United Business Media

Dr.Dobb's DIGEST

The Art and Business of Software Development

August 2010

 Published by Techweb, United Business Media Limited, 600 Harrison St., San Francisco, CA 94107 USA 415-947-6000.

The "Visual Programming" Silver Bullet

By Jonathan Erickson,
Editor In Chief

App Inventor (http://appinventor.googlelabs.com/about/) is a tool from Google Labs that is intended to make it easy for nonprogrammers to create mobile applications for Android-powered devices. What's missing in this is the word "good", as in " make it easy for non-programmers to create good mobile applications."

Not that you need a degree in computer science or mathematics to create mobile apps, but it does help at times. Far too often, as reader James Kosin correctly points out, "easy" translates too easily to sloppy and bloated, as in code I write. But then high-level tools like App Inventor are supposed to prevent that as well.

Instead of writing code with App Inventor, you visually design the way the app looks and use "blocks" to specify the app's behavior. Blocks are available for storing information, repeating actions, and performing actions under certain conditions. Even blocks to talk to services like Twitter.

The concept of blocks that encapsulate functionality isn't anything new. LEGOs MindStorm, for instance, implements similar functionality but in "bricks." Similarly, Apple Computer, led by Smalltalk guru Dan Ingalls, developed Fabrik (http://wiki.squeak.org/squeak/1227) — "Fa-brick," get it? — a kit of computational and UI components that you "wire" together to build new components and applications.

And when your talking about visual programming with blocks, you should, as Ron Martin reminded me, mention what is probably the most mature such system — LabVIE (http://www.ni.com/labview/)W.

The blocks editor that App Inventor uses is the Open Blocks Java library (http://dspace.mit.edu/handle/1721.1/41550) for creating visual block programming languages. Open Blocks is distributed by the MIT and derives from thesis research by Ricarose Roque. OpenBlocks lets developers build their own graphical block programming systems by specifying a single XML file. Open Blocks visual programming is closely related to the Scratch programming language (http://www.drdobbs.com/tools/209600388;jsessionid=ZZCAWU2A0AFYFQE1GHPSKH4ATMY32JVN), another MIT Media Lab project.

The compiler that translates the visual blocks language for implementation on Android uses the Kawa Language Framework (http://www.gnu.org/software/kawa/), a framework written in Java for implementing high-level and dynamic languages and compiling them into Java bytecodes, and Kawa's dialect of the Scheme programming language, developed by Per Bothner and distributed as part of the Gnu Operating System by the Free Software Foundation.

When it comes to software development, there's something gripping about the word "visual". It's routinely used in instances that are not visual in any sense of the word, although some uses are more visual than others. One of the true visual programming languages is Cube (http://www.drdobbs.com/architecture-and-design/184409677;jsessionid=ZZCAWU2A0AFYFQE1GHPSKH4ATMY32JVN) developed by Marc Najork. As Marc has pointed out, "a true visual-programming language can be considered 'executable graphics,' with no hidden text." So is App Inventor a true visual language that will enable programming for non-programmers? That remainds to be seen. What do you think?

To get started with App Inventor development, fill out the form at https://services.google.com/fb/forms/appinventorinterest/. More information is available at http://appinventor.googlelabs.com/learn/.

Return to Table of Contents

Computer Gamers Tackle Protein Folding

The challenge in developing the game was to make it fun while still producing valid scientific results

By Hannah Hickey

Biochemists and computer scientists at the University of Washington two years ago launched an ambitious project harnessing the brainpower of computer gamers to solve medical problems. The game, Foldit (http://fold.it/portal/), turns one of the hardest problems in molecular biology into a game a bit reminiscent of Tetris. Thousands of people have now played a game that asks them to fold a protein rather than stack colored blocks or rescue a princess.

Results reports show that Foldit is a success. It turns out that people can, indeed, compete with supercomputers in this arena. Analysis shows that players bested the computers on problems that required radical moves, risks, and long-term vision — the kinds of qualities that computers do not possess.

"People in the scientific community have known about Foldit for a while, and everybody thought it was a great idea, but the really fundamental question in most scientists' minds was 'What can it produce in terms of results? Is there any evidence that it's doing something useful?'"said principal investigator Zoran Popovic (http://www.cs.washington.edu/homes/zoran/), a UW associate professor of computer science and engineering.

Scientists know the pieces that make up a protein but cannot predict how those parts fit together into a 3D structure. And since proteins act like locks and keys, the structure is crucial.

At any moment, thousands of computers are working away at calculating how physical forces would cause a protein to fold. But no computer in the world is big enough, and computers may not take the smartest approach. So the UW team tried to make it into a game that people could play and compete. Foldit turns protein-folding into a game and awards points based on the internal energy of the 3D protein structure, dictated by the laws of physics.

Tens of thousands of players have taken the challenge. The author list for the paper includes an acknowledgment of more than 57,000 Foldit players, which may be unprecedented on a scientific publication.

A major challenge in developing the game was to make it fun while still producing valid scientific results. There was a constant back-and-forth between scientists, game developers, and players to achieve the best balance.

The class of problems in which humans were able to do better than computers required intuitive leaps or major shifts in strategy. Future work will aim to better combine the strengths of experts, computers, and thousands of game players.

"It's a new kind of collective intelligence, as opposed to individual intelligence, that we want to study," Popovic said. "We're opening eyes in terms of how people think about human intelligence and group intelligence, and what the possibilities are when you get huge numbers of people together to solve a very hard problem."

The Foldit energy calculations are carried out by Rosetta, the procedure for computing protein structures developed by coauthor David Baker, a UW biochemistry professor. Baker's group has previously used donated computer cycles through Rosetta@home to help crunch through the trillions

Dr.Dobb's

EDITOR-IN-CHIEF
Jonathan Erickson

EDITORIAL
MANAGING EDITOR
Deirdre Blake
COPY EDITOR
Amy Stephens
CONTRIBUTING EDITORS
Mike Riley, Herb Sutter
WEBMASTER
Sean Coady

VICE PRESIDENT, GROUP PUBLISHER
Brandon Friesen
VICE PRESIDENT GROUP SALES
Martha Schwartz

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

DR. DOBB'S
600 Harrison Street, 6th Floor, San Francisco, CA, 94107. 415-947-6000.
www.drdobbs.com

UBM LLC

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

TechWeb

Tony L. Uphoff Chief Executive Officer
John Dennehy, CFO
David Michael, CIO
John Siefert, Senior Vice President and Publisher, InformationWeek Business Technology Network
Bob Evans Senior Vice President and Content Director, InformationWeek Global CIO
Joseph Braue Senior Vice President, Light Reading Communications Network
Scott Vaughan Vice President, Marketing Services
John Ecke Vice President, Financial Technology Network
Beth Rivera Vice President, Human Resources
Fritz Nelson Executive Producer, TechWeb TV

techweb™

of possible orientations for the chains of amino acid molecules that make up proteins.

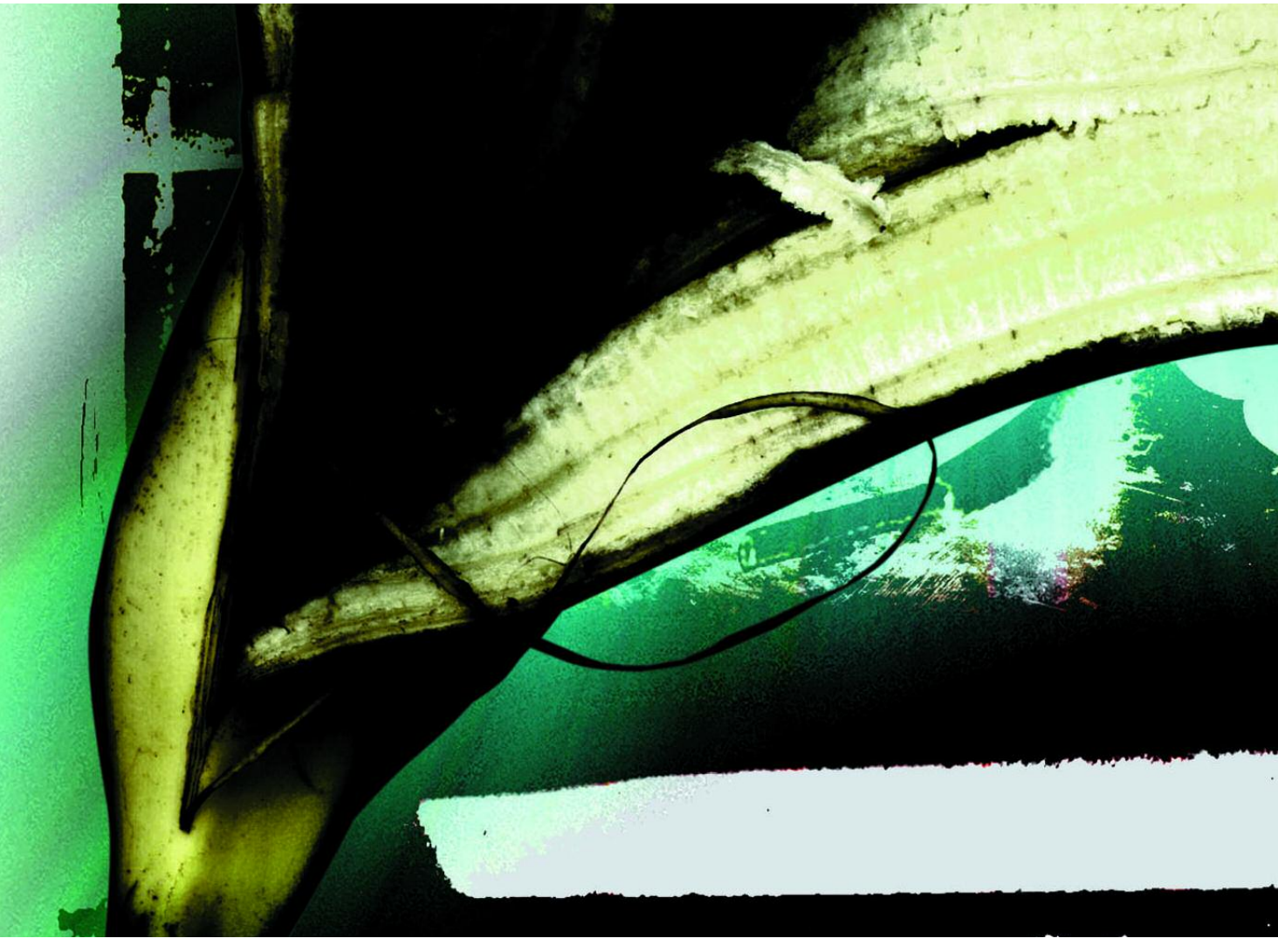
The human thinking patterns may now help bolster Rosetta's skills. Researchers in Baker's group are analyzing the most successful Foldit strategies and trying to replicate them in the computer-powered version.

