UBM

Learn today. Design tomorrow.

ESD

EMBEDDED SYSTEMS DESIGN

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

VOLUME 23, NUMBER 3

APRIL 2010

Code re-use enables Flexible Architecture

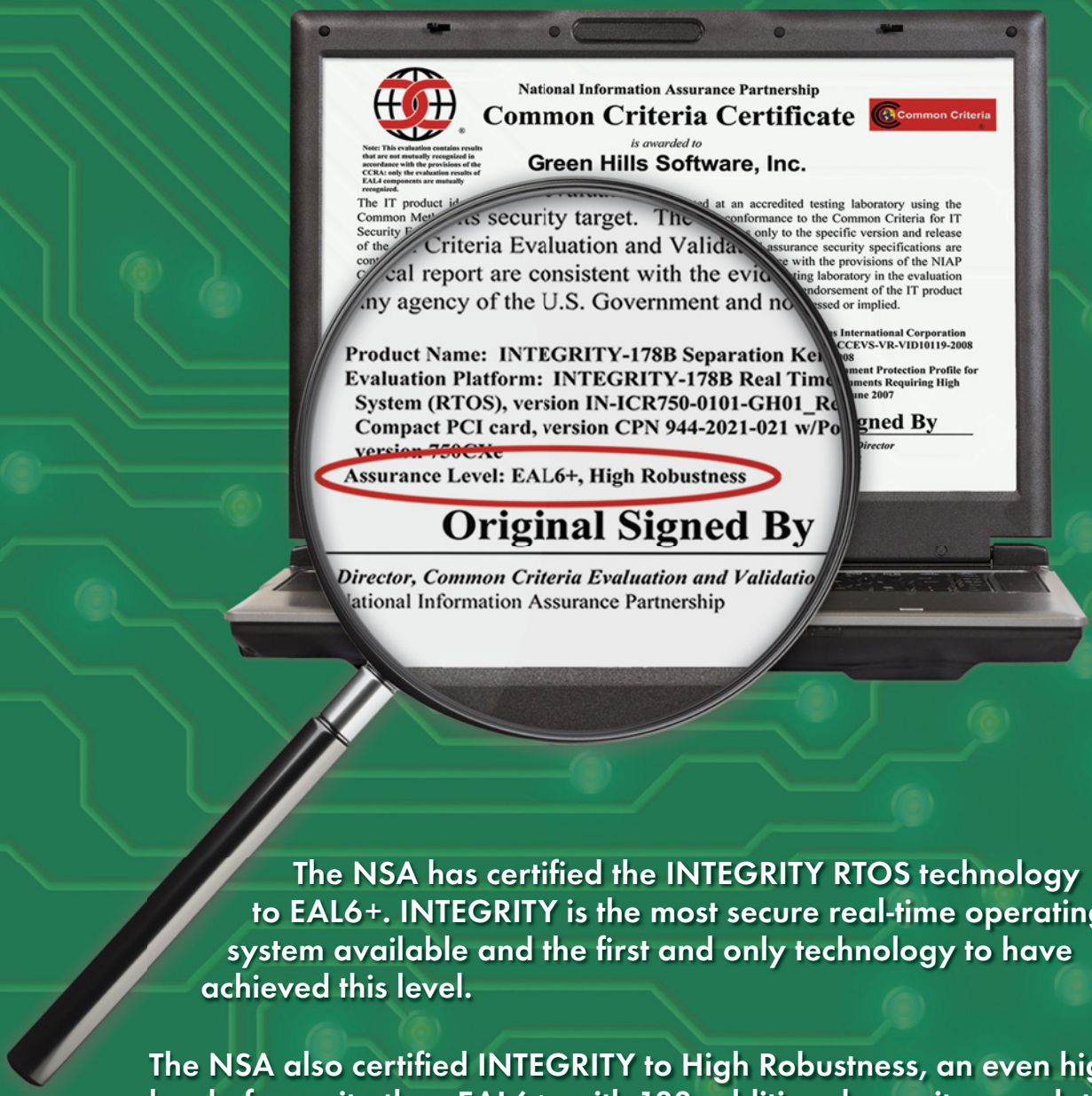
AS EMBEDDED TECHNOLOGY ADVANCES WE'RE RIGHT THERE WITH YOU

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

MENTOR EMBEDDED. Innovation is boundless. Mastering its possibilities requires a partner with the very latest in embedded software solutions. Mentor Embedded delivers solutions that reduce your development risk and shorten project cycles. We provide proven software and tools, along with services for Linux and Android development that enable our customers to succeed in emerging areas such as Android beyond mobile, multi-OS, multicore, and advanced 3D user interfaces. As the industry's leading independent vendor, Mentor Embedded stands ready for your next innovation. To learn more visit **www.mentor.com/embedded.**

Mentor Graphics®

INTEGRITY RTOS has it. No one else does.



The NSA has certified the INTEGRITY RTOS technology to EAL6+. INTEGRITY is the most secure real-time operating system available and the first and only technology to have achieved this level.

The NSA also certified INTEGRITY to High Robustness, an even higher level of security than EAL6+, with 133 additional security mandates over and above the 161 required for EAL6+.

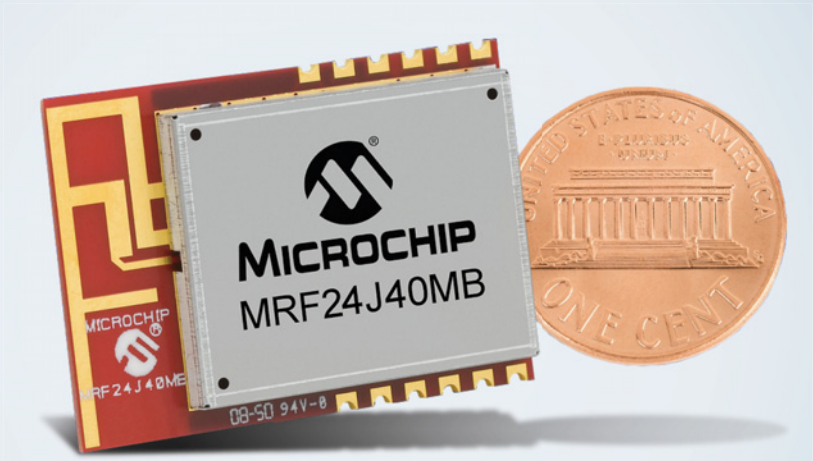
When security is required, Green Hills Software's INTEGRITY RTOS technology is the only option.

Green Hills® SOFTWARE, INC.

www.ghs.com

The Newest Products For Your Newest Designs

Embed Your Innovation into the Market Place.



MICROCHIP

MRF24J40MB 2.4 GHz RF Transceiver Module
mouser.com/microchipmrf24j40mb

FTDI Chip

USB-COM-PLUS Serial Modules
mouser.com/ftdiusbcomplus

AdaptivEnergy

JTRB-e12 Ground Transport Energy Harvesters
mouser.com/joulethieftransport

RABBIT

MiniCore™ RCM5600W Wi-Fi Module
mouser.com/rabbit_rcm5600w

WARNING: Designing with Hot, New Products May Cause A Time-to-Market Advantage.

With the newest embedded products and technologies you can get your designs to market faster. Experience Mouser's time-to-market advantage with no minimums and same-day shipping of the newest products from more than 400 leading suppliers.

MOUSER ELECTRONICS®

a tti company

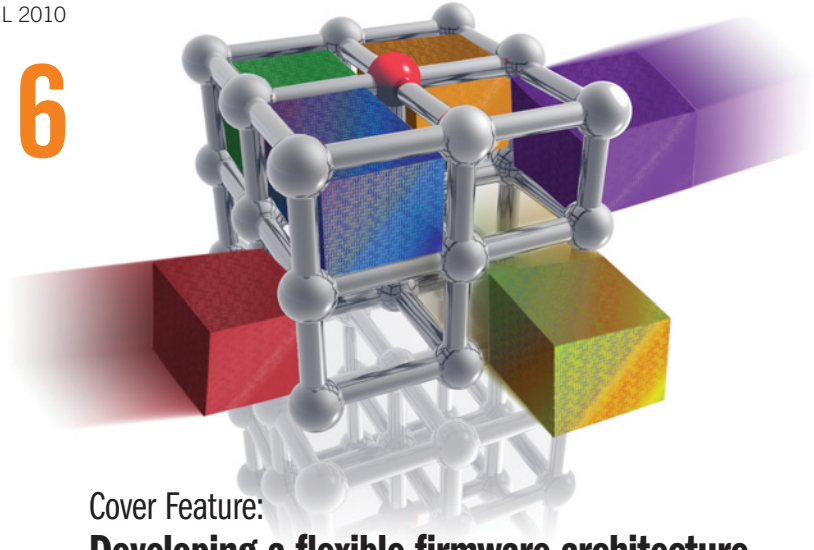
mouser.com **(800) 346-6873**

THE OFFICIAL PUBLICATION OF THE EMBEDDED SYSTEMS CONFERENCES AND EMBEDDED.COM

Learn today. Design tomorrow.

EMBEDDED SYSTEMS DESIGN

VOLUME 23, NUMBER 3
APRIL 2010



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, P.O. Box 3404, Northbrook, IL 60065-9468. For customer service, telephone toll-free (877) 676-9745. Please allow four to six weeks for change of address to take effect. Periodicals postage paid at San Francisco, CA and additional mailing offices. EMBEDDED SYSTEMS DESIGN is a registered trademark owned by the parent company, EE Times Group.

COLUMNS

DEPARTMENTS

IN PERSON

ESC Silicon Valley
April 26–29, 2010
www.embedded.com/esc/sv

ESC Chicago
June 8–9, 2010
esc-chicago.techinsightsevents.com/

ESC India
July 21–23, 2010
www.esc-india.com/

ESC Boston
September 20–23, 2010
www.embedded.com/esc/boston

ESC UK
October 12–14, 2010
www.embedded.co.uk

ONLINE

www.embedded.com

BUILD it [Reliably]

With Express Logic's award-winning BenchX® IDE or use tools from over 20 commercial offerings including those from ARM, Freescale, Green Hills, IAR, Microchip, MIPS, Renesas, and Wind River.

RUN it [Fast]

With Express Logic's small, fast, royalty-free and industry leading ThreadX® RTOS, NetX™ TCP/IP stack, FileX® FAT file system, and USBX™ USB stack.

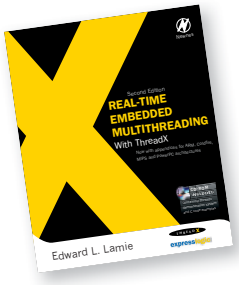
ANALYZE it [Easily]

With Express Logic's graphical TraceX® event analysis tool, and new StackX™ stack usage analysis tool. See exactly what is happening in your system, which is essential for both debugging and optimization.

SHIP it [Confidently]

No matter what "it" is you're developing, Express Logic's solutions will help you build it, analyze it, run it, and ship it better and in less time. Join the success of over 600,000,000 deployed products using Express Logic's ThreadX!

BENCHX THREADX TRACEX STACKX



expresslogic

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

ESD
EMBEDDED SYSTEMS DESIGN

Editorial Director
Richard Nass
(201) 288-1904
rich.nass@ubm.com

Managing Editor
Susan Rambo
susan.rambo@ubm.com

Contributing Editors
Michael Barr, John Canosa,
Jack W. Crenshaw, Jack G. Ganssle,
Dan Saks, Larry Mittag

Art Director
Debee Rommel
debee.rommel@ubm.com

European Correspondent
Colin Holland
colin.holland@ubm.com

Embedded.com Site Editor
Bernard Cole
bccole@acm.org

Production Director
Donna Ambrosino
dambrosino@ubm-us.com

Subscription Customer Service
P.O. Box 2165, Skokie, IL 60076
(800) 577-5356 (toll free)
Fax: (847) 763-9606
embeddedsystemsdesign@halldata.com
www.customerserviceesp.com

Article Reprints, E-prints, and Permissions
Mike O'Brien
Wright's Reprints
(877) 652-5295 (toll free)
(281) 419-5725 ext.117
Fax: (281) 419-5712
www.wrightsreprints.com/reprints/index.cfm?magid=2210

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

Editorial Review Board
Michael Barr, Jack W. Crenshaw,
Jack G. Ganssle, Bill Gatliff,
Nigel Jones, Niall Murphy, Dan Saks,
Miro Samek

EE Times Group

Corporate—EE Times Group

Paul Miller	*Chief Executive Officer*
Felicia Hamerman	*Group Marketing Director*
Brent Pearson	*Chief Information Officer*
Jean-Marie Enjuto	*Financial Director*
Amandeep Sandhu	*Manager Audience Engagement*
Barbara Couchois	*Vice President Sales Ops*

UBM

Corporate—UBM LLC

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

#include

BY Richard Nass

The results for 2010 are in!

If you've been around the embedded systems industry for any time, you're likely aware that the EE Times Group conducts an annual study of its readers, particularly those you who read *Embedded Systems Design* magazine, visit Embedded.com, or attend any of the global Embedded Systems Conferences. The results are now in, with more than 1,500 of you filing out the full study, which represents a pretty good sample.

In the study, we ask developers about the environment they work in and the design process they employ. Then we get into specific vendor choices, including which operating system, which processor, and so forth, they use or plan to use. In many instances, we go a few steps beyond the initial question and try to figure out why the answers are what they are.

The 2010 Embedded Market Study is complete and tallied. Big surprises? Yes and no.

The biggest takeaway from this year's study, for me at least, is that there was very little change from 2009 to 2010. However, one area that did have a significant change, thanks to the recession and (hopefully) the end of the recession, is the number of engineers working on the project at hand, specifically the software engineers. In 2008, development teams averaged 15.5 members (8.1 of whom were software engineers). In 2009, the number dropped to 13.6 (6.6 software). But in 2010, it rose to 18.9 (9.4 software). That's the highest number in at least five years.

Unfortunately, those extra folks didn't help you get your project finished on time. Like previous years, less than half of your projects were completed on time.

News that's not so good for the RTOS vendors is that the number of projects that employ internally developed operating systems is on the rise, from 26% last year to 32% this year (from 21% in 2008). The biggest hit there is against the commercial OSes, which fell from 47% to 38%.

