$SiO_2 + C \rightarrow Si + CO_2$$

That step takes on the order of 12 to 14 KWh/Kg of silicon. That's a lot of energy. The result is about 98% pure "metallurgical grade" silicon, which is useless for making ICs. The wafer-makers can only tolerate about one foreign atom per billion silicons, a purity so absurd it just shouts about the hubris of engineers. But if one part per billion is what's needed, well, engineers deliver.

The 98% pure result is ground and converted into trichlorosilane by reacting it with hydrochloric acid at 300°C:

$$Si + 3\ HCL \rightarrow HSiCl_3 + H_2$$

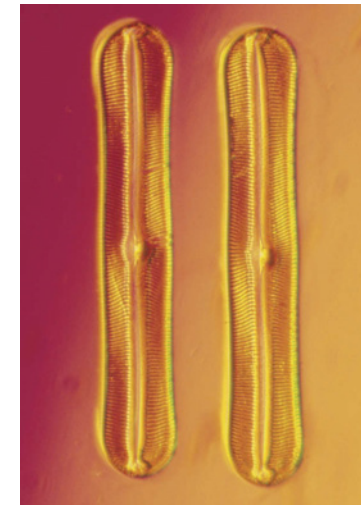
Trichlorosilane breaks down at 1,150°C:



Radiolaria shown in an artificially-colored photo from nasa.gov that was posted on from Wikipedia *(http://en.wikipedia.org/wiki/File:Radiolaria3434.JPG).*

Diatom: Pimularia novelie *© Getty Images; Credit: Comstock Images.*

Silicon is indeed life in some critters, like diatoms and radiolarians, which build their skeletons from silicon molecules.



$$2HSiCl_3 \rightarrow Si + 2HCL + SiCl_4$$

At this point it's very pure and is called electronics grade silicon. (Note that sometimes other very similar reactions are used to get the same result.)

IGNOTS

Silicon has to be in crystalline form for use in semiconductors. To make the job harder, it has to be monocrystalline—the crystal lattice must be unbroken and continuous. (Polycrystalline silicon, which is a jumble of crystals, is also used in electronics. About half of all polycrystalline Si goes into solar cells.)

Though there are a number of ways to create a monocrystalline silicon ingot, the Czochralski process is the most commonly used. The purified Si is heated in a quartz crucible (quartz is itself mostly SiO_2) to 1,420°C, which is close to the melting point of steel.

Dopants may be added. An element replaces one silicon atom in the lattice structure. Phosphorous has five valance electrons (one more than Si); four replace the covalent bonds that hold the lattice together and the fifth is an extra charge that's free to roam. Doping with phosphorous yields an N-type material. Boron, with three electrons in its outer ring, creates a "hole" and a P-type semiconductor.

A little doping goes a long way. Typically one dopant atom per 106 to 109 Si atoms is enough.

A single seed crystal of silicon attached to a puller rod is dipped into the molten Si with the crystal aligned in the same orientation required for the finished ingot. It's then pulled, ever so gently, back from the material. Surface tension causes some of the molten silicon to adhere to the crystal. Cooling, the atoms orient themselves to the crystal structure of the seed. The puller rod rotates in one direction and the crucible in the other.

For the large ingots that produce 300-mm wafers, the puller rod is extracted at a few cm/hr. Extreme care is