This summer, the Foldit community has been focused on problems in the Critical Assessment of Techniques for Protein Structure Prediction competition (http://prediction-center.org/casp9/index.cgi), the world's largest comparison of protein-folding computation strategies. Last year Foldit competed as part of the Baker lab team. This year for the first time Foldit players have their own team, taking on the most sophisticated supercomputers in the world. Contest results will be announced in December.

Now, Foldit players will focus on designing novel proteins. Last year a Texas player who goes by the name "BootsMcGraw" was the first Foldit player to have his new protein design synthesized in the Baker lab. Although this particular structure did not work, the researchers plan to try again and are optimistic about the possibilities.

"I think that design problems are an area where human computing has huge potential," Baker said. "People are good at building things, so I'm expecting that people will be very good at building proteins for different purposes. That's where I'm expecting really great things from Foldit."

Return to Table of Contents



The Requirements Payoff

Getting a project's requirements right from the start can speed development and ward off problems

By Karl Wiegers

Investing the time to create better requirements for a software project takes a major leap of faith, since people tend to think the extra up-front work will just delay development. That's generally true, but getting requirements right also prevents problems later that can not only delay projects but lead to their failure.

Incomplete requirements and specifications, as well as changing requirements, are a main cause of project distress. For example, the FBI abandoned its Virtual Case File case management software project in 2005, along with $170 million and 700,000 lines of code, due in large part to poorly defined and slowly evolving design requirements, according to a scathing report by Department of Justice's Inspector General Glenn Fine. "It was a classic case of not getting the requirements sufficiently defined ... from the beginning," Fine says in the report. "And so it required a continuous redefinition of requirements that had a cascading effect on what had already been designed and produced."

An error, omission, or misunderstanding in the requirements forces developers to redo work they've done based on the incorrect requirement. Thirty percent or more of the total effort spent on most projects goes into rework, and requirement errors can consume 70% to 85% of all project rework costs, according to a 1997 study by Dean Leffingwell. Development teams often implement functionality that someone swore they needed, only to find that no one ever uses it. Techniques that can prevent this wasted effort are a solid investment.

What's To Be Gained

Good preliminary requirements help CIOs make effective business decisions regarding which projects to fund. Well-defined and documented final requirements are critical to letting developers devise the most appropriate approach, estimate the effort needed to execute an iteration cycle or complete the project, incorporate changes that will deliver maximum customer value, and develop accurate test cases to verify the implemented functionality.

Putting more effort into requirements development can actually accelerate software development. At a large insurance company I worked with, increasing the front-end requirements effort from 19% to 33.6% of total effort reduced projects' overall effort and cost by an average of 4%. Another company trained its development teams on requirements engineering and implemented improved requirements practices. In a year, the share of projects that were behind schedule dropped from 21% to 12%, and the total days those projects were behind dropped from 1,738 to 518.

While no one can predict what ROI you're going to get from your investment in better requirements, consider these questions as a first step in determining payback:

- What fraction of your development effort do you expend on rework? Some rework is unavoidable and adds value, but a lot of rework is nothing but wasted effort.
- How much does a typical customer-reported defect cost your organization? A system-test defect? One of my clients spent an average of $4,200 to deal with each customer-reported defect. That was 21 times more than it cost them to discover a defect through formal inspection.
- What fraction of user-reported defects and what fraction of defects discovered through system testing stem from errors in requirements? Defect root-cause analysis is an excellent technique to gauge how much you could gain from improving quality.

- How much maintenance cost, from defect correction and unplanned enhancements, can you attribute to requirement defects such as missed requirements?
- How much could you shorten your delivery schedules if your project teams could reduce requirement defects by, say, 50%?

Practices that result in fewer requirement defects will reduce the amount of development rework your teams must perform. This provides immediate payback through reduced development costs and quicker time to market. Techniques that get your analysts and customers working closer together also lead to products that better meet customer needs and require less reworking. All of these approaches have the potential to increase customer satisfaction.

You can begin "testing" your software as soon as you've written the first requirement

Steps To Improve Requirements

In the quest for good requirements, first make sure you have appropriately skilled and trained business analysts who can guide the requirements development and management activities on each project.

As your teams get up to speed and requirements become more sophisticated, having the right tools can be a big help. Lots of requirements management tools are available, including IBM Rational RequisitePro, Micro Focus CaliberRM, Microsoft Visual Studio Team System 2010, MKS Integrity, and Ravenflow Raven 2010.

With new projects, I begin at the top by clearly establishing the business objectives. Defining the product vision and project scope help ensure that all the work done aligns with achieving those objectives. This includes assessing your current practices and identifying areas that aren't delivering desired results. Improvement might require writing new processes, modifying current processes, and selecting new templates for key requirements deliverables.

Next, identify distinct communities or classes of users and determine who will serve as the voice of the customer for each such user class. To engage appropriate customer representatives, I recommend designating "product champions" who are key customer representatives engaged with the project on an ongoing basis. This is much like the Agile development concept of the on-site customer.

The "use case" technique, which focuses on expected usage rather than on product features, is an excellent way to explore user product requirements. It should be clear how the use cases will align with business objectives. If they don't align, there's a problem. But use cases aren't sufficient in every situation. Your analysts also need to derive the functional requirements that developers will implement to let users perform the use cases.

In addition, explore and document nonfunctional, quality-related requirements such as usability, security, reliability, and robustness. These attributes are vital to customer satisfaction. Ongoing prioritization of the requirements also is crucial. Think of the backlog of pending requirements as a dynamic list, not a static snapshot frozen in time.

Don't just assume the requirements are correct: validate and verify them. You can begin "testing" your software as soon as you've written the first requirement. On one project I was involved in, I wrote a dozen or so functional requirements, then another dozen or so test cases based on my vision of how the code would operate. In writing the test cases, I discovered an error in one of my requirements. I generally find these sorts of errors, omissions, and ambiguities in my requirements at this point.

A well-structured and rigorous peer review or inspection of requirements documentation is also a good investment. And finally, an effective change management process will help ensure that your project delivers the product that customers need.

Of course, the greatest investment you can make is the time you spend eliciting, analyzing, specifying, validating, and managing requirements. Time spent on these efforts is likely to accelerate a project and make it run more smoothly.

These practices aren't free, but they're cheaper than waiting until the end of a project or iteration, and then fixing all the problems. The case for solid requirements practices is an economic one. With requirements, it's not "You can pay me now, or you can pay me later." Instead, it's "You can pay me now, or you can pay me a whole lot more later."

— Karl Wiegers is principal consultant with Process Impact, where he focuses on practical software process improvement.