The biggest surprise for me was in the question, "Please select all of the operating systems you are considering using in the next 12 months." Although, on the heels of the previous question, I probably should have been surprised. The number one choice was FreeRTOS (which can now be downloaded directly from Embedded.com, at *www.embedded.com/code.new/*, search by "Other": FreeRTOS is at the top of the list). The reason I was so astounded was that FreeRTOS didn't even show up in the study last year. That's quite a gain—from not on the chart to the number one position!

The rest of the study results will be released over the next few months. So stay tuned.

Richard Nass (rnass@techinsights.com) is the editorial director of* Embedded Systems Design *magazine, Embedded.com, and the Embedded Systems Conference.

Richard Nass
rnass@techinsights.com

336 Volts of Green Engineering

MEASURE IT – FIX IT

Developing a commercially viable fuel cell vehicle has been a significant challenge because of the considerable expense of designing and testing each new concept. With NI LabVIEW graphical programming and NI CompactRIO hardware, Ford quickly prototyped fuel cell control unit iterations, resulting in the world's first fuel cell plug-in hybrid.

MEASURE IT

Acquire
Acquire and measure data from any sensor or signal

Analyze
Analyze and extract information with signal processing

Present
Present data with HMIs, Web interfaces, and reports

FIX IT

Design
Design optimized control algorithms and systems

Prototype
Prototype designs on ready-to-run hardware

Deploy
Deploy to the hardware platform you choose

Ford is just one of many customers using the NI graphical system design platform to improve the world around them. Engineers and scientists in virtually every industry are creating new ways to measure and fix industrial machines and processes so they can do their jobs better and more efficiently. And, along the way, they are creating innovative solutions to address some of today's most pressing environmental issues.

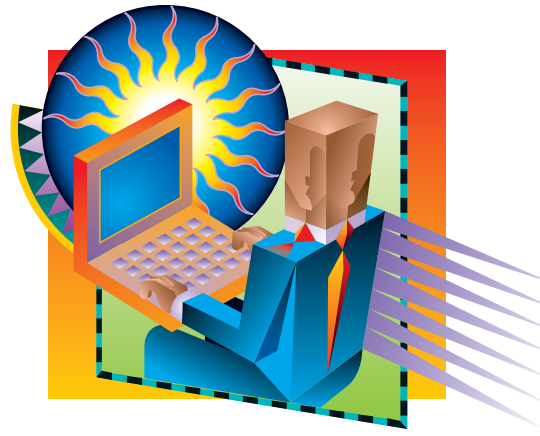
>> Download the Ford technical case study at **ni.com/336** **800 258 7018**

NATIONAL INSTRUMENTS™

Better resolution but increased accuracy?

In "Oversampling with averaging to increase ADC resolution," *(Franco Contadini, March 2010, p.19, available at www.embedded.com/223000385)* Franco Contadini describes how to get 16 bits of resolution from a 12-bit ADC, but it certainly doesn't guarantee 16 bits of accuracy. At the end of the day, accuracy is usually what we're after.

—Hardware Guy



You may get 16 bits of resolution from a 12-bit ADC, but it certainly doesn't guarantee 16-bits of accuracy.

A reference information for Figures 1 and 2 and supporting text would be very helpful. Values are stated as obvious, but not supported.

—casner
Sr. Staff Engineer

The author responds: *You can find the mathematical equation behind SNR value in the following tutorial at:* http://focus.ti.com/lit/an/slaa013/slaa013.pdf.

Look at *Chapter 5 QUANTIZATION EFFECTS.*

—Frank64 (Franco Contadini),

I am confused. The article says: "If we oversample an input signal at 16Fs, we collect enough samples within the required sampling period to average and produce 14 bits of output data, for a 14-bit measurement. This is accomplished by accumulating 16 consecutive samples and dividing the total by 16, as Figure 3 shows."

If I take, and add together, 16 12-bit samples and then divide by 16, I am left with 12-bits, not 14. What am I doing wrong?

It should be noted that this method fails to achieve gains in higher-bit resolution if the highest frequency of the signal being measured is not near the Nyquist rate of the sampling rate.

Dithering the signal with Gaussian or uniform electrical noise with levels near the last bit of resolution prior to the sampling will help get the extra bits of resolution for low-frequency signals through over sampling.

—PaulJr.
Senior Design EE

Complete 3D physics library

This discussion is all very useful *(Jack Crenshaw, "Random thoughts," March 2010, p.9, www.embedded.com/223000383)* and interesting, but if you want a complete 3D physics library, there's no need to reinvent the wheel. In his book, *Game Physics*, David H. Eberly has developed a very excellent and portable 3D physics libray in C++ for video games. In the book, he describes the background math and library implementation in a very readable way. You can get the source code in electronic from from the included CD or through his website.

—m1ke0
Engineer

Oscilloscope fan club

Great review! *(Jack Ganssle, "An MSO for the masses" March 2010, p.31, www.embedded.com/223000384)* I am pleased to have followed Tek since I ogled the ancient catalogs as a teenager and wished I could have all of the stuff (this was back when they had dual-beam scopes and the 519 with direct CRT plate drive). Then I used a 535 for years at UCLA, until finally some unspent year-end money popped up and I got a 475 and FET active probe, for a dramatic improvement in bandwidth and sensitivity.

Then I bought a 2236, with counter-timer-multimeter (including access from the Ch. 1 probe tip) for about 3k in 1985, and although some inevitable deterioration has set in, still serves. Based on 2010 dollars, the $5,400 asked for the scope reviewed is really a good deal.

There were some dark days at Tek, when they sent a lot of old-timers home and made some truly execrable scopes—I remember one that didn't even allow one channel to be triggered from the other! But eventually things righted themselves and they got back to engineering excellence.

—bcarso

Innovative Intelligent Integration

SmartFusion™

FPGA + ARM®Cortex™-M3 + Programmable Analog

Actel POWER MATTERS

Get Smart, visit: www.actel.com/smartfusion

Visit us at ESC Silicon Valley, Booth #930

More memories

I've just read "Remembering the Memories" (*Jack Ganssle, January/February 2010, p. , available at www.embedded.com/222400494*). Jack Ganssle's description of computers that used drum memory brought back some personal memories. My first computer, which I programmed in the summer of 1961, was an IBM 650. It had 2,000 (decimal) words (10 decimal digits plus sign) of memory arranged as 40 tracks of 50 words each. The heads were fixed (not even floating) and the 6-inch diameter drum rotated at 12,000 RPM. This arrangement caused some thermal issues when stopping and restarting the machine. Obviously, you didn't want head crashes during warm up. This meant that, after the drum stopped and while it was cooling, the drum and heads came in contact so that restarting would be disastrous.

Two additional drums were optional, as was a small amount of core memory (perhaps as much as 100 words). Another optional accessory was the RAMAC disk system. Unfortunately, I don't remember the capacity of RAMAC system.

Efficient programming was a real challenge. In order to avoid excess latency in accessing instructions, each instruction specified the location of its successor. In the case of branch instructions (all presumably conditional), each instruction specified two possible successors. Program loops were usually points at which instruction-placement optimization was relaxed, but such loops could be unrolled to minimize the loss—if you could afford the memory for the duplicated code.

There were table look up instructions that could search entire tracks in a single revolution. This was a great boon under the circumstances.

On the two 650s I used, all input and output was via punched cards, although I believe that a line-printing device might have been optional.

I still have one or more original manuals for this computer and the peripheral devices with which it was used.

—Ron Martin
Ann Arbor, MI

Jack Ganssle responds: *Ron, thanks for the write-up. Two whole generations have missed the vacuum tube era and take cheap computers for granted. When I was in college we had access to the campus' one machine (although only special people could ever actually see the thing), but no one could even dream of owning any sort of computer. How things have changed! All the best.*

Reverse engineering

The use of reverse engineering here is commendable (*"When good compilers go bad, or What you see is not what you execute," Paul Anderson and Thomas W. Reps, January/February, p.29, www.embedded.com/222400497*). I have long believed that we have to close the feedback loop and reverse engineer the code back into requirements (e.g., UML) in order to get automatic improvements. Georgia Tech has done a lot of reverse engineering work in this area.

What I don't understand is why these guys are still walking the streets. The "Patriot Act" and the DCMA should have put them out of the reverse engineering business long ago.

—stevev6
Software Engineer

What I don't understand is why these guys are still walking the streets. The "Patriot Act" and the DCMA should have put them out of the reverse engineering business long ago.

Part of my purpose is to eliminate the need for the kind of tools in the reference. If there is no "typical" compile process that generates code that is taken out in optimization, but rather a simulation/emulation that shows the execution process you truly see what you get. So far the responses range from "Huh What?" to "It's just another C2H gadget." Technically, it's a compiler although that brings the assumption of generate assembly, optimize, and so on. What really happens is that the RAM contents are microprogram controls and references to RAM locations that contain data/variables and control bits for the next cycles operation. The data flow is designed to allow a straightforward conversion from C code to "machine language."

In one other comment, I mentioned that a compiler can do unexpected things and immediately got a reply that the compiler only does exactly what it is told. Now we have this article describing research and tools to find such things. I still have bugs in the function call but should soon have a sample output that shows how few cycles it takes to execute C statements. The objective is to minimize the functions that currently are done in HDL because of time constraints, therefore those applications that have an acceptable amount of HDL do not need my design.

So about the costs: I see this as another custom component in something like Altera's SOPC builder, so it is just another block that can be selected and more or less connected automatically just as their NiosII processor. You no doubt have other concerns, but until I at least get the demo running to show the potential performance cost trade offs cannot be done.

—KarlS, CEngine Developer

We welcome your feedback. Letters to the editor may be edited. Send your comments to Richard Nass at rich.nass@ubm. com or fill out one of our feedback forms online, under the article you wish to discuss.

By Michael Barr

Five top causes of nasty embedded software bugs

Finding and killing latent bugs in embedded software is a difficult business. Heroic efforts and expensive tools are often required to trace backward from an observed crash, hang, or other unplanned run-time behavior to the root cause. In the worst cases, the root cause damages the code or data in a way that the system still appears to work fine or mostly fine—at least for a while.

Too often engineers give up trying to discover the cause of infrequent anomalies that cannot be easily reproduced in the lab—dismissing them as user errors or "glitches." Yet these ghosts in the machine live on. Here's a guide to the most frequent root causes of difficult to reproduce bugs. Look for these top five bugs whenever you are reading firmware source code. And follow the recommended best practices to prevent them from happening to you again.

Too often engineers give up trying to discover the cause of infrequent anomalies not easily reproduced in the lab. Yet these ghosts in the machine live on.

BUG 1: RACE CONDITION

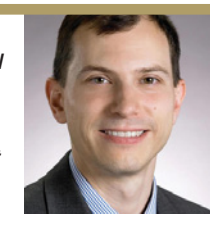
A race condition is any situation in which the combined outcome of two or more threads of execution (which can be either RTOS tasks or `main()` and an interrupt handler) varies depending on the precise order in which the interleaved instructions of each are executed on the processor.

For example, suppose you have two threads of execution in which one regularly increments a global variable (`g_counter += 1;`) and the other very occasionally zeroes it (`g_counter = 0;`). There is a race condition here if the increment cannot always be executed atomically (in other words, in a single instruction cycle). Think of the tasks as cars approaching the same intersection,
as illustrated in **Figure 1**. A collision between the two updates of the counter variable may never or only very rarely occur. But when it does, the counter will not actually be zeroed in memory that time; its value is corrupt at least until the next zeroing. The effect of this may have serious consequences for the system, although perhaps not until a long time after the actual collision.

■ **Best practice:** Race conditions can be prevented by surrounding critical sections of code that must be executed atomically with an appropriate preemption-limiting pair of behaviors. To prevent a race condition involving an ISR, at least one interrupt signal must be disabled for the duration of the other code's critical section. In the case of a race between RTOS tasks, the best practice is the creation of a mutex specific to that shared object, which each task must acquire before entering the critical section. Note that it is not a good idea to rely on the capabilities of a specific CPU to ensure atomicity, as that only prevents the race condition until a change of compiler or CPU.

Shared data and the random timing of preemption are culprits that cause the race condition. But the error might not always occur, making the tracking of race conditions from observed symptoms to root causes incredibly difficult. It is, therefore, important to be ever-vigilant about protecting all shared objects. Each shared object is an accident waiting to happen.

Michael Barr is the author of three books and over 50 articles about embedded systems design, as well as a former editor in chief of this magazine. Michael is also a popular speaker at the Embedded Systems Conference and the founder of embedded systems consultancy Netrino. You may reach him at mbarr@netrino.com or read more by him at www.embeddedgurus.net/barr-code.




XILINX
FASTER THAN THE SPEED OF CHANGE
The path to true innovation is never a straight line. Only Xilinx programmable silicon, software, IP and 3rd party support gives you the agility to stay ahead of the competition and adapt to changing market requirements, without slowing down. So you have the freedom to innovate without risk. Find out more about Xilinx Targeted Design Platforms at www.xilinx.com.
Visit Xilinx at ESC 2010!
Booth #1716 | April 27 - 29, 2010
McEnery Convention Center | San Jose, CA
© Copyright 2010 Xilinx, Inc. All rights reserved. Xilinx and the Xilinx logo are registered trademarks of Xilinx in the United States and other countries. All other trademarks are property of their respective holders.

The run-time errors caused by a non-reentrant function generally don't occur in a reproducible way—making them hard to debug. A non-reentrant function is also more difficult to spot in a code review than other types of race conditions.

- **Best practice:** Name all potentially shared objects—including global variables, heap objects, or peripheral registers and pointers to the same—in a way that the risk is immediately obvious to every future reader of the code; the *Netrino Embedded C Coding Standard* advocates the use of a "g_" prefix for this purpose. Locating all potentially shared objects would be the first step in a code audit for race conditions.

BUG 2: NON-REENTRANT FUNCTION

Technically speaking, the problem of a non-reentrant function is a special case of the problem of a race condition. And, for related reasons, the run-time errors caused by a non-reentrant function generally don't occur in a reproducible way—making them just as hard to debug. Unfortunately, a non-reentrant function is also more difficult to spot in a code review than other types of race conditions.