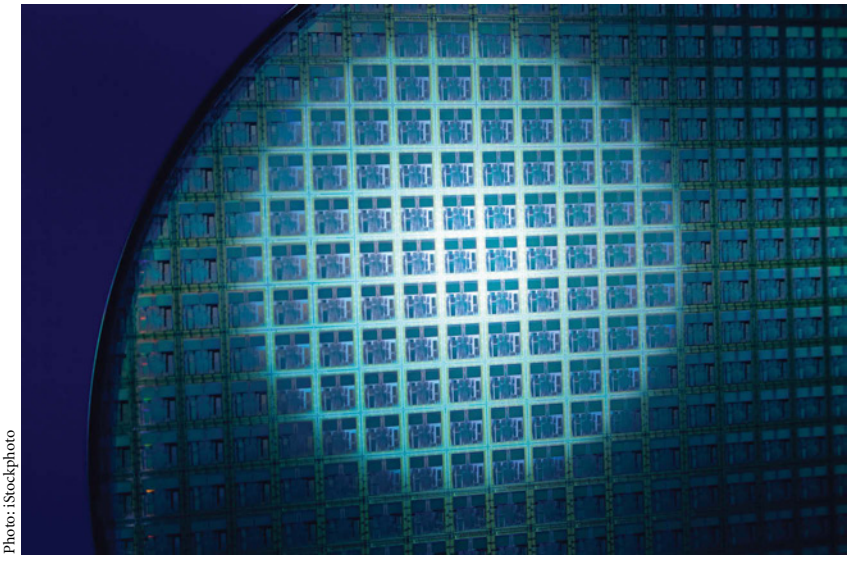


Photo: iStockphoto

The proto-chemists of the Middle Ages yearned to turn common materials into gold. But if there was ever alchemy, wafer production is it.

needed to avoid having the forming monocrystalline silicon break off. I imagine that if Shin-Etsu was pulling ingots during the earthquake they all would have failed.

The result is a single ingot somewhat over 300 mm in diameter and about 2-m tall. The lattice spacing is on the order of half a nanometer, or about twice the size of the Si atom.

SLICE AND DICE

The ingot is cleaned and a notch or flat is milled in it to indicate the crystal orientation. Now it's a long cylinder. But we want wafers, so a wire saw cuts the ingot.

The saw uses a single piece of wire that is threaded into a web that makes 500 or more slices per cut using an abrasive slurry (typically silicon carbide—yet another use of silicon).

About half the material is lost to the saw's kerf.

The finish is rough due to saw marks and other imperfections, so a lapping process smoothes the surface, thins the wafer, and relieves stress. It's etched a bit to alleviate microscopic cracks and lapping damage, and the edges are rounded to reduce the chance of breakage during later handling.

But the wafer is still far from perfect, and, when working at the crazy-small process nodes now common (32 nm today; 22 soon), perfection is required. The wafer goes through an extensive polishing process. Unlike smaller wafers, those in the 300-mm class are polished on both sides, producing a mirror finish. The use of polishing pads and slurry of increasing fineness eventually yields a finish roughness of under 0.5 nm.

Let's put that in perspective. Half a nanometer is the size of a pair of silicon atoms. Have you ever ground a telescope mirror? I made the mistake of doing that once, in college when money was especially scarce. Desired accuracy was a quarter wavelength of light, which is around 150 nm.

No wonder chipmakers wear bunny suits when handling wafers.

The wafer is cleaned and then inspected by an interferometer that may take several million data points. The final thickness is about 0.775 mm, and flatness is better than 40 µm.

Prime wafers are those that pass rigorous inspections and that are suitable for state-of-the art lithography. Test wafers are those that didn't make the grade but that are still useable for nonproduction work. Typically they have no flatness specification, may be scratched, and might be unpolished.

The proto-chemists of the Middle Ages yearned to turn common materials into gold. But if there was ever alchemy, wafer production is it. Worthless sand becomes a $250 wafer, which in turn is transformed into integrated circuits. A single wafer could produce $200,000 worth of Pentium processors.

JAPAN, RISING

Two Kamikazes blew 100 feet of the bow off of the ship my dad served on during World War II, so we kids grew up in a household that did not hold the Japanese in high regard. But when I first went to Japan, less than 30 years after that nation had been reduced to ruins, I was amazed to find Tokyo a completely rebuilt, vibrant hub of commerce. Since then I've been privileged to work with a number of brilliant Japanese engineers and managers, and have not the slightest doubt the people there will recover from the devastation. Shin-Etsu will—sooner than we can imagine—resume their miracle of turning sand into semiconductor gold. And I'm sure the rest of that blighted region will also rise from the ashes at a pace that will astonish us all. ■

ENDNOTES:

1. Shin-Etsu Chemical Co., Ltd. "Shin-Etsu Group current situation impacted by the 2011 off the Pacific Coast of Tohoku Earthquake (6th report)." Press release, March 26, 2011. Available at: *www.shinetsu.co.jp/e/news/s20110325.shtml.*
2. IC Insights. "IC Industry Consolidation Is Here!" June 3, 2010. Available at: *www.icinsights.com/news/bulletins/IC-Industry-Consolidation-Is-Here/*
3. Stokes, Jon. "Intel, Samsung, TSMC to hold hands, jump to new wafer size." Last updated 2 years ago: *http://arstechnica.com/hardware/news/2008/05/intel-samsung-tsmc-to-hold-hands-and-jump-to-new-wafer-size.ars*
4. Sage Concepts. "Report I Silicon Industry 2008 Summary: Consolidation" *www.sageconceptsonline.com/docs/report1.pdf*

FROM DESIGN INCEPTION TO FINAL PRODUCT IMPLEMENTATION, AND ALL DESTINATIONS IN BETWEEN...