Return to Table of Contents

Detecting Endian Issues with Static Analysis Tools

Best practices for using static analysis tools and enforcing correct programming in C/C++

By Carl Ek

"There are 0010 0000 kinds of people in the world: Those that understand the difference between Big Endian and Little Endian, and those that do not." Since all binary processors (hardware or software) have an endian design, correct processing of the data based on that endian design is extremely important. The statement above is a version of another joke, but with a twist: The binary is represented in little endian, giving some mild humor for those that understand. For those that don't understand endianness, the humor is lost, much like a processor that has an endian processing defect. In this article, I describe the kinds of defects that occur, and methods where static analysis tools can help detect programming errors and enforce correct programming. But first, let's define some terms:

- Endianness. The nature of byte layout in storage of multi-byte datatypes
- Big Endian. Byte store in most significant order across ascending address
- Little Endian. Byte store in most significant order across descending address

For example, the two-byte integer 54,321 = 0xD431

- Big Endian: stored as D4 31
- Little Endian: stored as 31 D4

It is easy to see that if a little endian processor stored the integer value 54,321 and subsequently processed in the incorrect ascending address order, the value would be incorrectly derived as 0x31D4 or 12,756.

Hence, endianness issues in programming are twofold:

1. When endianness is not mixed, care must be taken if programs are to be portable across both little and big endian processors: this is an "Endian Neutral" strategy.
2. When endianness in programs is mixed, datatypes must be processed with their corresponding correct byte order: this is an "Endian Protocol" strategy.

Endianness and the C Programming Language

The C language gives the programmer many ways to shoot one's self in the foot. Since the language gives access to the bits and bytes of storage through pointers and other mechanisms, data access is dependent upon the storage layout of the bytes.

The two strategies (a) and (b) above can both be handled with carefully coded algorithms. In the case of (a), some general rules for C/C++, if diligently followed, can avoid algorithmic difficulties to avoid endian-related problems. Here is a (non-exhaustive) list:

- Avoid using unions that combine different multi-byte datatypes — the layout of the unions may have different endian-related orders.
- Avoid accessing byte arrays outside of the byte datatype — the order of the byte array has an endian-related order.

- Avoid using bit-fields and byte-masks — since the layout of the storage is dependent upon endianness, the masking of the bytes and selection of the bit fields is endian sensitive.
- Avoid casting pointers from multi-byte type to other byte types — when a pointer is cast from one type to another, the endianness of the source (ie. the original target) is lost and subsequent processing may be incorrect.

All rules are similar in concept: Avoid processing bytes in an order with assumptions about their storage layout. More rules for C and C-style languages could be derived from the above, and rules could be developed for other languages. Enforcing rules such as the above will result in more lines of code that are portable across different endian systems. Good implementations of this strategy could be run on one endian system, and could be ported with minimal changes to another endian system.

What is the best practice when the endianness of datatypes in a single application is mixed? In this case the programming must properly process both Little Endian and Big Endian data, and an Endian Protocol strategy is needed. The fundamental operation in these algorithms require methods to swap bytes as data is processed. This can be done in hardware or in software. However, hardware solutions are not practical in many cases due to the variability of configurations and data to be handled. Software byte swapping methods have been developed where the endianness is defined or derived and the data bytes are processed in the correct context based on their endianness.

In the C language, numerous bytes swap facilities have been designed for this:

1. The *swab* function: to swap two adjacent bytes
2. Byte swapping macros:
    - *ntohs*: network to host short
    - *ntohl*: network to host long
    - etc.
3. Inline *bswap* macros:
4. CFSwapInt16BigToHost, ... :
5. and more

Static Analysis Tools Features: Built-In Detection and Extendable Detection

Some endian issues can be found with a static analysis tools' built-in features. These may or may not be flagged as specific to endianness, but due to a side-effect of another issue: For example, in the MISRA standards (http://www.misra-c.com/) there are numerous rules that pertain to issues using multi-byte storage across different datatypes. These are not specifically intended to enforce errors in endian processing but may indeed, as the general pattern they are detecting is common.

If the static analysis tool has extendable features, there can be many more custom issues detected. Key requirements for developing extensions are:

1. Define syntactical specifications identifying error case(s).
2. Define syntactical specifications identifying passing case(s).
3. For both 1 and 2, if possible, define any flow-sensitive issues to consider.

For best practices on the above, the error case must always give some possible hint to the solution. And if the defect is identified with a path which can be traced with the static analysis tool, fewer false positives for the error case can be the result.

Developing custom static analysis tools to detect issues requires that two main questions be answered first:

1. What is your strategy for coding:
   "Endian-neutral?" "Endian-protocols?" or a blend?
2. What coding issues are you to enforce:
   "What are the violations?" "What are the correct cases?"

Once those questions are answered, in particular question 2, static analysis rules can be developed to detect issues and enforce correctness.

Detecting Errors Related to Endianness: Syntax and Semantics

Outside of exhaustive unit testing and code inspection, it would be nice if compilers could tell programmers when endian issues are present. But any warnings that a compiler could give on any potential problems would be so noisy that they would most certainly be ignored.

All detection methods require some knowledge of the endianness of the data being processed before any warning can be given. Since compilers are not "smart enough" to know the contents of variables at run time, their ability to detect endian-related errors is limited. Detecting endian issues at a code inspection is easier if naming conventions are used to identify Big Endian or Little Endian data, but compilers are (usually) not privy to the knowledge of such coding standards.

This is where static analysis tools can be used to detect issues that are applicable to a specific environment. Static analysis tools do not have to behave like compilers, which must conform to a language standard; they can report on issues that violate such specific things as potential endian errors.

As a simple example, it is possible to develop a static analysis rule to warn on this situation:

```
union {
          short number;
          char  view[2];
} my_number;
```

But if the appropriate coding standard was in place, it could be tailored to give no warning for this: (if the tool knew that *regex(BigEndian.*)* names are Big Endian):

```
union {
          short number;
          char  BigEndian_view[2];
} my_number;
```

Detecting Errors Related to Endianness: Protocol and Paths

The aforementioned example with a union is a case of detecting errors by examining syntax and semantics. There are other errors that can be detected when programming protocols are used to correctly handle endianness processing.

In these schemes, proper byte swapping is necessary, and protocols must be followed in order that numeric values are correctly interpreted.

A simple, but typical, example of a protocol: reading from and writing to a network. In this example, the length values must be passed into the function-style macros *ntohl()* and *htonl()* before they are used in the calls to *data_alloc()* and *net_write*(). Proper byte-swapping is done for input and output information:

```
n = net_read (&sock, &netLen, 4);
/* ... process errors ... */
len = ntohl(netLen);.              /* A */
data_alloc (&packet, len);        /* B */
          ...
len = strlen(ret_string) + 1;
netLen = htonl(len);               /* C */
if (net_write (context, &sock, &netLen, 4) != 4) /* D */
/* ... process errors ... */
```

A static analysis tool detector could be developed to keep track of specific functions whose parameters must be processed by the byte-swapping routines. If the byte-swapping routine has not been called for each parameter (or the incorrect one has been called!) an error or warning message could be issued.

The types and complexities of the protocols that can be implemented depend a great deal upon the customization features and abilities of the particular static analysis tool, the usability, and other restrictions common across all static analysis tools.

Simple Rules Detecting Complex Problems

These examples show very simple coding for illustrative purposes. But these simple rules are important to consider: They can be defined with clarity and implemented with robustness. As well, if implemented in static analysis tools, they can detect subtle defects that exist inside complex code. As datatypes are defined with higher complexity, the syntax and semantic guidelines can be violated in very obscure ways, resulting in defects. Likewise, as coding is modified over time, protocol violations can be mismanaged in non-obvious manners, where incorrect execution paths result in unexpected but feasible cases, resulting in defects.

Higher Quality Code Earlier In the Development Cycle

With correct development of static analysis tools, defects can be found early in the development cycle: If the code can be compiled, defects can be detected. Detection with a compile-time warning can be developed as a custom warning for the environment and application, warning of potential endianness errors. Even with false positives, our experience has shown that clustering of warnings on issues can indicate problems where protocols and/or data definitions may not quite be clean enough, which could warrant recoding. And any software engineering requirement that all code should pass a designated coding standard could certainly be applied to passing static analysis tools that strictly enforce the standard and detect defects and enforce correctness.

Some Basic Best Practices for Utilizing Static Analysis Tools

Technology cannot always be left to do the job: the working plans and processes of the software engineering teams must be considered.

The beginning of the project is the best phase to define and institute any endian defect detection processes: "How are we going to find the problems?" The answers to that question are done in parallel with identifying the strategy for coding around endian issues:

- "Byte-swapping methods?"
- "Data definitions?"
- "What is the correct process?"

If static analysis tools are to be used, the most important phase is in the definition of violations and corrections. This is the phase when detection methods can be developed. Once these are developed, the results of the detection during development can be augmented by retuning the algorithms to enhance analysis detection and iterating on more improvements.

The final step in the process is enforcing fixes to be made, reviewing false positives for validity, and maintaining a regular analysis schedule to detect defects as early as possible.

— Carl Ek is an architect at Code Integrity Solutions (http://www.codeintegritysolutions.com/), a consultancy focused on helping companies get more value from their static source code analysis and build tool investments. Prior to Code Integrity Solutions, Carl held senior engineering positions at Electronic Arts and in the compiler development division at IBM.