Figure 2 shows a typical scenario. Here the software entities subject to preemption are also RTOS tasks. But rather than manipulating a shared object directly, they do so by way of function call indirection. For example, suppose that Task A calls a sockets-layer protocol function, which calls a TCP-layer protocol function, which calls an IP-layer protocol function, which calls an Ethernet driver. In order for the system to behave reliably, all of these functions must be reentrant.

But all of the functions of the Ethernet driver manipulate the same global object in the form of the registers of the Ethernet controller chip. If preemption is permitted during these register manipulations, Task B may preempt Task A after Packet A has been queued but before the transmit is begun. Then Task B calls the sockets-layer function, which calls the TCP-layer function, which calls the IP-layer function, which calls the Ethernet driver, which queues and transmits Packet B. When control of the CPU returns to Task A, it requests a transmission. Depending on the design of the Ethernet con-

Each shared resource is an accident waiting to happen.

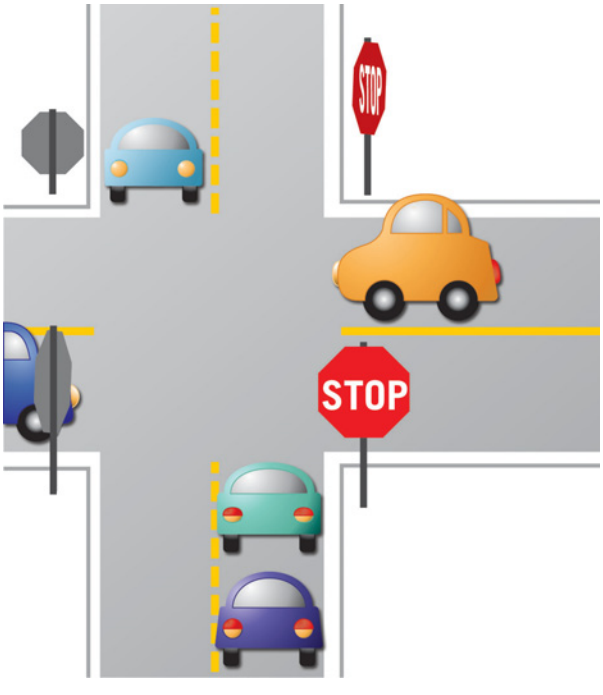


Figure 1

Libraries and device drivers can harbor non-reentrant functions.

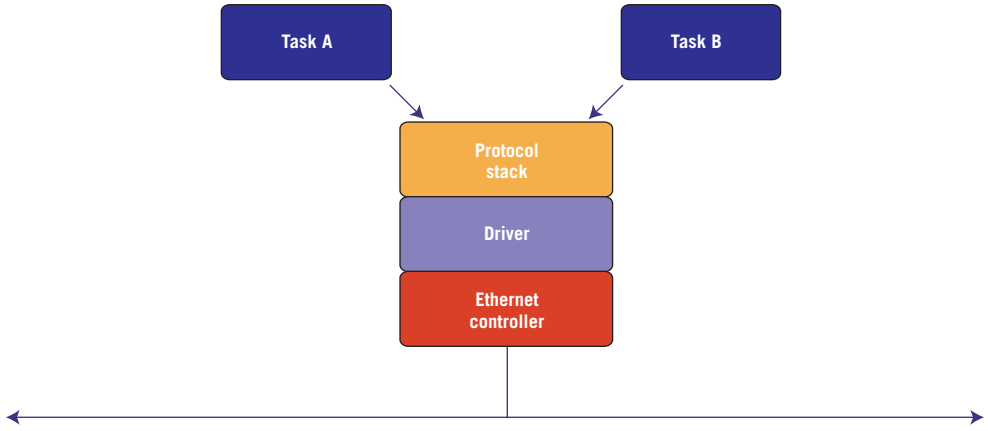


Figure 2

troller chip, this may either retransmit Packet B or generate an error. Packet A is lost and does not go out onto the network.

In order for the functions of this Ethernet driver to be callable from multiple RTOS tasks near-simultaneously, they must be made reentrant. If they each use only stack variables, there is nothing to do. Thus the most common style for C functions is inherently reentrant. But drivers and some other functions will be non-reentrant unless carefully designed.

The key to making functions reentrant is to suspend preemption around all accesses of peripheral registers, global variables including static local variables, persistent heap objects, and shared memory areas. This can be done either by disabling one or more interrupts or by acquiring and releasing a mutex. The specifics of the problem dictate the best solution.

- **Best practice:** Create and hide a mutex within each library or driver module that is not intrinsically reentrant. Make acquisition of this mutex a precondition for the manipulation of any persistent data or shared registers used within the module as a whole. For example, the same mutex may be used to prevent race conditions involving both the Ethernet controller registers and a global or static local packet counter. All functions in the module that access this data, must follow the protocol to acquire the mutex before manipulating these objects.

  Beware that non-reentrant functions may come into your code base as part of third-party middleware, legacy code, or device drivers. Disturbingly, non-reentrant functions may even be part of the standard C or C++ library provided with your compiler. If you are using the GNU compiler to build RTOS-based applications, take note that you should be using the reentrant "newlib" standard C library rather than the default.

BUG 3: MISSING VOLATILE KEYWORD

Failure to tag certain types of variables with C's `volatile` keyword can cause a number of unexpected behaviors in a system that works properly only when the compiler's opti-

If you write code like this, the optimizer may try to make your program both faster and smaller by eliminating the first line—to the detriment of the patient's health.

mizer is set to a low level or disabled. The `volatile` qualifier is used during variable declarations, where its purpose is to prevent optimization of the reads and writes of that variable.

For example, if you write code like that in **Listing 1**, the optimizer may try to make your program both faster and smaller by eliminating the first line—to the detriment of the patient's health. If, however, `g_alarm` is declared as `volatile`, then this optimization will not be permitted.

- **Best practice:** The `volatile` keyword should be used to declare every:
  - Global variable accessed by an ISR and any other part of the code,
  - Global variable accessed by two or more RTOS tasks (even when race conditions in those accesses have been prevented),
  - Pointer to a memory-mapped peripheral register (or set or registers), and
  - Delay loop counter.

Note that in addition to ensuring all reads and writes take place for a given variable, the use of `volatile` also constrains the compiler by adding additional "sequence points." Other volatile accesses above the read or write of a volatile variable must be executed prior to that access.

BUG 4: STACK OVERFLOW

Every programmer knows that a stack overflow is a Very Bad Thing. The effect of each stack overflow varies, though. The nature of the damage and the timing of the misbehavior depend entirely on which data or instructions are clobbered and how they are used. Importantly, the length of time between a stack overflow and its negative effects on the system depends on how long it is before the clobbered bits are used.

Unfortunately, stack overflow afflicts embedded systems far more often than it does desktop computers. This is for

Listing 1

```
g_alarm = ALARM_ON;        // Patient dying; alert a nurse.
                           // Other code; with no reads of g_alarm state.
g_alarm = ALARM_OFF;       // Patient stable.
```

Fixed-sized memory pools don't suffer fragmentation.

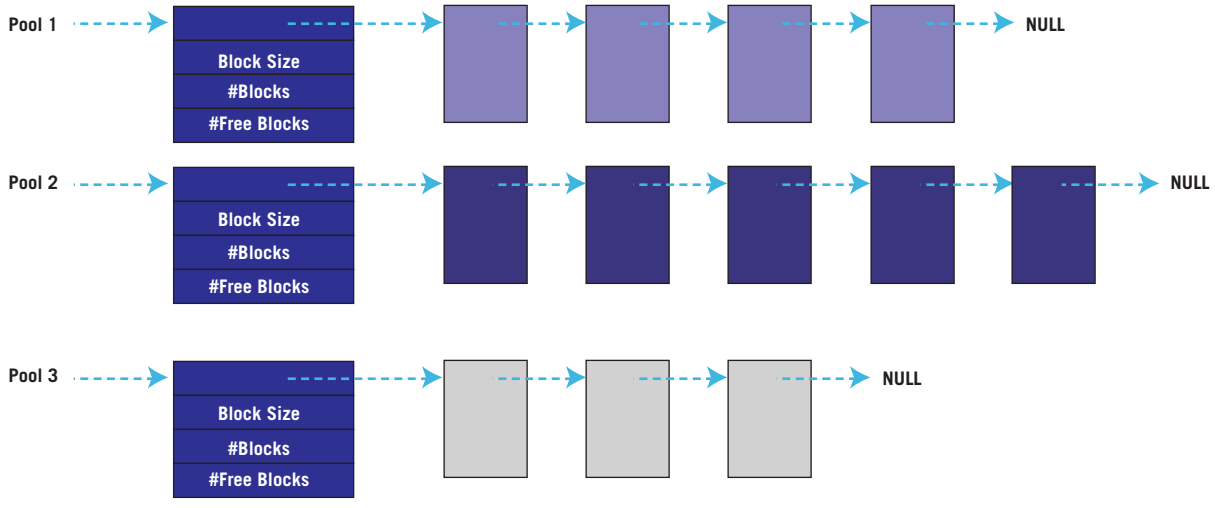


Figure 3

several reasons, including: (1) embedded systems usually have to get by on a smaller amount of RAM; (2) there is typically no virtual memory to fall back on (because there is no disk); (3) firmware designs based on RTOS tasks utilize multiple stacks (one per task), each of which must be sized sufficiently to ensure against unique worst-case stack depth; and (4) interrupt handlers may try to use those same stacks.

Further complicating this issue, no amount of testing can ensure that a particular stack is sufficiently large. You can test your system under all sorts of loading conditions but you can only test it for so long. A stack overflow that only occurs "once in a blue moon" may not be witnessed by tests that run for only "half a blue moon." Demonstrating that a stack overflow will never occur can, under algorithmic limitations (such as no recursion), be done with a top-down analysis of the control flow of the code. But a top-down analysis will need to be redone every time the code is changed.

- **Best practice:** On startup, paint an unlikely memory pattern throughout the stack(s). (I like to use hex 23 3D 3D 23, which looks like a fence '#==#' in an ASCII memory dump.) At runtime, have a supervisor task periodically check that none of the paint above some pre-established high water mark has been changed. If something is found to be amiss with a stack, log the specific error (such as which stack and how high the flood) in nonvolatile memory and do something safe for users of the product (for example, a controlled shut down or reset) before a true overflow can occur. This is a nice additional safety feature to add to the watchdog task.

Avoiding all use of the heap is a sure way of preventing this bug. But if dynamic memory allocation is either necessary or convenient in your system, there is an alternative way.

BUG 5: HEAP FRAGMENTATION

Dynamic memory allocation is not widely used by embedded software developers—and for good reasons. One of those is the problem of fragmentation of the heap.

All data structures created via C's `malloc()` standard library routine or C++'s `new` keyword live on the heap. The heap is a specific area in RAM of a predetermined maximum size. Initially, each allocation from the heap reduces the amount of remaining "free" space by the same number of bytes. For example, the heap in a particular system might span 10 KB starting from address 0x20200000. An allocation of a pair of 4-KB data structures would leave 2 KB of free space.

The storage for data structures that are no longer needed can be returned to the heap by a call to `free()` or use of the `delete` keyword. In theory this makes that storage space available for reuse during subsequent allocations. But the order of allocations and deletions is generally at least pseudo-random—leading the heap to become a mess of smaller fragments.

To see how fragmentation can be a problem, consider what would happen if the first of the above 4 KB data structures is free. Now the heap consists of one 4-KB free chunk and another 2-KB free chunk; they are not adjacent and cannot be combined. So our heap is already fragmented. Despite 6 KB of total free space, allocations of more than 4 KB will fail.

Fragmentation is similar to entropy: both increase over time. In a long running system (in other words, most every embedded system ever created), fragmentation may eventually cause some allocation requests to fail. And what then? How should your firmware handle the case of a failed heap allocation request?

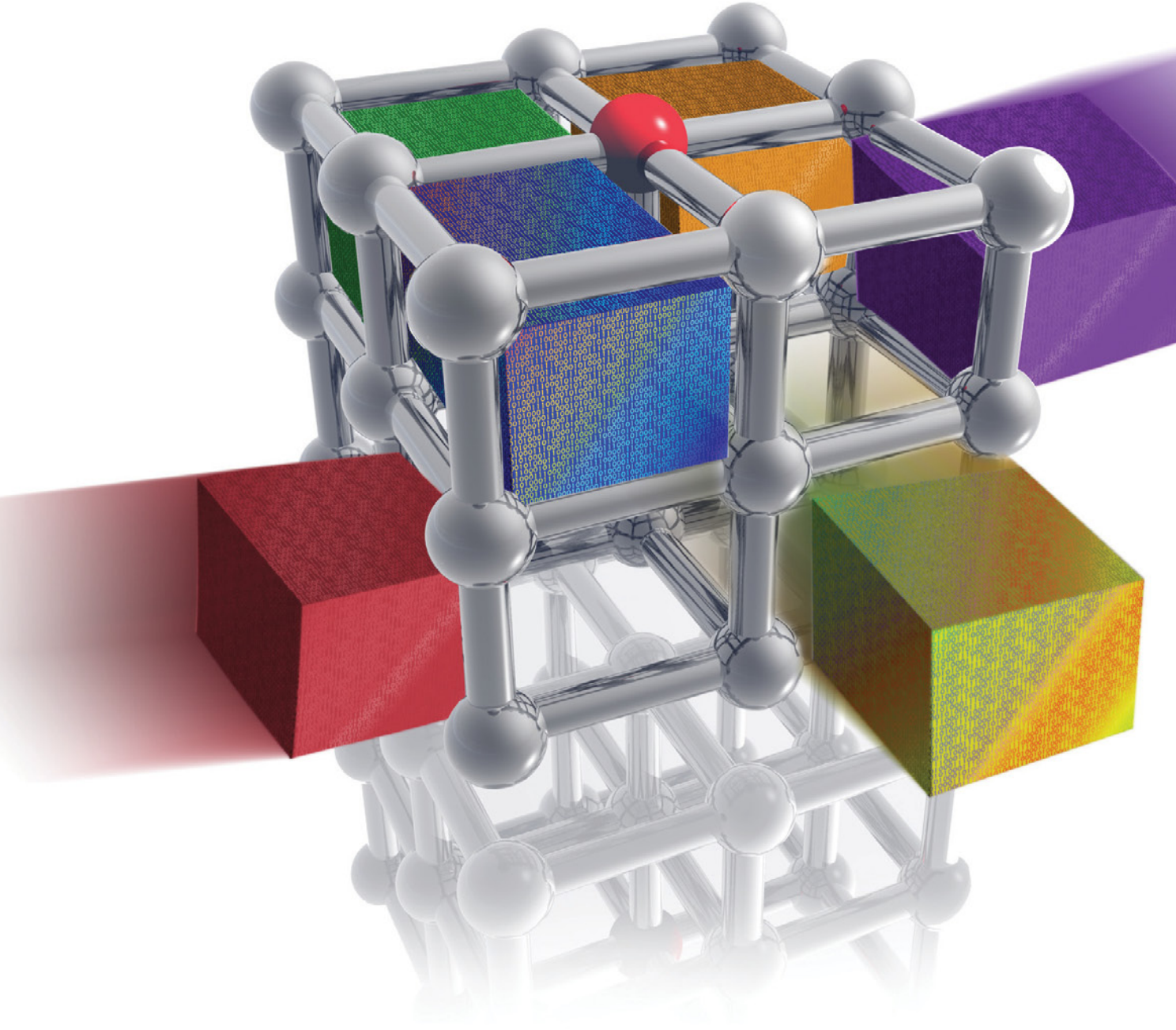
- **Best practice:** Avoiding all use of the heap is a sure way of preventing this bug. But if dynamic memory allocation is either necessary or convenient in your system, there is an alternative way of structuring the heap that will prevent fragmentation. The key observation is that the problem is caused by variable sized requests. If all of the requests were of the same size, then any free block is as good as any other—even if it happens not to be adjacent to any of the other free blocks. **Figure 3** shows how the use of multiple "heaps"—each for allocation requests of a specific size—can be implemented as a "memory pool" data structure.