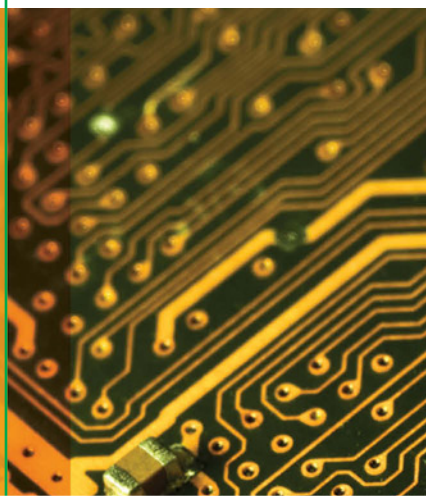
A N D R O I D ▪ N U C L E U S ▪ L I N U X

MENTOR EMBEDDED. Nobody assures success like Mentor Embedded. Whether you're a small design team or a large enterprise spread across the globe, only Mentor Embedded has the products, services, and expertise to help you develop your product. As a long-standing open source software contributor and maintainer of trusted device software and tools, we stand ready to assist you no matter where you are in the design cycle. We make it a point to deliver profitability, design efficiency, and innovation to you. To learn more visit **www.mentor.com/embedded.**

Mentor Graphics®

PCB Assembly
for R&D PROTOTYPES

The new standard for pcb assembly

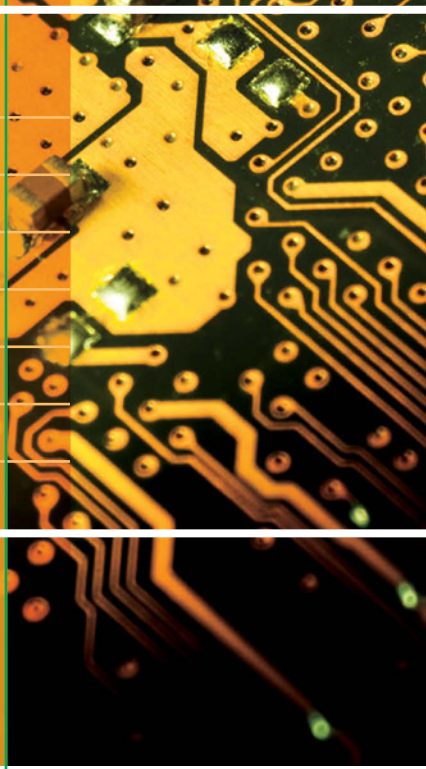


$50 in 5-Days

Advanced Assembly specializes in the machine assembly of low volume and prototype PCBs in 5 days or less. It is our only focus and we do it better and faster than CEM's, board fabricators, or your local assembly shops. Our assembly process and professional service have established a higher industry standard for quality in PCB assembly. Let us earn your business today.

R&D Assembly Pricing — Includes free tooling and programming

Number of SMT parts per board	1st Board	2nd Board	Boards 3-5 each	Stencil per side
1 through 25	$50	$35	$30	$75
26 through 50	$95	$65	$45	$75
51 through 100	$125	$85	$60	$75
101 through 150	$180	$120	$85	$75
151 through 200	$225	$150	$120	$100
201 through 250	$275	$190	$145	$100
251 through 400	Call	Call	Call	Call



- Machine-placed SMTs
- Parts in bulk, cut tape or reels
- Full turn-key or consignment
- Free digital image of your board before assembly

Advanced Assembly®
SMT ASSEMBLY FOR ENGINEERS℠

AAPCB.com/aa3
1.800.838.5650

IPC® Member

Lead Free

ISO 9001:2008 Certified