Return to Table of Contents

iPhone: Recording, Playing, and Accessing Video

Putting the SDK to work

By Brandon Trebitowski

One of the major updates included with the iPhone 3G was a built-in video camera. This allows users to easily record video and save it to their media library. The code for recording video is almost identical to the code to show the camera. It does, however, have a few checks that are required. Listing 1 shows the code for bringing up the video camera interface.

Listing 1: Displaying the video camera

```
-(void) showVideoCamera {
          if ([UIImagePickerController                                          #1
               isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {
              myImagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;
              } else {
                     NSLog(@"Camera not supported");
                     Return;
               }
                                                                               #2
NSArray *media = [UIImagePickerController availableMediaTypesForSourceType:
       UIImagePickerControllerSourceTypeCamera];

        If([media containsObject:kUTTypeMovie]) {                              #3
                 myImagePicker.mediaTypes = [NSArray
                       arrayWithObjects: kUTTypeMovie,nil];
                 [self presentModalViewController:myImagePicker animated:YES];  #4

                 } else {
                       NSLog(@"Video not supported");
                 }
}

#1   check to see if the camera is available
#2   get a list of the media types the camera supports
#3   check to see if the camera supports video
#4   show the video camera
```

The first thing we are doing in #1 is checking to see if the device has camera support. There are two cases where this would return False. The first is if the user has an iPod touch. As of this writing, the iPod touch does not support taking photos. The other case is if the camera is damaged on the iPhone.

In #2, we are checking to see what media types are supported by the camera. In this case, we are looking for the media type *kUTTypeMovie*. If this is found, then the camera will support video. #3 performs this check and sets the media type of our picker to *kUTTypeMovie* to tell it to display the video camera. By default, it is set to *kUTTypeImage*, which specifies photos, so it is necessary that we set it.

Finally, in #4 we display the video camera on the screen. One great feature that Apple added is the ability to edit the video on-the-fly. This is very easy to integrate in our code. Simply add this line prior to displaying the video camera:

```
myImagePicker.allowsEditing = YES;
```

This great one-liner from Apple adds a ton of functionality. Once the user finishes recording the video, the delegate method *didFinishPickingMediaWithInfo*: for the picker will be called. The dictionary passed to this method will contain a system path URL to the video file that was just recorded. Listing 2 shows how to use this path to retrieve and play back the video that was just recorded.

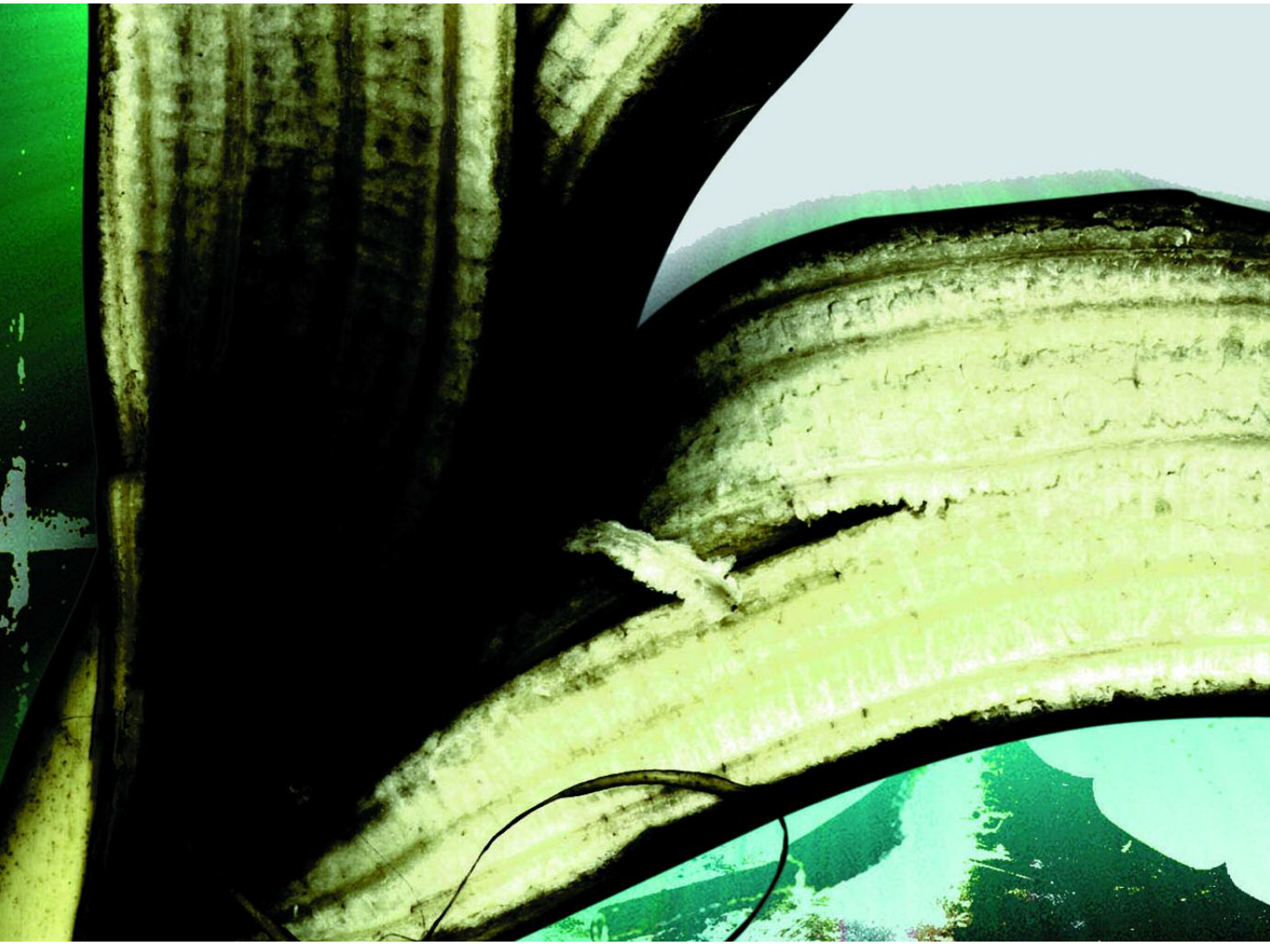
Listing 2: Playing the recorded video

```
- (void)imagePickerController:(UIImagePickerController *)picker
            didFinishPickingMediaWithInfo:(NSDictionary *)info {
            NSURL * pathURL = [info objectForKey:
UIImagePickerControllerMediaURL];
           MPMoviePlayerController * player =
                  [[MPMoviePlayerController alloc]
                      initWithContentURL:pathURL];
          [player play];
}
```

The first thing this method does is retrieve the path URL from the info dictionary. The path URL is the object stored with the key *UIImagePickerControllerMediaURL*. Next, an *MPMoviePlayer-Controller* is allocated with the contents of the path URL. This will load the video and prepare it to play. The last thing to do is call the play method and the video will begin.

— *Brandon Trebitowski is the author of* iPhone and iPad in Action *(http://www.manning.com/trebitowski/).*

Return to Table of Contents



Technical Writing for the Kindle

What does Kindle mean for the authors of technical books?

By Al Stevens

The "eBook" paradigm is gaining popularity mainly because of the success of Amazon's Kindle e-reader device and the hundreds of thousands of eBooks available for download. This popularity was underscored by Amazon's recent announcement that its eBook sales have for the first time surpassed sales of hard-back (not paperback) books. Other companies are on the bandwagon, including Apple with its iPad and Barnes and Noble's with its nook. eBook reader software is available on PC and Apple platforms. You can download eBooks to your iPhone, Blackberry, and iPod.

At relatively high prices. Kindle books are typically narrative text, fiction, and nonfiction comprising mostly words. Programming books have generally lagged behind mainstream books because technical books have requirements beyond that. This article is about using tools available to Kindle authors to write and publish computer programming books for the Kindle.

Pros and Cons

The advantages to using a Kindle are threefold:

- Books are instantly available for wireless download without an external Internet connection
- Books are usually less expensive than their printed counterparts
- You can carry your entire library with you wherever you go.

Many books are free. Many others cost only $0.99.

The advantages to authors are even greater. Kindle authors are typically self-publishers. You write, edit, layout, and publish your work yourself.

Kindle is like a "vanity press," in which authors pay a publisher to edit, produce, and print the book. But with Kindle you don't pay anyone. And you do all the work yourself.

With traditional publishing, you submit a proposal to a publisher or literary agent and wait for a publishing contract. Or a rejection. If your book is accepted, you deal with acquisitions editors, copy editors, production editors, and all that until the book is finally published. Not so with a Kindle book. You do it all. Your book is not rejected simply because a publisher thinks it won't sell. You even decide the price to charge.

Whatever you submit to Amazon gets published, assuming, of course, you have the rights to publish it. If you want to change something later — content, price, cover — do so and upload the changes.

Publishing on Kindle gives you access to Amazon's Author Page, which provides an online personal website for you to list your books, both print and Kindle editions. Here's a link to my Author Page on which you can find links to my books: http://www.amazon.com/-/e/B001K8M1SA.