Many real-time operating systems feature a fixed-size memory pool API. If you have access to one of those, use it instead of `malloc()` and `free()`. Or write your own fixed-sized memory pool API. You'll just need three functions: one to create a new pool (of size *M* chunks by *N* bytes); another to allocate one chunk (from a specified pool); and a third to replace `free()`.

CODE REVIEW IS STILL THE BEST PRACTICE

You can save yourself a lot of debugging headaches by ensuring that none of these bugs is present in your system in the first place. The best way to do that is to have someone inside or outside your company perform a thorough code review. Coding standard rules that mandate the use of the best practices I've described here should also be helpful. If you suspect you have one of these nasty bugs in existing code, performing a code review may prove faster than trying to trace backward from an observed glitch to the root cause.

In an upcoming column, I'll introduce you to a few other common causes of nasty embedded software bugs. ■



Firmware modularization is the ticket to low cost and (relatively) painless product development.

Developing a flexible firmware architecture

BY VINCENT CAI

One of the goals for embedded systems developers is to create firmware in a programming environment that supports and enhances low bill-of-materials cost, software reliability, and fast development time. The best way to achieve such a programming environment is to use a unified firmware architecture that acts as the skeleton during product development and supports "firmware modularization."

A design architecture that incorporates the right mix of firmware modularization, testability, and compatibility can then be applied to any firmware development project to maximize code reusability, speed up firmware debugging, and increase firmware portability.

FIRMWARE MODULARIZATION?

Modular programming breaks down program functions into firmware modules, each of which performs a single function and contains all the source code and variables needed to complete that function (**Figure 1** shows an example).

Modularity helps to coordinate the work of many people on a team, manage interdependency between various parts of a project, and enable designers to assemble complex systems in a reliable way. Specifically, it helps designer achieve and manage complexity. As applications grow in size and functionality, modularity is necessary to separate them into individual pieces (whether as "components," "modules," or "plugins"). Each such separated piece then becomes one element of the modular architecture. As such, each piece can be isolated and accessed using a well-defined interface. In addition, modular programming improves firmware readability while simplifying firmware debugging, testing, and maintenance.

Modular programming is a key component of realizing code reusability. All the source code and variables needed to accomplish the specific function are integrated into a single independent source file and can be easily reused in other projects. Modular programming also speeds

firmware development. Over time, developers can create a library of firmware modules, with every module's function and interface clearly defined. To create a new system, the firmware engineer only has to build up the firmware system with these modules in a manner similar to how engineers build up the hardware system with individual components.

FIRMWARE MODULES PRINCIPLES

The basic concept of modular programming in firmware development is to create *firmware modules*. Conceptually, modules represent *separation of concerns* (SoC). In computer science, SoC is the process of breaking a computer program into distinct features that rarely overlap in functionality. A *concern* is any piece of interest, or function, of a program and is synonymous with features or behaviors. Progress toward SoC is traditionally achieved through modularity and encapsulation.

Firmware modules can be categorized into several types. For example:

- Code related to each user module is implemented as an individual firmware module. For example, Um_Adc.c is the firmware module for an ADC user module and Um_Timer.c is the firmware module for a Timer user module.
- Code for specific pure software algorithms is implemented as an individual firmware module. For example, Alg_SoftFilter.c is the firmware module to perform software filters such as a median filter, mean filter, or weighted mean filter.
- Code for a specific application is implemented as an individual firmware module. For example, App_BatteryCharger is the firmware module for a battery charger application.
- Code for a specific tool is implemented as an individual firmware module. For example, Tool_Printf.c is the firmware module to implement printing features.

There are general principles to keep in mind when implementing firmware modules:

- All module-related functions should be integrated into a single source file.
- Have a header file that declares all the resources (MACROs/constants/variables/functions) of the FW module.
- Include self-test code section within the source file to implement all the self-test functions of a FW module.
- The interfaces of FW module should be well-designed and defined.
- Since firmware is dependent on hardware, hardware dependencies need to be mentioned clearly in the source file header.
- Often, FW modules are available for other team members to use in other

A firmware architecture with modular programming.

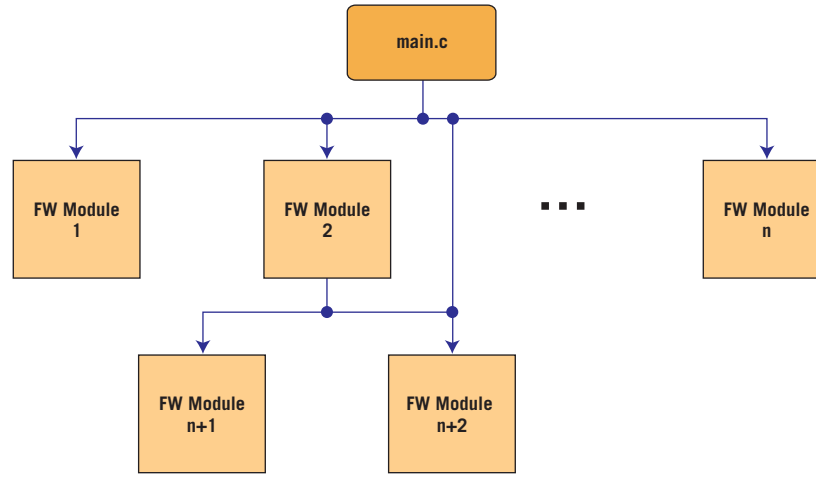
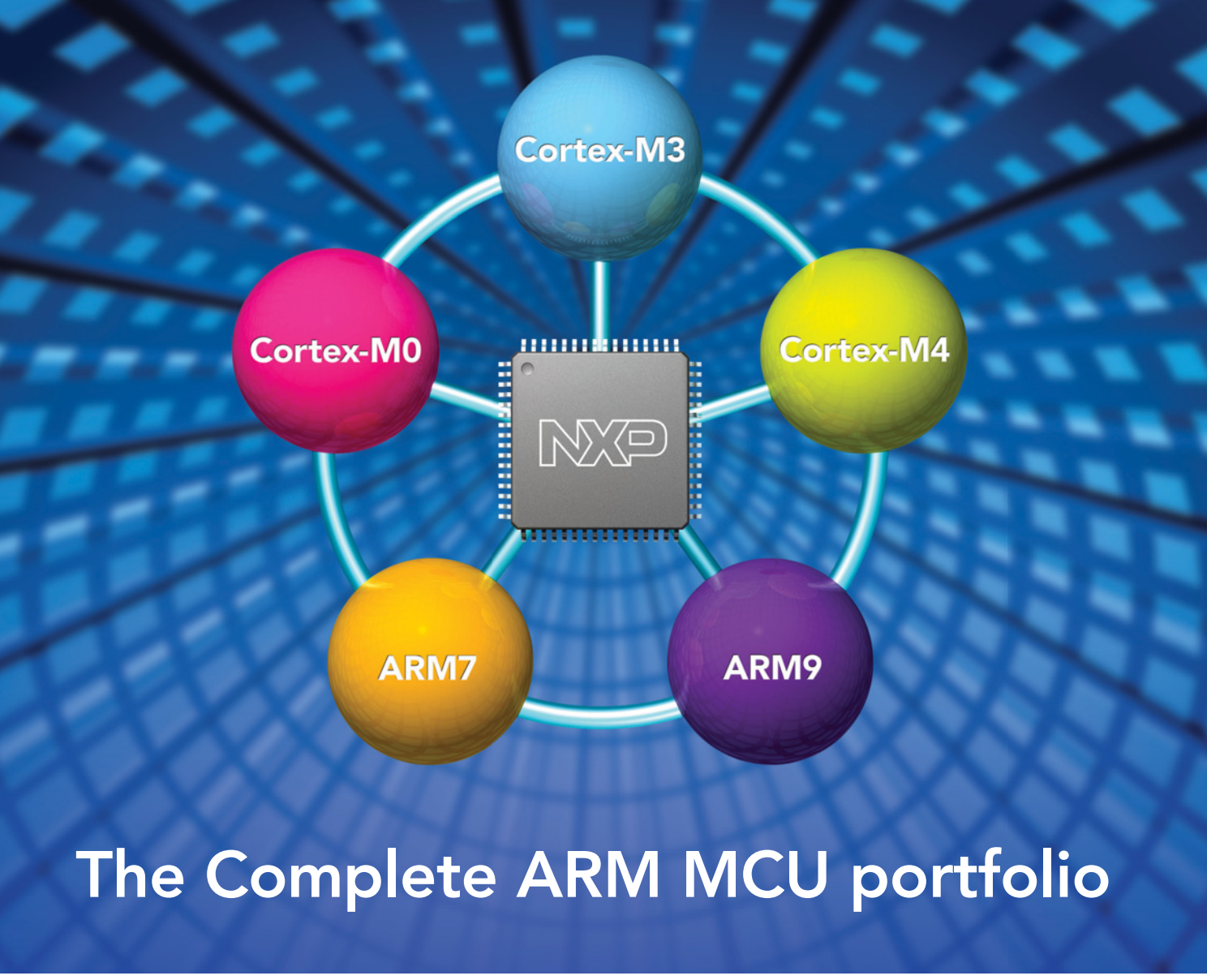


Figure 1



The Complete ARM MCU portfolio

NXP Semiconductors offers a comprehensive line of industry-leading microcontrollers — including the latest in Cortex™ based microcontrollers.

From low cost and low power, to small footprint and high performance, NXP has an ARM-based microcontroller to fit any embedded application. Our ARM microcontrollers are pin- and software-compatible across the cores, so you have more options in finding exactly what you need.

Visit the NXP LPC1100 Design Challenge to view what other designers are currently creating with NXP microcontrollers. Go to www.LPC1100challenge.com for more details.

Provide feedback on the latest LPC1100 designs and you could win a $1000 Apple™ gift card. Log on today!

www.LPC1100challenge.com

www.nxp.com/microcontrollers

NXP

projects. To manage changes, an owner should maintain the module. The source file header should contain "owner" and "version" information.

- Firmware is dependent on the compiler to some extent. A "code tested with" item should be in the source file header to specify compiler or IDE-related information.

Firmware modularization does not come without cost. Modularization may introduce extra layers of function calls due to encapsulation as well as increase overall code size. However, the improved portability and reusability could be an acceptable tradeoff. To optimize code size, pay more attention to related technologies and accumulate more experience instead of throwing away FW modularization. In regards to code execution speed, encapsulation offers minimal overhead: with a CPU clock running at 24 MHz, one call depth consumes less than 1 µs, which can be ignored except for timing critical cases. For timing-critical cases, if necessary, the timing-critical code could be encapsulated into a special FW module to minimize overhead.

For detailed information and specific examples, refer to the source code associated with this article, CyTemplate_FwModule.c and CyTemplate_FwModule.h, which are FW module template files for Cypress PSoC firmware design and can be easily applied to other MCUs.

SELF-TESTABILITY

As firmware is developed, temporary test functions and flows may become necessary to implement directly within subroutines or functions, particularly when debugging system-level problems or complex designs with many function dependencies. There are several drawbacks to this method. First, the test code modifies the original source code, potentially introducing new bugs. Secondly, such test code is difficult to preserve for later use. Because the same problem or bug may recur, if test code is not preserved, developers will have to write the same test code repeatedly.

Alternatively, developers can build self-test into their firmware architecture. Such an evolved firmware architecture is shown in **Figure 2**.

Compared with Figure 1, this

A firmware architecture with modular programming and self-testability.

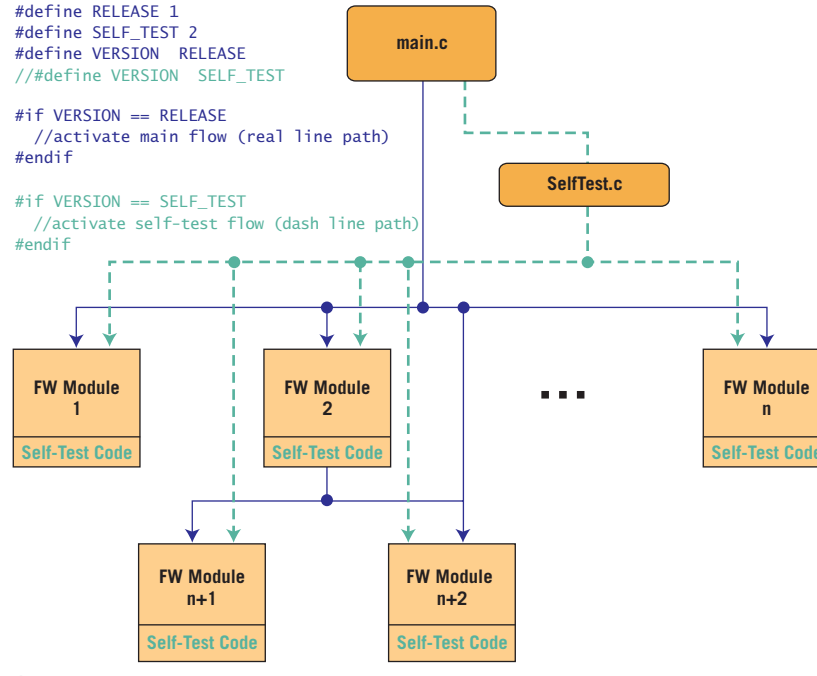
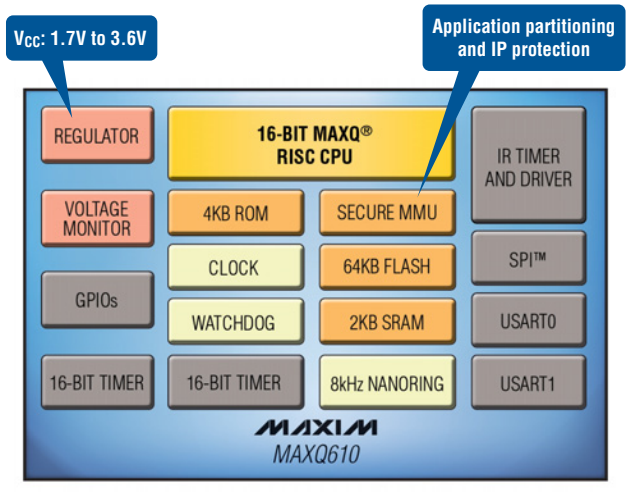


Figure 2

Extend battery life with the MAXQ610 16-bit microcontroller

Up to 12 MIPS in under 4mA—industry's highest MIPS/mA!

The MAXQ610 microcontroller is designed for low-cost, high-performance, battery-powered applications. This 16-bit, RISC-based microcontroller has a wide operating range (down to 1.7V) for long battery life and ultra-low power consumption. Its anticloning features and secure MMU enable you to protect your IP.



Microcontroller

- 16-bit MAXQ RISC core
- 64KB flash memory, 2KB SRAM
- Ultra-low supply current
  - Active mode: 3.75mA at 12MHz
  - Stop mode: 200nA (typ), 2.0µA (max)
- Wide, 1.7V to 3.6V operating voltage range
- IP protection
  - Secure MMU supports multiple privilege levels, helps protect code from copying and reverse engineering

Peripherals

- Two USARTs and one SPI master/slave communication port
- Two 16-bit timers/counters
- 8kHz nanoring functions as programmable wakeup timer
- IR timer simplifies support for low-speed infrared communication
- IR driver capable of 25mA sink current

Part	Temp Range (°C)	Program Memory	Data Memory	Operating Voltage (V)	Package	Price† ($)
MAXQ610B	0 to 70	64KB flash	2KB SRAM	1.7 to 3.6	40-TQFN	1.45

MAXQ is a registered trademark of Maxim Integrated Products, Inc.
SPI is a trademark of Motorola, Inc.
†1000-up recommended resale. Prices provided are for design guidance and are FOB USA. International prices will differ due to local duties, taxes, and exchange rates. Not all packages are offered in 1k increments, and some may require minimum order quantities.

www.maxim-ic.com/MAXQ610-info

MAXIM DIRECT
www.maxim-ic.com/shop

AVNET electronics marketing
www.em.avnet.com/maxim

MAXIM INNOVATION DELIVERED™

For free samples or technical support, visit our website.

evolved firmware architecture provides a new execution path when running test code. It also specifically defines how test code is added and used. Individual self-test code related to the module is placed inside the FW module itself. Such module-based self-test code is used to debug the individual FW module.

Likewise, code used to test build-up functions of FW modules is called *system-based self-test code*. This code is placed in its own individual source file SelfTest.c. If a bug is triggered at the system level, it means that all the FW modules work fine individually but manifest a bug when combined together. Then the system-based self-test code helps to figure it out.

As shown in Figure 2, using conditional compilation preprocessor directives enables source code to be compiled into different versions for both `RELEASE` and `SELF-TEST`. When `VERSION` equals `RELEASE`, all the self-test code is not compiled into the final hex file. Therefore, neither code size nor code speed is affected by self-test code.

For detailed information and examples, refer to the SelfTest.c implementation used in CyProject_Test.c and CyProject_Test.h., downloadable at *www.embedded.com/code.new/*.

FIRMWARE COMPATIBILITY

Often, it's difficult for the existing firmware code of a specific project to be reused in other projects. Designers can spend a great deal of time trying to apply existing code to a new project. Sometimes, efforts to reuse code can exceed the time required to simply rewrite new code. However, by considering firmware compatibility during firmware development and creating a unified interface, code reusability can be improved significantly.

Firmware compatibility is affected by the following factors:

Hardware

- *Hardware platform:* The target hardware platform may have different versions (such as different pin assignments) for different development stages or different customers. It's quite normal to support different hardware platforms with the same firmware
- *MCU:* MCUs in the same series or family are similar and compatible with each other to some extent. It's common to need to migrate firmware between MCUs in the same series
- *MCU configuration:* Some MCUs are highly configurable. For these processors, firmware should be updated to correspond with different MCU configurations

Integrated development environment

The firmware discussed in this article is the user-defined firmware that is only

EE Times
Virtual Conference

Upcoming Virtual Conferences

EE Times Virtual Conference:
Designing with ARM
Engineer an optimal ARM-based system

When: Thurs., March 25, 2010, 11am-6pm EDT
www.eetimes.com/arm

On Demand Virtual Conferences

Connected Devices:
Entering a new age of embedded design
www.eetimes.com/devices

Many-Core:
Designing with 100 cores
www.eetimes.com/manycore

LEDs and Lighting:
Designing for long life, low power
www.eetimes.com/LED

System-on-Chip:
Designing next generation SoCs
www.eetimes.com/SoC

Power Management:
Designing for efficiency
www.eetimes.com/power

EE Times, the leading resource for design decision makers in the electronics industry brings to you a series of Virtual Conferences. These fully interactive events incorporate online learning, active movement in and out of exhibit booths and sessions, vendor presentations and more. Because the conference is virtual you can experience it from the comfort of your own desk. So you can get right to the industry information and solutions you seek.

Why you should attend:

- Learn from top industry speakers
- Participate in educational sessions in real time
- Easy access to EE Times library of resources
- Interact with experts and vendors at the Virtual Expo Floor
- Find design solutions for your business

For sponsorship information, please contact:
David Blaza, 415-947-6929 or david.blaza@ubm.com

Firmware architecture with modularization, self-testability, and compatibility.

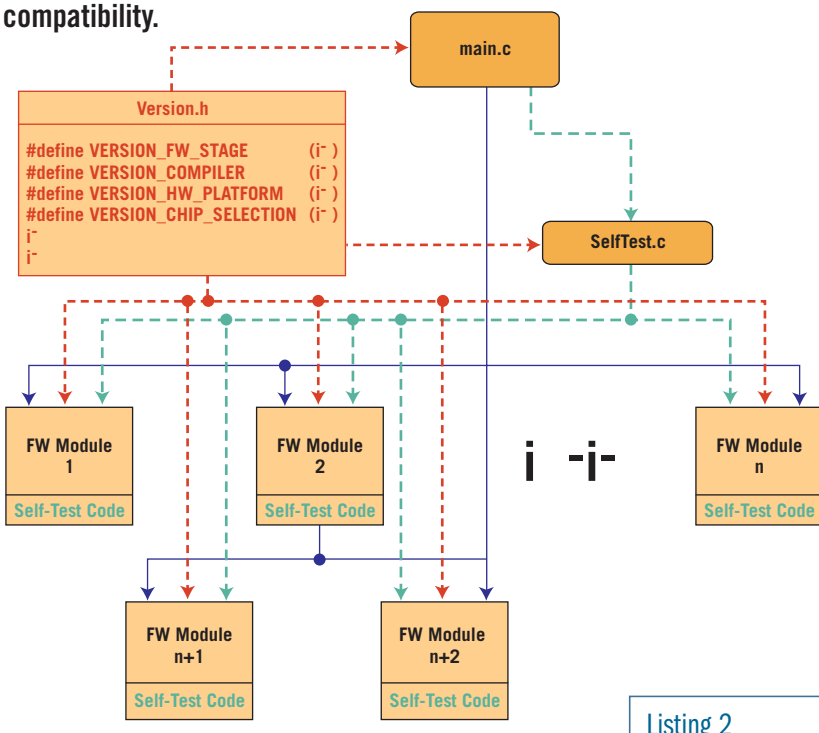


Figure 3

To switch firmware development stages requires only two simple steps. First, define the firmware development stage MACROs in Version.h, shown in **Listing 1**. Second, apply the firmware development stage MACROs in source code **Listing 2**.

Thus, firmware development stage switching is easily controlled using `VERSION_FW_STAGE`.

Designing with a flexible firmware architecture provides developers with many advantages. The ability to integrate all reusable code into an expandable and sharable firmware module library facilitates code reusability. The architecture also supports self-testability to accelerate code debugging and testing as well as provides a unified interface to manipulate firmware compatibility. Modularization also supports

Listing 1

```
#define   VERSION_RELEASE     (1 << 0)
#define   VERSION_SELFTEST    (1 << 1)
#define   VERSION_DEBUG       (1 << 2)
#define   VERSION_FW_STAGE    VERSION_SELFTEST
```

Listing 2

```
void main(void)
{
    #if (VERSION_FW_STAGE & VERSION_SELFTEST)
        SelfTest();
    #endif
    /* Insert your custom code below */
}
```

part of the whole firmware system. Almost all MCUs provide their own IDE which is responsible for managing the other components of the firmware architecture. For example, Cypress PSoC IDE (PSoC Designer) provides boot.asm, PSoCConfig.asm, and low-level drivers for the user modules. These automatically generated source files are the dependencies of user-defined firmware.

Firmware development stage

Firmware development may be divided into debug, self-test, and release stages. Different stages require different firmware behaviors.

Compiler

It's common that the same firmware code compiled with different compilers generates different results. Compilers may differ in compiler features that lead to different compiling results, including code size and code execution speed.

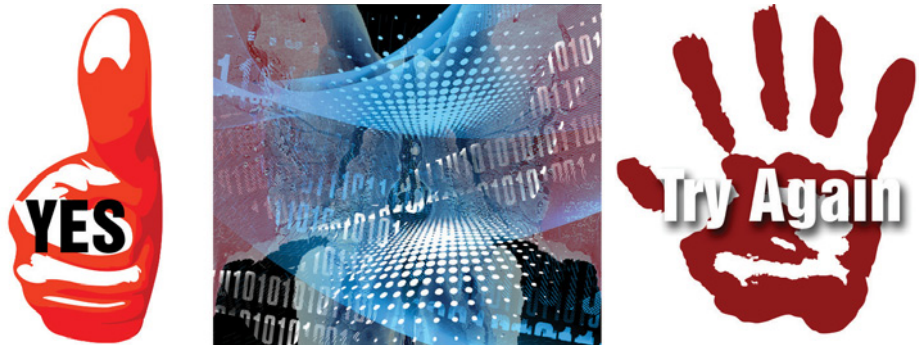
Given all of a system's firmware dependencies, firmware development can be simplified with a unified interface that allows developers to manipulate firmware dependencies and increase firmware compatibility as much as possible. Such an evolved firmware architecture is shown in the **Figure 3**.

Compared with Figure 2, this evolved firmware architecture introduces a new individual header file Version.h, which defines all firmware dependencies as MACROs. Version.h acts on all the source code (as the dotted lines show) to control firmware versions through C's conditional compilation directives (`#if`, `#else`, `#endif`, `#ifdef`, `#ifndef`, and so on).

customization of code while increasing readability, maintainability, and expandability.

While the concept of firmware modularization is one that can be applied to any microcontroller architecture or design, a firmware reference example description, along with the accompanying source files based on the use of the Cypress PSoC is available for download from *www.embedded.com/code.new/*. ■

Vincent Cai is an application engineer with Cypress Semiconductor and is responsible for the application of the PowerPSoC. He has a master's in electrical engineering from Xi'Dian University and has spent more than a decade in firmware development. Contact him at CaiWG.NiosII@gmail.com. His Chinese website is *NiosII.cublog.cn*.



The ease with which code is analyzable depends of the language type you use.

Expressive vs. permissive languages: Is that the question?

BY YANNICK MOY

Static analysis is becoming mainstream, with mature bug-finding tools for C and Java, including products such as Coverity Prevent, Grammatech CodeSonar, and Fortify SCA. These products limit the level of "noise" (false warnings) inherent to such tools to a minimum. However, by carefully selecting those cases for which they report a problem, these bug-finders hide the fact that they are largely uncertain about the overall correctness of the program.

To see this, it is usually sufficient to lower the ranking threshold below which problems are not reported, which immediately returns an extremely long list of possible problems, representing only a fraction of all the potential problems. By "problem" here, I mean any software bug, whether it's an error possibly detected at runtime (leading to an exception) or a mismatch between the programmer's intent and the obtained behavior.

As a developer and user of such tools, I have tried many times to answer a recurring question that most users have: *how can I make my program analyzable?* To tell the truth, beyond a few tips on features to avoid and idioms to prefer, usually not much can be done because so many decisions are outside the control of the programmer and instead depend upon the features of the programming language itself.