There are disadvantages to being a Kindle author, too. Some of them align with the advantages just discussed. The good news is also the bad news.

First, you are your own worst editor, which means the book is no better than your ability to write and produce a book. Second, there is no marketing support. No salesmen plugging your book at bookstores. No bookstore shelves for exposure. No reviews in newspapers and magazines. And third, you compete with hundreds of thousands of other Kindle books, many — perhaps most — of which are not very good. Remember, this is the poor man's vanity press.

The Kindle is not particularly conducive to research. You can't flip pages, for example, and a traditional index is impossible to produce because there are no page numbers. There being no pages, as such, there can be no footnotes, although endnotes are possible. The page format has limited dimensions and is user-configurable,

which can defeat your best formatting intentions. Source code, tables, and such can be a bit of a challenge.

Disadvantages notwithstanding, now is an opportune time for computer authors to join the eBook trend. Print media computer books are on the critical list. Unless you are writing about the latest trendy hot topic, publishers are not as interested in programming and applications books as they used to be.

The Kindle Format

I've been working on some Kindle programming book projects, and I hit some walls. The device has format limitations that get in a tech writer's way. Much of what you need to know to fit technical content onto a Kindle is either undocumented or just hard to find.

I had to learn how to publish software code, screen shots, equations, tables, lists, flow charts, and so on, all the things you typically find in a programming book that are not narrative text.

Writing a novel? You can find lots of online help in formatting narrative text in the forums at Amazon's Digital Text Platform Community Support forum website (http://forums.digitaltextplatform.com/dtpforums/index.jspa).

Your technical book will have narrative prose, too, so you should get all such help. This article deals specifically with managing technical content in a book.

I'll describe the process that works best for me. There are other ways to do it. This is the procedure I've chosen.

Write the Book

First, you write the book. I use Microsoft Word for that task, building a Word document file for each chapter. You can put all the chapters in one file, but that makes for a cumbersome document, difficult to manage.

Use Word's hierarchy of Header styles to organize each chapter's outline. That facilitates automatic generation of a table of contents later. Don't worry about fonts at this time. Just use whatever makes it easiest for you to read and review as you write.

Use Word's Normal style for the narrative.

I built my own styles for figure captions and program listing titles. Those elements are centered and italicized using Kindle's default text font.

Illustrations

Build illustrations and figures with whatever image editor program you prefer. Embed them in the chapters by using Word's Insert/Picture/From File command. You can flow text around pictures if you want, but it's better to have each figure stand alone.

The Kindle is a monochrome device, and screen shots can be a bit hazy in that format. You might want to play with the contrast settings in some figures. Make sure you critically preview all your illustrations before you publish the book.

If you use the GIF image format, ensure that the transparency mode for those files is disabled. Kindle doesn't like it.

Page Layout

Kindle's pages are not fixed-length. Page breaks depend on which Kindle is used and how the user configures the display. Don't depend on page integrity to enhance your visual presentation. It won't work. Where you absolutely need a page break, use Word's Page Break command (Ctrl+Shift) to force one.

Cover Image

Use an image editor to make your cover image. Make the image 1200 pixels high by 900 pixels wide. You will include the cover image at the front of the book. You will also upload it when you publish the book so the book's listing at Amazon has a cover to display.

Source Code

For code, begin with a paragraph style that has 10-point Courier font. Word has a style named *code* that fits those criteria. Then after keying the code in, select the code and apply Word's HTML code style, a character style that you apply to keywords embedded in narrative text as well as to program listings.

Lines of code should be less than 47 characters to avoid word wrapping on the smaller Kindles. If that is not possible, advise your readers in the introductory chapter to read code-ridden chapters in landscape mode and perhaps with a smaller font.

Tables

Kindle does not display tables very well. The Kindle 1 does not support borders. So, don't use Word's table feature to portray tabular data. Instead, build tables as illustrations with a graphics editor program and embed them in your Word document as you do other graphical images.

Equations

For equations, use Word's Insert/Object/Equation Editor, with which you create and edit an equation image that Word inserts into the document as a graphical image.

When you save the chapter as HTML (discussed later) Word saves equation .gifs in a folder named [filename]_files. These images are badly formed. Word saves the .gif with incorrect transparency data, and you have to fix it.

If you use a browser to display the chapter, the equation looks fine. But when converted to Kindle format, the equation displays as a solid black rectangle. Interestingly, when you open the equation's .gif in an image editor, such as Windows Paint, it too displays the solid block.

Open the .gif in a program such as Paint Shop Pro (Windows Paint won't do), and change the first color in the color palette to solid white. That's the background color, which is black, and which is supposed to be transparent but isn't. Save the .gif using the option that disables transparency. That fixes the problem. You must do that for all the equation images in all the chapters in the book.

Links

Your Kindle book might have shortcuts that allow the reader to move from the current page to another topic referenced elsewhere in the book. That topic might be in the current or a different chapter. You can use Word's Insert/Hyperlink command for these links. Open the Hyperlink dialog and specify the text to display in the hyperlink and a place for the link to go when the reader clicks it.

If the link is to a different chapter file, your first tendency is to use the .doc filename. But that won't work. The .doc file is only where you saved the chapter. It is going to be converted to HTML in a subsequent step, and the book will be built from HTML files. So, when you build the hyperlink, have it point to the filename you will use for the chapter's HTML file.

Links need a fully qualified path. Most likely all your html files will be in a common folder, in which case you can prefix the linked filename with the dot-backslash (.\) token, which Word expands to a fully qualified path.

If the link is to a location in the same chapter, that location should have a paragraph header associated with it. The Hyperlink dialog includes a section titled Place in this Document, which shows the document's outline as defined by Header styled text. Select that option and choose a paragraph title to which to link.

Save to HTML

Save each chapter file as the Web Page, Filtered file type. That builds an HTML document without a lot of the excess CSS and HTML code that Word adds. From this point forward you'll work with HTML files in a text editor.

Tweak the Code

There are a couple of changes to make to the HTML files to get code to display properly. Since every line of code is a paragraph as far as Kindle is concerned, Kindle will indent it to the default paragraph indent level. In order to get the maximum line width, you can set the indents for code to zero. Open the chapter's HTML file and insert this line in the CSS <style> code at the top of the file.

```
p.code { text-indent:0in; text-align:left;}
```

This assumes that you use the Word style named "code" for code listings.

If your code has blank lines that are important, you'll have to manually insert them. The conversion process, described soon, eats blank lines in code listings. Look for all places in the HTML file for where you find this code:

```
<p class=code>'<code> </code></p>
```

Insert this tag after it:

```
<br>
```

Don't use multiple adjacent <*br*> tags for multiple blank lines. Insert a <*br*> tag after each <*p*…> tag that represents a blank line.

Make an Index

An index is a daunting task for any publication medium, but the Kindle makes it even more difficult. Kindle has no page numbers. Consequently the traditional format with indexed items displayed to the left of one or more page numbers does not work.

You can, however, build an index that associates indexed items with a list of hyperlinks to paragraph headings in the text. Doing a comprehensive index that way for a book of any size is an huge, tedious, error-prone task.

Many authors consider it not worth the effort, particularly since the Kindle has a text searching feature. But indexes often use phrases not found in the narrative as indexes items.

I have not built one yet, but I am a strong believer in having a comprehensive index in a technical book. Years ago, I wrote and published in *Dr. Dobb's Journal* programs to assist in extracting indexeed items from word processing files and automatically building the index. Those programs were, of course, relevant to traditional publishing media and indexed to page numbers. No doubt before I tackle my first Kindle index I will develop a way to streamline the procedure with software. The plan is to use a special Word character style that can be used to mark indexed items in the Word versions of the chapters. These styles serve only as markers.

Then the first program I write will scan the HTML files and insert anchor tags into the text that identify the paragraphs. Another program will extract the terms and association them with the paragraphs in which they are found. A sort and a formatting process builds the HTML file for the index.

Convert to Publishing Format

At this point, you have a number of HTML chapter files ready for publication and a cover image. You need to get all this together into a package in a format for publication. I use the freely available program named Mobipocket Creator (http://www.mobipocket.com/en/DownloadSoft/ProductDetailsCreator.asp) for that procedure.

To build the publishable book file, make a new project in Mobipocket Creator. This step creates a .opf file in the a subfolder of the My Publications folder in My Documents. Subsequently double-clicking that file starts up Mobipocket Creator with your project loaded.

Add to the project all the chapters' HTML files in the order in which they should appear in the book. Add the cover image to the project. Open the Table of Content view and fill in the form telling Mobipocket Creator which HTML tags (h1, h2, etc.) to use in the table of contents. Click the Build button. Mobipocket Creator builds a file with the filename extension .prc. That is your publishable file.

Preview the Book

As you build your book, you should preview it frequently, particularly the technical content. While you have only html to work with, before converting to publishing format, you can preview the book in your browser, which gives a rough preview of how the content is formatted. But your CSS and HTML code might include features that Kindle does not support. Eventually you need to see the real thing.

Amazon's Digital Text Platform includes a Preview feature that displays the book in a webpage popup window. To use that preview, you must open a new project and upload the book's content. This process is time-consuming, and the preview does not accurately emulate Kindle's display device. There is a more convenient way. Downloaded the free Kindle for PC application (http://www.amazon.com/gp/feature.html/ref=kcp_pc_mkt_lnd?docId=1000426311) to preview your work.

Install the program. Then double-click the .prc file that Mobipocket Creator created with its Build command. That runs Kindle for PC with your book open. This is a reasonable facsimile of the book as Kindle will display it. The illustrations are crisper and in color, however, and the display itself has more flexibility with dimensions, so you should do the final preview on the Kindle itself.

To preview on the Kindle, connect it to your computer with its USB cable. It looks like a disk drive on your PC just as cameras, memory chips, and MP3 players do. Copy the .prc file to the Kindle's Documents folder. Eject the Kindle and go to its Home page. Your book is now among the other books stored in your Kindle, and you can preview it.

Check out the illustrations and code text to ensure that they display properly.

Publish Your Book

Publishing a Kindle book involves uploading the book's contents to Amazon's Digital Text Platform website (https://dtp.amazon.com/mn/signin).

You must set up an account and be logged in. There is no charge for the service. You create each new project from this site. You upload the book's content file and its cover image to the dashboard. After you set a price and assure Amazon that you have rights to the book, you can publish it. A day or two later the book is available for readers to download from Amazon's Kindle Store website. Amazon keeps an account of your sales and sends you some money from time to time.

— Al Stevens is the author of numerous programming books and a long-time contributor to Dr. Dobb's. Al can be contacted at alstevens.com

Return to Table of Contents

YAFFS2: Yet Another Flash File System

Welcome to the Flash File System club

By Sasha Sirotkin

As Linux has become more widely used in embedded systems, the number of file systems that work directly with the flash storage (i.e. via MTD device as opposed to some block device hardware emulation layer such as the one present on most DiskOnKey devices) has grown substantially.

First, embedded Linux systems used read-only CramFS and SquashFS file systems. These are still very much in use today, as many embedded devices such as routers do not need a real read-write file system. Such devices typically store only a small amount of configuration information that can change, occupying only a few KBs (i.e. less than a single flash block and usually written directly to the flash while root file system resides on CramFS or SquashFS).

As the flash sizes increased and Linux moved into more embedded niches, the need for read-write flash file systems was answered by JFFS2 (http://sources.redhat.com/jffs2/), which for a long time was the de-facto standard Linux flash file system. As flash sizes grew even more and devices such as cellular phones that store large amounts of information (pictures, mp3 files) started using Linux, JFFS2 reached its scalability limits. As a result, new file systems specifically designed for large NAND flash devices were developed — UBIFS (http://www.linux-mtd.infradead.org/doc/ubifs.html), LogFS (http://www.logfs.org), and YAFFS. For a long time only UBIFS was part of the mainline kernel and both YAFFS2 and LogFS were available as a patches. At some point, it appeared as though the development of LogFS stagnated, with the latest patch available for kernel version 2.6.24. However, LogFS suddenly resurfaced and rather surprisingly was quickly merged into kernel 2.6.34, indicating that its developers kept working on this project, albeit with little publicity. YAFFS2, which contrary to LogFS was widely used, has undergone a similar process with respect to inclusion into mainlaine Linux kernel. In the past, YAFFS2 developers did not make any significant effort to put it into the mainline kernel, but that is going to change now.

When I started my embedded file system evaluation, I was almost certain that eventually I would choose UBIFS simply because it is part of the mainline kernel and YAFFS2 is an external patch. However, it turned out that YAFFS2 is actually easier to configure — I kept getting errors while mounting the UBIFS partition until I disabled the "Verify NAND page writes" kernel parameter. Apparently this is a rather old and well-known bug, still present in kernel 2.6.32, which I use on my systems. This is pretty subjective, but I had zero issues with YAFFS2 even though I had to patch the kernel. The patch works smoothly as all YAFFS2 files reside in a single directory "fs/yaffs2" and the only files that need to be modified are those related to the build system.

YAFFS2

YAFFS, short for "Yet Another Flash File System" (http://www.yaffs.net/), is a fast robust file system for NAND and NOR Flash. YAFFS has been around for several years, mainly used with embedded systems and consumer devices. But YAFFS hasn't gained much traction, at least not until the release of YAFFS2 (http://www.yaffs.net/yaffs-2-specification-and-development-notes).

All flash file systems, including YAFFS2, have to support standard features such as wear leveling and must be power-down resistant. Contrary to JFFS2, YAFFS2 (as well as UBIFS) was initially designed for NAND flash even though it will work with NOR too, which has some peculiar characteristics. The differences between NAND and NOR flash are numerous and full comparison is beyond the scope of this article, but as far as flash file systems are concerned, the following ones are the most important:

- Capacity. NAND flash tends to be much bigger than NOR; 16MB–512MB vs. 1MB–32MB
- Access interface. Random access for NOR vs. serial access for NAND
- Speed. NAND is faster, especially for write and erase operations
- Reliability. NAND devices are allowed to ship with bad blocks

YAFFS was designed with the above NAND characteristics in mind. To improve the mount time of large flash devices, YAFFS2 will try to save the RAM summary of the file system status on shutdown, which can save the scan time. It only writes to the flash sequentially in order to be compliant with modern NAND specifications. YAFFS2 uses out-of-band OOB flash data to mark and skip bad blocks.

Core YAFFS2 algorithms dealing with flash were developed as a user space application. This code resides in *yaffs_guts.[ch]*. This clear separation between OS dependent and independent parts makes YAFFS2 very portable. It was initially developed for Linux but later was successfully ported to WinCE, eCOS, pSOS, and VxWorks. It is even supported by some bootloaders, such as U-Boot.

As with most flash file systems, YAFFS2 is a log structured file system, which allows it to write to flash storage sequentially. The entries in the log are either data or header chunks. Each chunk has an associated tag with the following information: object ID, chunk ID, sequencer number, byte count, and some additional fields.

- Object ID identifies which object, i.e. inode or dentry, the chunk belongs to.
- Chunk ID, along with byte count, gives the location of the chunk inside the object.
- Sequence number is used to indicate obsolete chunks since YAFFS2 cannot modify an existing one — each chunk has to be written only once. The sequence number is incremented when a new block (that is, a flash unit of erasure) is allocated for usage, thus allowing chunks to be sorted in chronological order.

File system	Mount time	Write time	Read time
JFFS2	6.01s	3m 1.12s	11.82s
YAFFS2	0.18s	1m 15.88s	1.10s
UBIFS	0.17s	1m 16.74s	1.33s

Table 1

When a file is modified, instead of changing the information directly in the storage as most file systems do, YAFFS2 writes a new chunk to the log. Over time, some flash blocks will have a certain number of deleted, i.e. obsolete, chunks that need to be garbage collected. The garbage collection process finds a suitable block depending on the number of available erased blocks — it will try harder by even processing blocks with a small number of obsolete chunks, going into something called "aggressive mode" if there are too few. When a block worth collecting is found, it will iterate through all its chunks and copy those that are in use to a new block. After that the block can be erased. YAFFS2 uses various data RAM structures in order to increase performance. For example, YAFFS2 holds a tree of chunks for each file, which allows it to quickly find data chunks that belong to the file. For each flash device there is a data structure that describes block information, including the state of each chunk of that block.

Using YAFFS2 is straightforward. Assuming you already patched the kernel, just erase your MTD partition using:

```
flash_eraseall /dev/mtd[x]
```

and mount it as YAFFS using:

```
mount -t yaffs /dev/mtd[x] /mnt/flash
```

You can copy files to it right away and YAFFS2 will take care of everything. It is also possible to create a YAFFS2 image offline using the "mkyaffs2image" utility, but the former option is preferrable as YAFFS2 will properly use the OOB bits it is responsible for.