Therefore, the real question is: *which language should I choose to make my program analyzable?* This can be rephrased as: *what does a given programming language offer me "for free"?* By *free*, I mean information available from the source code without further analysis—information that, although not strictly required to express functional behavior, nonetheless captures part of the programmer's intention. Because software analysis is always

Tools for all languages strive to recover missing information, but as layers of missing info pile up, tools struggle harder to recover higher layers.

short of information about the programmer's intent, the information that comes "for free" in a programming language is the most valuable asset available to an analyzer. By this criterion, C ranks quite below Java, which ranks quite below Ada.

Of course, tools for all languages strive to recover missing information, but as layers of missing information pile up, tools struggle harder to recover higher layers. For example, tools that analyze programs in assembly language are largely limited to recovering only declared types and control-flow, whereas tools that analyze C programs more easily recover higher layers too, such as type-safety and memory-safety properties. Similarly, tools that analyze Java programs can often deduce integer ranges and non-nullity of pointers, in addition to the properties recovered from C programs. Tools that analyze Ada programs are best at also recovering "function contracts," the ultimate goal in software analysis, because so much of this free information is available.

A *function contract* is made up of a *precondition* that a caller of the function should satisfy prior to the call and a *postcondition* that the function should satisfy prior to returning. While user-defined contracts have appeared initially in some pioneering languages (especially Eiffel), they're now available in various widely recognized environments, for example in the .NET platform (Code Contracts) and the GNAT GCC compiler for Ada. Internally, many analyzers are based on generated contracts, also called *function summaries*, that they compute during the analysis.

I said contracts are the ultimate goal because complete contracts allow analyzing each function in isolation, much like function signatures allow separate compilation. However, only one programming language, SPARK, has gone as far as to require contracts from the programmer. So, in the general case, it's up to the analyzer to generate the contracts.

Now two aspects of a programming language define how much "free" information it provides: expressiveness and permissiveness. Often, these two di-

Listing 1 A function allocates an array of N structures, where N is the number of available processors defined in some configuration (in C, Java, and Ada).

In C:

```
Proc* alloc_processors (Config* conf) {
  Proc* res = (Proc*) malloc (conf->num_proc * sizeof(Proc));
  if (!res) exit (1);
  return res;
}
```

In Java:

```
Proc[] AllocProcessors (Config conf) {
  return new Proc[conf.num_proc];
}
```

In Ada:

```
type Proc_Number is range 1 .. 64;
type Proc_Array is array (Proc_Number range <>) of Proc;
type Proc_Array_Ptr is not null access Proc_Array;

procedure Alloc_Processors (Conf     : in Config;
                            Arr_Proc : out Proc_Array_Ptr) is
begin
   Arr_Proc := new Proc_Array (1 .. Conf.Num_Proc);
end Alloc_Processors;
```

mensions are confused, since they both seem to refer to the ability to express the programmer's intent in many different ways. But with analysis in mind, they have quite different meanings, one supporting analysis while the other undermines it.

A language is *expressive* when it allows a programmer to easily convey his/her intent, with errors detected early. In contrast to assembly languages, C allows one to constrain variables to hold values from specific types, so it is more expressive here. In contrast to C, Java allows to test if a value is within the length `a.length` of an array `a`, so it is more expressive here. Contrary to Java, Ada allows one to constrain scalar variables to hold values from specific ranges, so it is more expressive here.

A language is *permissive* when it allows constructs that interfere with a program's reliability or readability. This degree of permissiveness can be controlled through strong semantics and restrictions. In contrast to assembly languages, C provides structured control-flow statements (if-then-else, loops) and it prevents arbitrary jumps to and from functions, so it is less permissive here. In contrast to C, Java provides default-initialization to null for pointers and prevents conversions between pointers and integers, so it is less permissive here. In contrast to Java, Ada provides modes (in/out/in-out) for subprogram parameters and it detects integer overflows in computations, so it is less permissive here.

A SIMPLE EXAMPLE IN C/JAVA/ADA

To see how a more expressive and less permissive language helps in analyzing a piece of software, consider a very simple example in which a function allocates an array of *N* structures, where *N* is the number of available processors defined in some configuration.

Listing 1 illustrates straightforward implementations of this functionality in C, Java, and Ada, where I don't detail the types of structures for processors (`Proc`) and configurations (`Config`).

Although each of these implementations is correct when used appropriately, it's still possible that errors occur during the execution of each one, due to the failure of the caller to set up an appropriate environment.

Two aspects of a programming language define how much "free" information it provides: expressiveness and permissiveness.

In each case, there might not be enough memory left for the allocation to succeed. In C, `malloc` returns 0 and the function exits. In Java, an exception `OutOfMemoryError` is thrown. In Ada, an exception `Storage_Error` is raised. Notice that in Ada, the maximum number of processors (64) is precisely given by the type `Proc_Num-`

ber, while in C and Java the type of num_proc is a predefined integer type, whose maximal value might be huge.

In each case, the field num_proc may be uninitialized, which results in an arbitrary allocation size being picked.

In C and Java, conf might be a null pointer, in which case the access to its field num_proc fails. In C, a segmentation fault is usually reported. In Java, an exception NullPointerException is thrown. In Ada, the parameter is not passed as a reference at the level of the source text, although the compiler may decide to do so for efficiency, so the possibility of a null reference doesn't arise.

In C, the computation of the size to allocate in bytes may overflow, which results in a very small array being allocated instead of a very large one. This is a common source of security attacks, as an otherwise valid array access may then write beyond the bounds of the array. In Java and Ada, the runtime makes sure there is no overflow here.

Summarizing these findings, this makes a total of five possible errors in C, three in Java, and two in Ada. All this for a very simple function!

In C, conf might be an invalid pointer, either because it was never initialized or because the location it pointed to was freed or went out-of-scope. This usually results in hard-to-diagnose arbitrary behavior depending on the compiler and the program. In Java and Ada, language rules ensure that pointers are default-initialized to null.

Summarizing these findings, this makes a total of five possible errors in C, three in Java, and two in Ada. All this for a very simple function! Now replicate this effort for large functions with complex interactions, and you get C and Java programs that resist the analysis compared with the equivalent Ada programs.

Now, what else do these programs give for free for the analysis from the caller's point of view? In C, the type of the result of alloc_processors is a pointer, which may or may not be initialized, which may or may not be null, which may or may not be pointing to some valid memory. In Java, the type of the result of AllocProcessors is a pointer which may or may not be null. In Ada, the type of the result of Alloc_Processors is a non-null pointer, which may only be initialized to some non-null valid value. Also in Ada, the mode in for parameter Conf informs us that whatever is passed as Conf argument is not modified by calling Alloc_Processors.

This code snippet corresponds to real cases that I encountered repeatedly when searching for security integer overflow bugs in a large Microsoft code base in 2009, using an internal Microsoft analyzer called PREfix. Since this code base was in C, I usually had to rely on variable naming to decide whether or not a possible integer overflow detected by the tool introduced a security hole: while num_proc was not likely to be very large or controlled by an attacker, num_connections could legitimately be. Such reliance on naming schemes for detecting security breaches is unfortunately the best one can do in many cases with a language such as C.

INTEGERS AND POINTERS

The previous example made use of two critical types in programming lan-

guages: integers and pointers. It's not surprising that the degrees to which a programming language is expressive or permissive are for a large part tied to the way they treat integers and pointers.

Consider first integers. In most programming languages (C, C++, Java, C#), a direct mapping occurs between integral types defined in the language and machine integers. This is a source of tricky bit-vector manipulations, where either shifting is synonymous for dividing or a negative quantity might as well be positive (or both). In other languages, integers have the mighty power of mathematics, not being bounded by any sensible quantity, not even the number of atoms in the universe.

Both the machine view and the mathematical view lead to hard-to-analyze software. Integers in programs are quantities, neither bit-vectors nor abstract numbers. For the purpose of analysis, quantities should be appropriately bounded, which corresponds to having a valid range for those integral types, and letting users decide these bounds as they see fit.

This is the case in Ada. Being expressive, Ada lets the programmer define integer types and constraints. Not being permissive, Ada requires the use of compatible types in expressions, or else that appropriate type conversions are added.

Consider now pointers. In most programming languages, there is a unique kind of pointer that serves all purposes, leading to well-known programming hazards. In C, a pointer may refer to stack locations or heap locations, to a single reference or an array or even part of an array. In Java, a reference may refer to objects with very different lifetimes and it may be null.

In Ada the notions of references, arrays, access types, and addresses are carefully distinguished and may be further subdivided according to their use. A reference is of mode in when the corresponding procedure may only read from it; it is of mode out when the corresponding procedure may only write to it; it is of mode in-out when the corresponding procedure may read and write from it. An access type defines a storage pool in its scope, so that pointers of different access types cannot alias, even if they point to the same underlying type. An access type may be marked as non-null, in which case no value of such a type may be null.

One language, SPARK, is more expressive and less permissive than Ada. Not surprisingly, it consists in a subset of Ada together with contracts.

I could say much more about the degree of refinement with which Ada treats integers and pointers that makes this language both more expressive and less permissive, but the point should be clear already.

FUNCTION CONTRACTS

One language, SPARK, is more expressive and less permissive than Ada. Not surprisingly, it's partly built on Ada. SPARK consists in a subset of Ada, with stronger semantics and restrictions, together with contracts expressed as stylized Ada comments, so that a regular Ada compiler can compile SPARK code. While Ada was designed to facilitate program analysis, SPARK was designed to facilitate program verification.

SPARK contracts allow one to express data-flow (global annotation expressing reads and writes of global variables), information-flow (derives annotation), preconditions (pre annotation), and postconditions (post annotation), as the example code in **Listing 2** shows.

This contract specifies that procedure `Linear_Search` reads and writes the global variable `Counter`, that the value of each of its outputs (both out parameters and `Counter`) depend upon the value of all its inputs (both in parameters and `Counter`), that the procedure shall only be called when `Counter` is not the maximal integer value and that when the procedure re-

Listing 2 SPARK contracts allow one to express data-flow (global annotation expressing reads and writes of global variables), information-flow (derives annotation), preconditions (pre annotation), and postconditions (post annotation).

```
procedure Linear_Search
   (Table : in     IntArray;
    Value : in     Integer;
    Found :    out Boolean;
    Index :    out Integer);
--# global in out Counter;
--# derives Counter, Found, Index from Counter, Table, Value;
--# pre Counter < Integer'Last;
--# post Found -> (Table(Index) = Value and
--#                Counter = Counter~ + 1);
```

turns with `Found` being true, the `Value` searched is indeed at the output `Index` in the input `Table`, while the `Counter` has been incremented with respect to its initial value at procedure entry (denoted `Counter~`).

Analyzers for SPARK allow one to completely prove, statically, that such a contract is respected by the procedure's body, and that there are no runtime errors during its execution. For example, SPARK semantics and restrictions make sure that all reads of uninitialized values are caught by the analyzer, and dynamic allocations are not allowed, so that the errors on the Ada program `Alloc_Processors` are not possible in SPARK.

Because of these restrictions, SPARK may not be suitable in all contexts. Still, reliance on function contracts may be valuable for other programming languages:

- Contracts make it possible to analyze a library or a program that uses a library for which you do not have access to the source code;
- Contracts allow to split the analysis in many small independent ones, which will be collectively much cheaper to run, not mentioning the possibility of running them in parallel on a multicore multiprocessors machine;
- Since contracts allow the verification of individual functions, much more powerful techniques may be used, such as the SAT-based exploration performed in modern SMT-provers.

! Ada and SPARK are languages that are both more expressive and less permissive than C and Java, which greatly facilitates analysis.

In fact, various analyzers propose their own language extensions for C and Java to make up for what these languages are missing, for example, non-null annotations for Java pointers in Findbugs, in/out modes for C parameters in Splint, size annotations for C array parameters in PREfix, and so forth. These language extensions are mostly user-friendly ways of adding limited function contracts to the language.

As exemplified by SPARK, contract-based analysis is the most direct way to make software analysis as pervasive as compilation in some near future, in particular when associated to a programming language suitable for analysis, being both more expressive and less permissive.

IMPROVING CONFIDENCE

Applying static analysis to software is a sure way to improve on software correctness, as we do find and correct bugs along the way. When all bugs found have been corrected and the tool returns no more (true) warnings, it remains to be seen if that increases our confidence that the software is correct.

The answer depends more on the programming language chosen than on the tool used or the program analyzed. Indeed, programming languages are far apart when it comes to features that facilitate analysis. Ada and SPARK are languages that are both more expressive and less permissive than C and Java, which greatly facilitates analysis.

According to a study reported in 2003 by Andy German on military systems varying in size from 3,000 lines of code to 300,000 lines of code, these languages are also those in which programmers make less errors, four per thousand lines on average for SPARK, between 4.8 and 50 per thousand lines for Ada, between 12.5 and 500 (sic) per thousand lines for C.[1] For further reading, John Barnes' introduction to Ada 2005 gives much insight into why more expressive and less permissive programming languages lead to better software.[2] ■

Yannick Moy is a senior software engineer at AdaCore, where he is an expert in software source code analyzers that are used to detect defects or verify safety/security properties. Moy previously worked on source analyzers for PolySpace (now The MathWorks), INRIA Research Labs, Orange Labs, and Microsoft Research.

ENDNOTES:

1. German, Andy. "Software Static Code Analysis Lessons Learned," *CrossTalk*, November 2003.
2. Barnes, John. "Safe and Secure Software," 2008, *www.adacore.com/2008/04/08/gem-30/*.

For over 20 years, the Embedded Systems Conference has offered continuing education for embedded systems developers. Here are just a few of the ESC classes that Embedded.com Editor Bernard Cole finds intriguing for 2010.

Classes for embedded systems designers

BY BERNARD COLE

The best teachers of any subject are usually the professionals with long-term experience who can communicate what they've learned with insight and a healthy dose of humor. At the Embedded Systems Conference, these pros teach in over 20 carefully tailored design tracks.

Some of the speakers at ESC are regular columnists from this magazine, while others have contributed articles (and books) here and elsewhere in the field. Although the Internet makes their work more accessible than ever, listening to the presenter in the flesh gives students the advantage of face-to-face interactions: you'll learn faster and get questions answered more quickly (dare I say, improving your time to market?).