Flash File System Evaluations

The Web is full of JFFS2 vs. YAFFS2 vs. UBIFS benchmarks, but since they tend to be very subjective, I decided to measure what is important for my particular application, i.e. mount time, writing performance, and file system scan. I tested all three on an ARM9-based AT91SAM926 processor with 64MB NAND flash and 128MB SDRAM, running the Linux 2.6.32 kernel with the latest YAFFS2 patch. Not surprisingly, both YAFFS2 and UBIFS performed significantly better than JFFS2. What was surprising is the fact that both the YAFFS2 and UBIFS performance were very similar. Mount time was measured using:

```
time mount -t jffs2 /dev/mtdblock0 /tmp/test/
time mount -t yaffs /dev/mtdblock0 /tmp/test
time mount -t ubifs ubi0:test /tmp/test
```

Write time using:

```
time tar xf /test/rootfs.arm.tar
```

That created an 81MB file system with 1865 files and read time using:

```
time du -shc /tmp/test/
```

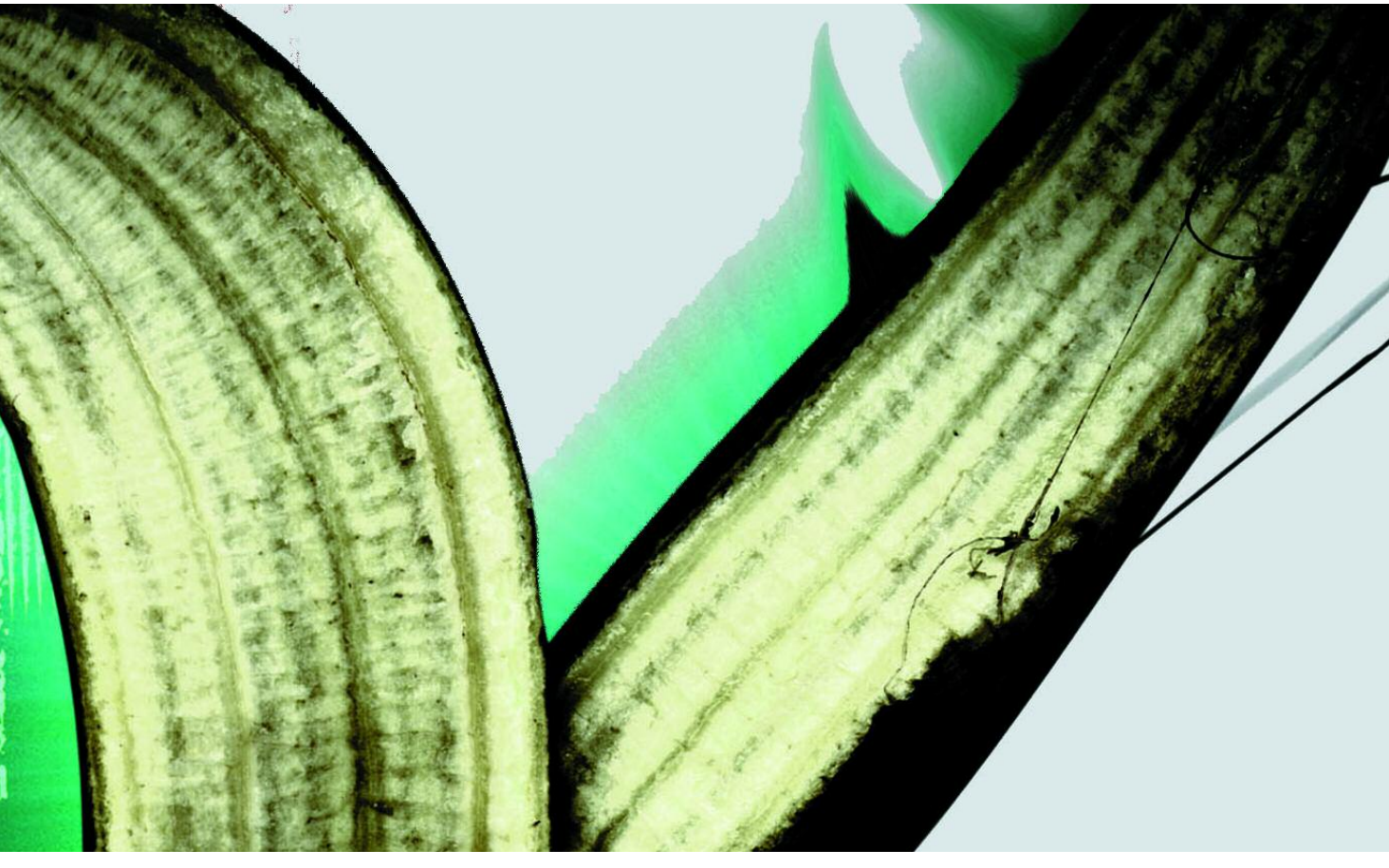
which scans the above file system. Table 1 summarizes the results.

As you can see, for some operations YAFFS2 is slightly faster than UBIFS, and for some it is the other way around. All in all, the difference does not look significant, at least for my sort of application, so I had a hard time choosing the right one for my FemtoLinux Project (http://femtolinux.com/). Eventually, I decided to go with YAFFS2; despite its "external patch" status, it seems to be more reliable and easier to use than UBIFS and it certainly has better track record in commercial products. Probably the only YAFFS2 disadvantage is the lack of compression support; however, for most embedded applications it is probably not an issue.

YAFFS2 is distributed free of charge under the GPL license. A different license is also available from Aleph One, which is a commercial company that develops and supports YAFFS. When YAFFS2 is used with the Linux kernel it is distributed under GPL; however, for other situations different licensing terms are available.

— Sasha Sirotkin currently works on the FemtoLinux Project (http://femtolinux.com/), improving Linux latency and real-time capabilities. FemtoLinux's design goal is to bring Linux performance closer to that of an RTOS, such as VxWorks, to allow complex real-time application development and easy porting.

Return to Table of Contents



Q&A: What's Behind Good Requirements

Requirements are the first step to delivering real software, to real people, in real businesses

by Jonathan Erickson

As VP of product development for DuckCreek Technologies, Michael Witt deals with requirements every day. He recently took time to talk with *Dr. Dobb's* editor in chief Jon Erickson.

Dr. Dobb's: What is the first step in implementing a requirements strategy?

Witt: A move to either introduce a requirements strategy where none currently exist or implement a different requirements strategy requires careful attention and planning to change management. As a first step it is critical to create a sense of need and urgency throughout the organization that will be affected. People know the difference between a real need and the next "silver bullet". Establish real need in the minds of everyone involved, but don't overlook capturing their hearts as well. Use both statistics & metrics and anecdotes to create a compelling case for change, and involve 20% of your organization in facilitating the change. A core group of knowledgeable and motivated individuals can facilitate change more rapidly than a single leader from a soap box. Finally, include in this opening salvo the training necessary to educate people on requirements development and management. This means that you have made some decision on the requirements model, the goals of your requirements program, the principles and best practices that you want to implement, and have at least narrowed down your tools options to no more than two or three potential vendors.

Dr. Dobb's: Who should be involved in defining and managing requirements?

Witt: The people who are going to use the software built from the requirements. That first sentence cuts to the chase; if you don't have people who will use the software involved in the requirements process then the software will not fit the intended business purpose. However, there are multiple skill sets required to create software requirements that are usable themselves. Requirements elicitation is a different skill from requirements engineering. It is generally a good idea to consider an out-facing product manager to be responsible primarily for requirements elicitation and a business analyst to be responsible for requirement engineering and development. The key to success is flexibility. If you have a hard and fast rule that business analysts can never talk to outside stakeholders, then you lose opportunities for speed and efficiency. With these two roles covered, you cannot overlook the need for the entire team to also participate on some level. If the team doesn't own the requirements, but is merely responsible for reading them and filling the order, then you lose ownership and choice. Employee engagement can enable you to do more with fewer resources than an organization that lacks employee engagement. Enabling ownership and choice among the development team in building products can make the difference between getting by and being wildly successful.

Dr. Dobb's: What should a requirements set cover?

Witt: Requirements should provide insight into the major functionality, features, and non-functional characteristics of the system. There are very few companies out there that are trying to build software that human lives depend on, but a lot of companies are building their requirements that way. Requirements should provide a general direction for the software, but software is a creative enterprise. Most organizations begin with requirements out of fear. They struggle to deliver software with business value and the growing disquiet among their user base drives them to determine that developers don't know what they are doing. Implementing requirements management in order to control developers is a mistake.