If not attending the conference, you can download some ESC classes from Embedded on Demand (*www.embedded-on-demand.com*), but many popular speakers, such as Jack Ganssle and Dan Saks, don't offer their classes online, and may never. Attending ESC is still the best way to access training from these popular teachers.

This year all the classes are "new," says the conference staff. All speakers have updated their content, even though a title and speaker may seem like an old favorite, it's in theory, a new and improved.

2010 EE Times ACE AWARDS

EE TIMES ACE AWARDS 2010

Tuesday, April 27, 2010
Fairmont, San Jose, CA

Congratulations to the 2010 EE Times ACE Awards Finalists

On April 27th EE Times will recognize creators of technology who demonstrate leadership and innovation in the global electronics industry and shape the world we live in.

2010 EE Times ACE Awards Finalists

Design Team of the Year

Advanced Micro Devices (AMD)

Android on MIPS Design Team

IDT Canada

Intel Corporation - Nehalem Family Design Team

Xilinx - Programmable Platforms Development Team

Executive of the Year

Doug Grose
Global Foundries

Sehat Sutardja
Marvell Semiconductors

Steve Sanghi
Microchip Technology, Inc.

Carlo Bozotti
STMicroelectronics

Ken Klein
Wind River

Innovator of the Year

eSilicon Corporation
Intersil Corporation
Numonyx
picoChip Designs Ltd
Rambus

Startup of the Year

Adesto Technologies
Energy Micro
Lunera Lighting
Open Kernel Labs
RFaxis, Inc

Company of the Year

ARM
Marvell Semiconductor, Inc.
Qualcomm
Silicon Labs
Texas Instruments

Most Promising New Technology

Hewlett Packard
Nujira
POWERVATION
Tabula
Verayo

Most Promising Renewable Energy Award

Cymbet
National Semiconductor
PlasmERG Inc
PV Powered, Inc.
Solar Roadways

Educator of the Year

Bassam Matar

Student of the Year

Kristie DAmbrosio

To purchase a ticket to the EE Times ACE Awards Cocktail Reception and Award Presentation go to ace.eetimeslive.com

We are proud to recognize the 2010 EE Times ACE Awards Lifetime Achievement Award Winner

Pasquale Pistorio
Honorary Chairman
STMicroelectronics

EE Times Group

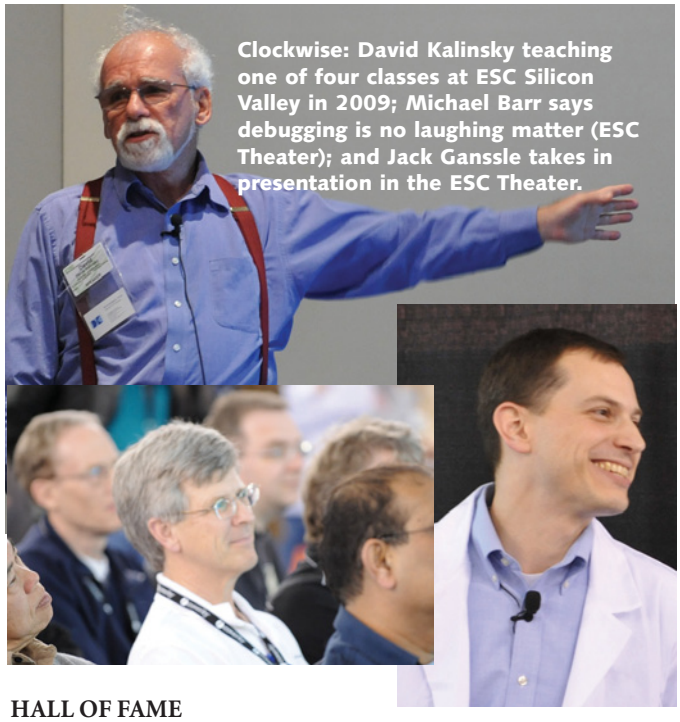
SILVER SPONSORS

AVNET technology solutions

XILINX

ASSOCIATION MEDIA SPONSOR

ieee spectrum INSIDE TECHNOLOGY



Clockwise: David Kalinsky teaching one of four classes at ESC Silicon Valley in 2009; Michael Barr says debugging is no laughing matter (ESC Theater); and Jack Ganssle takes in presentation in the ESC Theater.

HALL OF FAME

This year at ESC, look for **Jack Ganssle, Dan Saks**, and **Michael Barr**, all regular contributors to ESD; long-time ESC participants **David Kalinsky**, **Bill Gatliff**, **Christian Legare** and **Jean Labrosse** of Micrium, **David Kleidermacher** of Green Hills, **Dave Stewart** of InHand Electronics, **James Grenning** of Renaissance Computing (see Jack Ganssle's interview with Grenning on page 35), **Robert Oshana** of Freescale Semiconductor, Caltech's **David Hawkins**, and IBM's UML evangelist **Bruce Powel Douglass**.

A small sampling of the classes they'll be teaching include firmware flaws, coding standards, RTOS basics, adopting C++, managing firmware, structuring code for real time, product failures in the field, signal processing, software testing, and multicore debugging.

UP AND COMING

You can learn plenty from a wide range of speakers and topics, including:

"Zigbee Smart Energy Profiles (ESC-200)"—San Juan Software's **Drew Gialason** covers the basics of the various wireless Zigbee software stacks and Zigbee's Smart Energy Profiles, as well as designing energy-efficient home and building network systems.

"Agile Sensor Design for the Smart Grid (ESC-240)"—**Richard Newell**, senior principal product architect at Actel, describes the design of an intelligent sensor based on a state-of-the art FPGA with embedded microcontroller for use in smart energy grids.

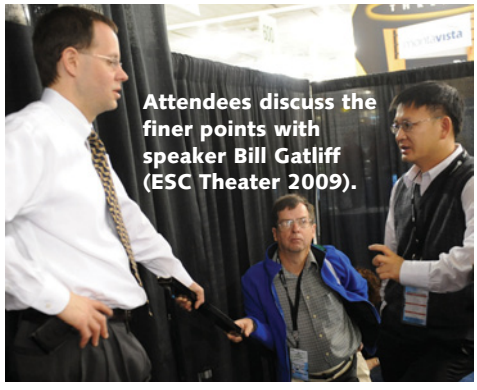
"Designing Power Efficient Motor Control (ESC-400)"—**Brad Landseadel**, president of Power and Control Design, addresses the need to drive control efficiency up to improve system-level efficiency while driving component count, costs, and complexity down.

TRACKS

- Aerospace and Military
- Build Your Own Embedded System 1 and 2
- Designing with Open-Source Software, including Linux and Android
- Developing for Windows Embedded
- Embedded Internet/Telecom
- Graphics, Displays, and Lighting
- Green Engineering
- Medical
- Microprocessors/Microcontrollers/DSPs
- Multicore Expo and Virtualization
- Multimedia and Signal Processing, including Consumer Electronics
- Networking and Connectivity
- Operating System Selections, Tips, and Tricks
- Programmable Logic
- Project Management
- Real-Time System Development
- Robotics, Motor Control, and More Industrial Change Makers
- Safety and Security
- Software Debugging Techniques
- Software Engineering
- System Integration and Test

A class at the Embedded Systems Conference in San Jose, 2009

All photos in this article were taken by Trish Tunney, *www.trishtunney.com*, 2009.



Attendees discuss the finer points with speaker Bill Gatliff (ESC Theater 2009).

Teardowns are a big draw in the ESC Theater where hands-on learning happens alongside hands-on giving of free stuff.

"Just Say No: Compile-time Enforcement of Hardware Semantics (ESC-202)"—**Stephen Dewhurst**, president, Semantics Consulting Inc. (not a new speaker to ESC), presents his new class on how to enforce extra-language and hardware-specific constraints at compile time, causing whole categories of common errors to simply disappear from your code.

"Building a Connected Device With Open Source Software (ESC-347)"—**Arlen Nipper**, president and CTO of Eurotech, Inc., describes how to apply open-source and Java-based middleware tools to virtually any networked and connected embedded device for fast development, quick time to market, and an extended product life span.

"Common Mistakes and Lessons Learned in Software Testing (ESC-267)"—**Mark Kraeling**, product manager, GE Transportation, will detail some common mistakes in embedded software testing, how to avoid these mistakes, and some real-world solutions to these issues.

"Fundamentals of Testing Embedded Systems (ESC-404)"—**Gina Bonini**, technical marketing manager, Tektronix, Inc., takes attendees through the latest debug techniques for complex embedded systems by performing a series of guided, hands-on exercises, using a mixed signal oscilloscope to validate and troubleshoot operation of a 16-bit parallel bus and I²C serial bus.

"7 Deadly Sins of Slow Software Builds (ESC-303),"—**Usman Muzaffer** and **Erin Curtis** of Electric Cloud delineate how to and how not to design build scripts and Makefiles for maximum parallel performance, covering what doesn't work, why, and how to make things better today without starting over (and without spending money).

"Best Practices of Top-Gun Engineering Managers (ESC-406)"—**Ronald Collett**, CEO, Numetrics Management Systems, presents methods and techniques that engineering managers use to differentiate their design team's execution-to-schedule excellence by calibrating to best-in-class industry competition.

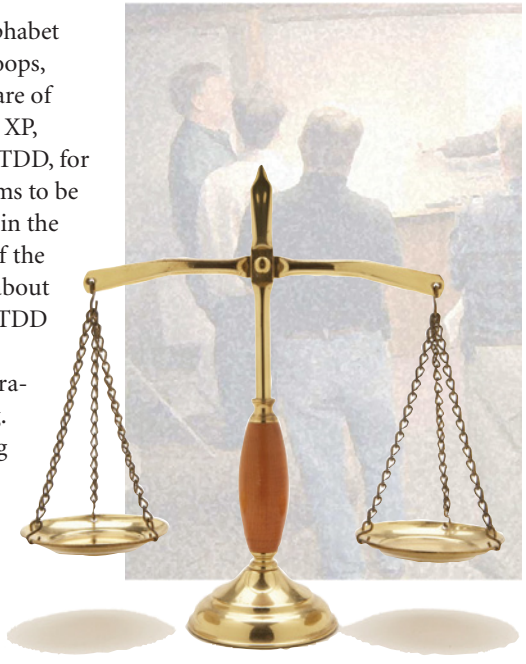
"Boost MCU Performance with an RTOS & Middleware (ESC-203)"—**Nilesh Rajbharti**, product manager at Renesas Technology America, teaches tricks of the trade he's learned to incorporate more functionality into a system within a shorter schedule and tight cost budget. ■

Teardown of flight data recorder, ESC SV 2009.

Bernard Cole is the site editor on Embedded.com and acquires the articles for Embedded Systems Design magazine and Embedded.com. He may be reached at bccole@acm.org

By Jack G. Ganssle

An interview with James Grenning



Jack Ganssle puts the Agile Manifesto's James Grenning on the hot seat about test-driven development's suitability for embedded systems.

The agile community's alphabet SOUP of acronyms (whoops, SOUP stands for "Software of Unknown Pedigree") include XP, TDD, FDD, and many more. TDD, for test-driven development, seems to be getting much more exposure in the embedded arena than most of the others. There's much to like about it, but I find some aspects of TDD unnerving.

TDD inverts the usual paradigm of software engineering. Its adherents advocate writing the tests first, followed by the application code that satisfies the tests. That's appealing; when testing is left to the end of the project, which is invariably running behind schedule, the validation gets dropped or shortchanged. Early testing also ensures the system is being built correctly: errors tend not to require massive restructuring of the system.

Further, TDD is implemented in very short cycles. Write a test, write some code, pass the test, and move on. As in eXtreme Programming the assumption is that code changes are low-risk since the tests are so comprehensive and fast to run.

The tests must run automatically; programmer intervention slows the process down. The suite returns a series of pass/fail indicators.

Kent Beck extolled the virtues of TDD in his 2003 tome *Test-Driven Development, by Example.*[1] I found the book shocking: the examples paint TDD as rampant hacking. Need a function that converts between dollars and Francs? Hard code the conversion rate. That will pass the tests. Then, over the course of far too many iterations, generalize the code to be the sort of stuff that, well, most of us would write from the outset.

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.

IS TDD JUST HACKING?

In my travels I often see companies use "agile" as a cover for a completely undisciplined development process. Agile methods, like all processes, fail in the absence of rigor. So, no, TDD is not about hacking.

But I take issue with some of the ideas behind TDD and think that it needs to be modified for firmware projects. James Grenning (*www.renaissancesoftware.net*), a signer of the Agile Manifesto and well-known speaker and trainer on agile methods, kindly agreed to be interviewed about TDD.[2, 3]

Jack: James, I have three reservations about TDD: in my view it deprecates the importance of design (and requirements gathering), and, though the focus on testing is fantastic, it seems test is used to the exclusion of everything else. Finally, I think TDD raises some important business concerns. Let's look at design first.

TDD newbies usually think that test-driven development is all about using tests to crank code, while the experts claim it's primarily about design. Yet the activities are all coding. How can TDD be about design?

James: Jack, thanks for the opportunity to have this dialog. I'll see if I can straighten you out on some of your misunderstandings.

I'll make several points about how TDD is really about design. Let's start with modularity and loose coupling: you cannot unit test a chunk of code if

A new era, a new ESC

New tracks. New courses. New focus.

ESC Chicago brings together systems architects, design engineers, suppliers, analysts, and media from across the globe. With cutting edge product demonstrations, visionary keynotes, and hundreds of essential training classes, ESC is the ideal conference for the embedded design community to learn, collaborate, and recognize excellence.

ESC Chicago 2010 Tracks Include:

- Designing for Embedded Linux (and Android)
- Medical
- Networking and Connectivity
- Open Source Software
- Project Management
- Project Management
- Real-Time System Development
- Robotics, Motor Control, and More Industrial Change Makers
- Safety and Security
- Software Debugging Techniques

Start your own personal development at ESC 2010. You can't afford to miss it.

Chicago
Donald E. Stevens Convention Center, Rosemont, IL
Conference: June 7-9, 2010
Expo: June 8-9, 2010

Register Today.
www.embedded.com/chicago

sensors expo & conference ESC is Co-locating with the Sensor Expo.
ESC attendees will have access to the Sensors expo floor.

Learn today. Design tomorrow.
ESC

EMBEDDED SYSTEMS DESIGN

ADVERTISING SALES

MEDIA KIT: ***www.embedded.com/mediakit***

EE Times Group

Sales Contacts

600 Harrison St., 5th Flr., San Francisco, CA 94107
600 Community Drive, Manhasset, NY 11030

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

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

Barbara Couchois
Vice President, Sales Operations
(415) 947-6928
barbara.couchois@ubm.com

Advertising Coordination and Production

600 Community Drive, Manhasset, NY 11030

Donna Ambrosino
Production Director
(516) 562-5115
dambrosi@ubm-us.com

it cannot be isolated and placed in a test harness. To do this you need modularity and loose coupling. TDD leads to it. Testability is confronted daily, keeping modularity and loose coupling (a.k.a, good design) throughout the life of the product.

Then there's doing appropriate up front work: Different situations and people require differing amounts of up front design. Before a development effort commences, a team should spend some time exploring design alternatives. This tends not to be formally documented in smaller teams. Also any documentation is usually kept high level, so that it is not mired in the detail that causes a lot of churn. I am not saying you cannot do documentation. I am just saying we don't have to do it up front while things are in flux. If formal documents are needed, we'd create them after getting the ideas to work.

An important part of being successful with TDD is having an architectural vision that guides the team in partitioning the software. The vision also evolves. Like many practicing TDD, I am schooled in the Robert Martin SOLID design principles.[4] Essentially, the SOLID principles provide guidance that help produce modular and loosely coupled code. In general, test-driven developers apply these principles to their design vision and daily work. One key idea is that modularity at a small scale supports good architecture.

The code is the design: An important paper, "What Is Software Design?" written by Jack W. Reeves in 1992, made a case that the code is the design, and it is a good case.[5] The code is analogous to the blueprints needed to manufacture a bridge or a car, while the makefile contains the assembly instructions. We need to give coding its due respect. The code specifies the behavior of the executable program in all its necessary detail. Take a look at Mr. Reeves' paper (*www.developerdotstar.com/mag/articles/PDF/DevDotStar_Reeves_CodeAsDesign.pdf*). It may change how you think about code.

We practice continuous design: TDD is a continuous design activity.

James Grenning

The code is analogous to the blueprints needed to manufacture a bridge or a car, while the makefile contains the assembly instructions. We need to give coding its due respect.

Design is never done; it is not a phase. As the design evolves, new modules are identified. The initial test cases for the new module are used to explore the interface of the module. The test is the module's first user. This really helps develop clean and easy to use interfaces.

During development, we might find that the design does not cleanly handle some new requirement. In TDD we have a complete set of automated tests. These tests form a safety net that takes a lot of the risk out of refactoring code. In the situation where there are no automated tests, the developer might shoe horn in the new functionality. It's just a small change after all, but it starts the code rot process that has led to many legacy code messes. How does a design rot? One line at a time.

Tests are a design spec: As TDD proceeds, the detailed design requirements are captured in the test cases. This is a very powerful form of documentation. It does not degrade into lies like many other forms of documentation do over time. Once developers become familiar with tests as documentation, they find if much more useful than prose. It also is the first place to look when considering using a module or understanding it so that a change can be made.

Are you aware of code rot radar? TDD acts as an early warning system for design problems. Imagine you can see in the production code where a change is needed for a new feature or bug fix. But you also find that you cannot devise a test that covers this change. This is an early warning of a design problem. The code is telling you, before it is too late, that code rot is starting. TDD gives warnings of functions that are too long, too deeply nested, or are taking on too many responsibilities. If you listen to what the code is saying, code rot can be prevented by making needed design improvements as they become necessary. Like I said a bit ago, the tests form a safety net to reduce the risk of restructuring working code. .

Jack: The agile community says we often don't know what we want, so we wind up building the wrong things. That observation drives a lot of the agile practices. My experience with embedded systems is that we generally have a reasonably clear idea of the end-product, though some features and UI aspects might be fuzzy at first. But most embedded apps have no real UI, or a UI that is fixed as it consists of some LEDs and switches or other clearly-defined I/O.

I have observed that a large part of the fuzziness stems from poor requirements elicitation. Often that's because doing requirements analysis is no fun. TDD seems to favor downplaying the requirements phase. Instead, why don't we devote more effort to it?

James: The situation you mention is well suited for incremental product development; some things are clear, some are fuzzy. Design is a continuous

process. Should requirements gathering and elaboration be continuous as well? When are the requirements really known? When are they done? The common answer I get to that question is "the day the product ships." Statements like that tell me that teams following a waterfall approach fall back to an evolutionary approach when things get tough.

Imagine if car manufactures welded the engine into the car, only running the engine after the weld cooled. We're doing that when we create code that can only be tested on the target.

In Agile, we consider that development engineering and requirements engineering proceed in parallel. Why delay development when there are important requirements, already known, that we could start developing? I don't know of a good reason for delay, but waiting has a definite downside. Serializing requirements and development adds unnecessary delay to overall product delivery timeline. You won't be able to get that time back that was wasted in delaying the start. Also, consider the time spent in a serial process, detailing out requirements only to cut them when you discover the date is in jeopardy. It's another waste you can avoid.

In agile, we strive to get good at evolving requirements and code concurrently. We want to get the product to market sooner. TDD says nothing about requirements collection. It is an incremental approach to development, which happens to work very well with incremental requirements gathering.

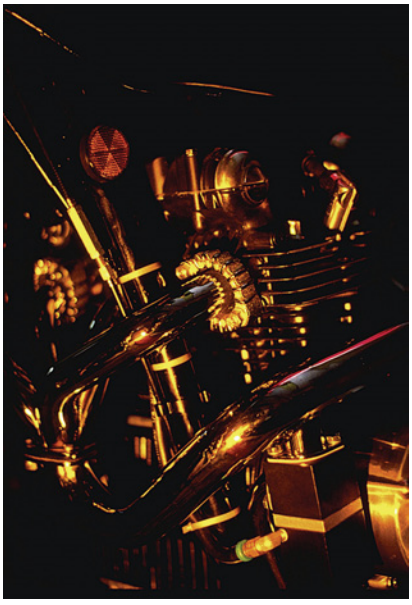
We will get some of the requirements wrong. This is the case for upfront requirements too. Development is invention; there is uncertainty. Building the product helps us figure out what we know and don't know, far better than more noodling over the illusive complete requirements.

One common concern I often hear is that some missing requirement will invalidate all our work. Sure it can happen if we started building a cell phone and in the end needed a washing machine. Requirements surprises are probably a greater risk for non-TDD projects. The TDD code base will be better able to handle a requirements mistake because of it resulting well-structured, modular code with automated tests.

One more thing on requirements, if you plan on staying in business, you better hope for new requirements; they are the opportunities that drive business. How many engineers out there are starting brand new products? I bet there are more of you working on ten-year-old code bases than the fresh greenfield product. Ask yourself, could you have defined the requirements that drive today's work ten years ago? Of course not. Today's requirements might not have even been known one year ago for some evolving products. TDD addresses the world that most developers find themselves in, a world where requirements and design must evolve.

Jack: Sure, but with embedded systems, we have to be pretty clear about requirements from the outset, as these impact the hardware design. That can't change much during the project since changes can drive the recurring manufacturing costs to the point where the product is no longer viable.

James: Are requirements really that clear? I bet some might argue with you. In the early stages, the hardware software boundary must be considered. Committing too early is a problem, too. Try to make boundary decisions reversible or finalize them when there is enough information. My main interest is the software side of this discussion, but there are people being more agile with hardware design. The X-15 rocket

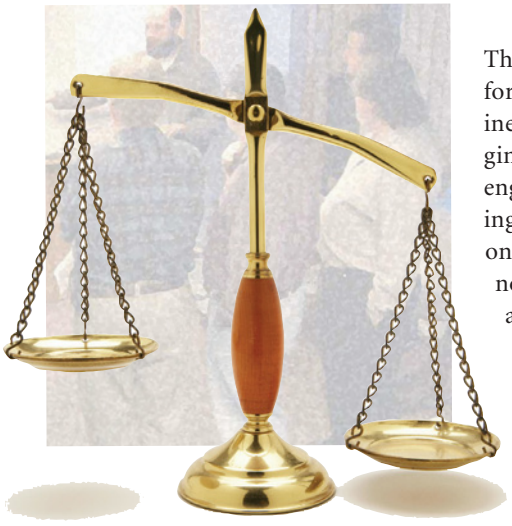


plane, completely a hardware project, was developed using incremental design techniques. There are some very agile hardware architectures evolving as well like the Cypress Programmable System on a Chip.

Jack: Let's talk about the heart of TDD: testing. TDD's automated testing is brilliant. I wish we all did more of it. But that runs into some problems in the embedded world. How do you automate tests when someone has to press buttons and watch displays?

James: At some level, a button press is resolved to a function call. A test case can make the same call as the button press event handler. On the output side, a simple test fixture can intercept messages destined for a display. I bet most if not all your readers have written a test main to exercise a new piece of code before integrating it. Likely that test fixture was discarded, once integration was complete. We do the same thing, but we write the test code so that it plugs into the test harness and continues to preserve the correct behavior of the code long after integration.

TDD requires learning new skills and honing existing skills. Creating automated tests and testable designs is a challenge. Dependencies on hardware



Testing at the product level should be more about making sure the pieces play well together, as well as some ad hoc exploratory testing.

and operating system must be managed. Functionality must be divided into modules, and those modules must be tested independently.

I never worked at a car manufacturer, but I bet that the engine goes through some test before being bolted into the car. It's modular; it's testable. The engine is run in a test fixture before bolting it to the car frame. Imagine if car manufactures welded the engine into the car, only running the engine after the weld cooled. We're doing that when we create code that can only be tested on the target. A weld is not a very flexible interface. I'm afraid I see too much of that kind of embedded software.

It's interesting; this is the same kind if objection some desktop and business application software developers express. The answer turns out be the same in principle: build modular code, testable code.

Jack: I have seen some way cool stuff being done with LabVIEW's vision module; people use a TV camera to watch the embedded device and see that it's doing the right things. But if we're doing tests on a mock, then we're really not testing the system; we're testing some simulation of it that may not have perfect fidelity. And that suggests that the very hardest tests—integration—are left to the end, or are not part of the central thesis of TDD, which is continuous automated testing.

James: The LabVIEW approach sounds pretty cool. Do they have robot arms pushing buttons too? Teams should definitely invest in automated system tests. Testing as a product is necessary but not adequate; the pieces must be tested too. If the pieces are not solid, the system won't be solid. In complex systems, it is numerically impossible to test all the code paths and error cases when exercised as a fully integrated product. Now, at the unit test level, it is possible to exercise all paths. If we only test at the system level, many code paths will go untried until the product bumps into that one in a million event while in the hands of the customer. I think you have a story like this while you were out on the Atlantic with no land in sight.

Testing at the product level should be more about making sure the pieces play well together, as well as some ad hoc exploratory testing. The full 100% coverage ideal is only practical at the unit level. If you had two interacting components that each required 10 tests, they would require as many as 100 tests if they were integrated. That's not practical, but 20 tests and a handful of integration tests is.

Jack: Thanks, James. I know you've got more to say, so we'll carry this discussion on to next month's issue of *Embedded Systems Design*. ■

IS AGILE GOOD FOR EMBEDDED SYSTEMS DESIGN?

James Grenning's short, humorous video from ESC Boston shows all the reasons agile won't work for embedded software development.

www.renaissancesoftware.net/component/content/article/17/76-agile-for-embedded-impossible.html

ENDNOTES:

1. Beck, Kent. T*est-Driven Development, by Example*. (Pearson Education, Boston, MA, 2003.
2. James Grenning's bio and class list for ESC Silicon Valley 2010, www.cmpevents.com/ESCw10/a.asp?option=G&V=3&id=249428
3. Agile Manifesto, http://agilemanifesto.org/
4. Robert Martin (calling himself Uncle Bob) talks about his design principles here *http://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start*, with a link to his article "Principles and Patterns" written in 2000 (*www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf*)
5. Reeves, Jack W. "What Is Software Design?" *developer*. magazine*, 1992. *www.developerdotstar.com/mag/articles/PDF/DevDotStar_Reeves_CodeAsDesign.pdf*


THE EMBEDDED
DEVELOPER
Microsoft
Windows Embedded
BUILT WITH WINDOWS 7 TECHNOLOGIES
Vol. 5
A MALFUNCTION IN THE SYSTEM COULD COST THE PLANT MILLIONS...
THE DEVICE HAS TO...
WORK PERFECTLY, TO THE MICROSECOND, AND HAVE THE CONNECTIVITY TO TRACK PERFORMANCE IN REAL TIME.
THEY'RE COUNTING ON ME TO DELIVER.
WINDOWS® EMBEDDED OFFERS A HIGHLY RELIABLE PLATFORM, WITH THE LEVEL OF PERFORMANCE YOU NEED TO HELP DELIVER CONNECTED DEVICES THAT STAND OUT.
WHICH WINDOWS® EMBEDDED-PLATFORM CAN HELP YOU DELIVER STANDOUT DEVICES?
FIND OUT AT WINDOWSEMBEDDED.COM/DEVICESTORIES

R&D Prototype

PCB Assembly

$50 in 3-Days

Advanced Assembly specializes in fast assembly for R&D prototypes, NPI, and low-volume orders. We machine-place all SMT parts and carefully follow each board through the entire process to deliver accurately assembled boards in three days or less.

R&D Assembly Pricing Matrix/Free tooling and programming

Up to # SMT Parts	25	50	100	150	200	250	300	Over 300
1st board	$50	$85	$105	$155	$205	$255	$305	
2nd board	$30	$55	$65	$95	$125	$165	$185	Call for Pricing
Each additional board	$25	$35	$45	$65	$95	$125	$155	
Stencil	$50	$50	$50	$50	$50	$50	$50	

aapcb.com/esd3
1.800.838.5650

Advanced Assembly®
SMT ASSEMBLY FOR ENGINEERS™

Lead Free
SO 9001:2008 Certified

The new standard for pcb assembly