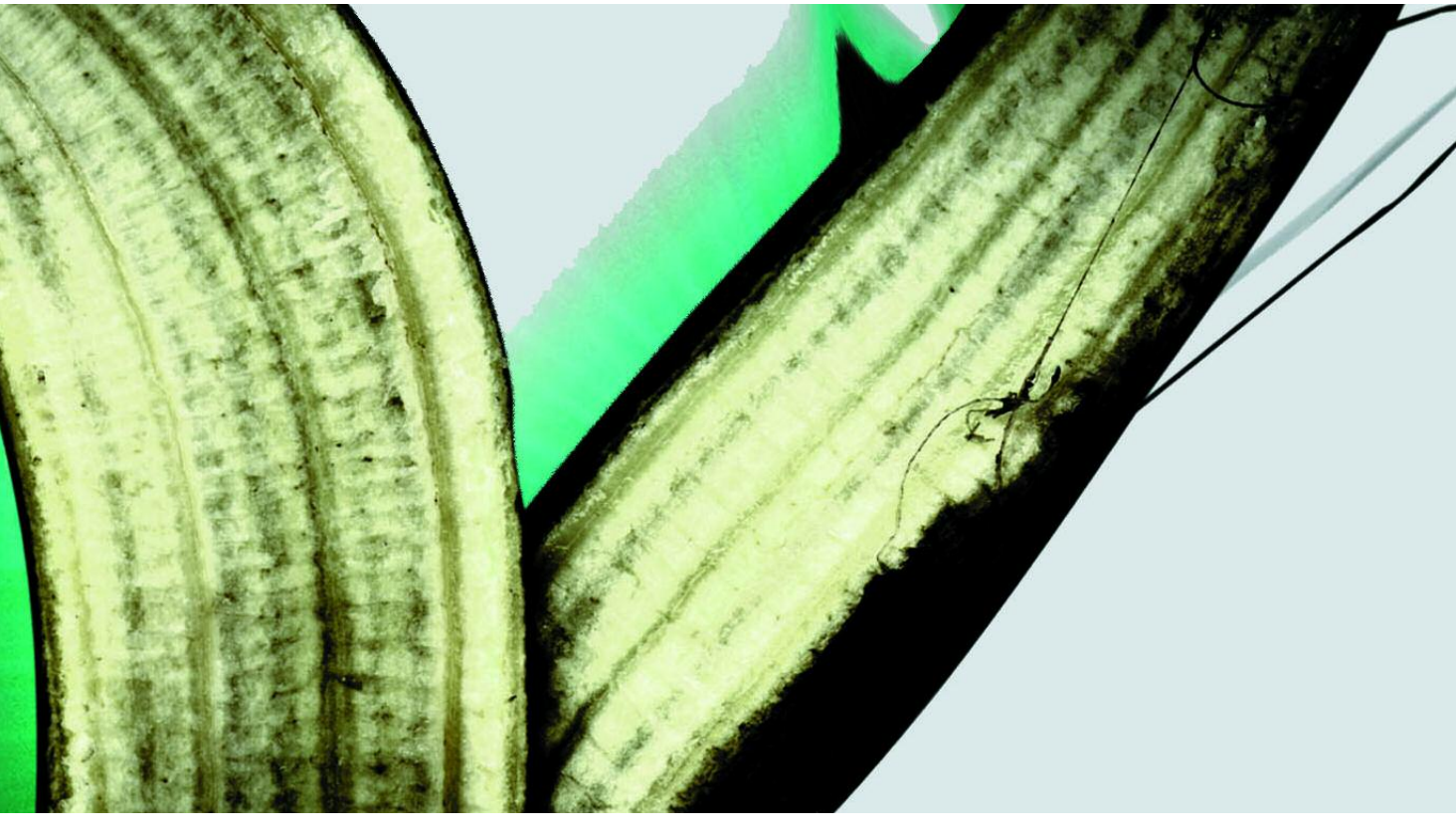
Requirements management should be implemented in order to drive business value. Keeping the goal of delivering business value as priority one in

your requirements effort will get you closer to the right coverage in your requirements set. Stay focused on business requirements, don't be afraid to specify technical requirements, and enable development teams to have a say in all of it.

Dr. Dobb's: Who should write, read, and sign off requirements?
Witt: I believe strongly in a team-based approach. I don't think that a single person can make the kinds of decisions required to ensure that a requirements set is right. Having individuals on the team responsible for their area of professional expertise allows you to distribute sign-off responsibilities to people who know. That being said, I have to ask why it is that sign-off is important. Is sign-off a part of your process of ensuring that you are delivering business value or is it a method of protection? As a method of protection requirements sign-off stinks because it is impossible to communicate the nuances present in a massive requirements document. In the end it isn't protection at all because your "customer" still ends up angry if you forget something. If you are using requirements to hold the customer or stakeholders accountable for their decisions then you are doing requirements wrong and for the wrong reasons. It is the job of those eliciting and engineering the requirements to ensure that they capture the business needs. And it is a mistake to use sign-off as a method of holding business owners accountable; all that does is enable the team to have an escape clause. Success in software is delivering products that meet business needs, not in delivering requirements that enable you to enforce a contract.

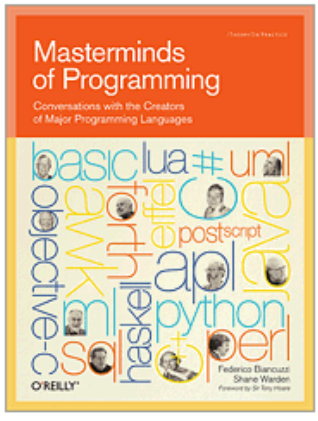
Dr. Dobb's: Are requirements cast in stone?
Witt: The only alternative path to change is obsolescence. Requirements should reflect the business needs and goals of the software. The easiest way to answer this question is to ask, How often do your business needs for the software change? By asking this question before you begin, software projects can be setup for success. If your business requires software that meets business needs that change monthly or quarterly (insurance software for Commercial Lines meets this criteria), then your requirements management system must enable the business need for change. Requirements are pointless in and of themselves, they serve only to deliver real software, to real people, in real businesses.

Return to Table of Contents



Jolt Awards: Excellence in Books

By Jonathan Kurz



Masterminds of Programming: Conversations with the Creators of Major Programming Languages
by Federico Biancuzzi and Shane Warden
O'Reilly Media
$39.99

For any programmers out there who have ever asked the question, "What were they thinking when they designed this language?" this book is for you. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages,* by Federico Biancuzzi and Shane Warden, takes you back to the early days of many programming languages to find out why the designers did what they did. Authors Biancuzzi and Warden do an outstanding job of asking pertinent questions of the designers. In fact, the questions and answers are more in the form of a conversation than a questionnaire. The chapters of the book are dedicated to a single language and can be read in any order.

While discussing the language "APL," the issue of complexity was raised. Having spent some time with APL, I was very eager to know what Adin Falkoff, codesigner of the language, had to say about this, which was: "I do not agree with the statement that it is 'complex.'" Indeed.

The languages covered in this book are not limited to programming. I was pleasantly surprised to read what Grady Booch himself had to say about the origin and evolution of UML. In this particular case, I found a strong correlation between the applied use of UML and its drive toward simplicity.

Sometimes it takes a book like this to gain an appreciation for the decisions that were made way back when. From the chapter on SQL, I learned how the language was intentionally made to be declarative rather than procedural for the reason that it facilitates optimization. Had the decision been made to make it procedural, SQL may not have become the ubiquitous and accepted database language that it is today.

What makes *Masterminds of Programming* particularly interesting and useful is that these conversations did not take place at the beginning of the languages, but rather in present-day. Having the benefit of time to reflect on strengths and weaknesses of the languages, their merits, and the context of why certain design decisions were made, gives readers a unique insight and perspective of these languages and their creators in a very enjoyable and thorough book.

Return to Table of Contents

5 Trillion Digits of Pi

By Mark Nelson

Back in 1981, fresh out of school, I was awestruck by Steve Wozniak's program that calculated over 100,000 digits of e on an Apple II. (Anyone who has a scan of his article in the June 1981 issue of *Byte*, please email me a copy!) Shortly after reading the article, I ported his program to the PDP-11 at my office and duplicated his results, down to the last digit.

These days the stakes are much higher when it comes to calculating the values of constants. Alexander Yee and Shigeru Kondo have just announced the calculation of pi to 5 trillion digits (http://www.numberworld.org/misc_runs/pi-5t/details.html). And oddly enough, this was accomplished on a desktop machine running Windows server, not the Linux cluster I would have expected.

Here are some key stats:

Operating System

Windows Server 2008 R2 Enterprise x64

Software

y-cruncher (www.numberworld.org/y-cruncher/)

Processor

2 Intel Xeon X5680 @3.33 GHz offering 24 hyperthreaded processors

Disk Space

The computation required roughly 22TB of disk space, and the compressed result takes another 3.8TB

Time

The task took 90 days to complete, and 66 hours to verify

The y-cruncher software package that broke this record also holds records for several other constants, including one trillion digits of e. So in 30 years, more or less, the desktop PC has gone from constant calculations under a million digits to over a trillion digits. That growth rate of 1.7x per year maps pretty well to Moore's Law (roughly, that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years), suggesting that we can expect these numbers to continue climbing for at least a few more years.

Kudos to Alexander Yee and Shigeru Kondo on a smashing accomplishment!

CLICK HERE TO COMMENT ON THIS POST
(http://www.drdobbs.com/blog/archives/2010/08/5_trillion_digi.html)

Return to Table of Contents

Prefer Using Futures or Callbacks to Communicate Asynchronous Results

Handling asynchronous communication back to the caller

By Herb Sutter

Active objects offer an important abstraction above raw threads. In a previous article, we saw how active objects let us hide the private thread, deterministically organize the thread's work, isolate its private data, express asynchronous messages as ordinary method calls, and express thread/task lifetimes as object lifetimes so that we can manage both using the same normal language features. [1]

What we didn't cover, however, was how to handle methods' return values and/or "out" parameters, and other kinds of communication back to the caller. This time, we'll answer the following questions:

- How should we express return values and out parameters from an asynchronous function, including an active object method?
- How should we give back multiple partial results, such as partial computations or even just "percent done" progress information?
- Which mechanisms are suited to callers that want to "pull" results, as opposed to having the callee "push" the results back proactively? And how can "pull" be converted to "push" when we need it?

Let's dig in.

Getting Results: Return Values and "Out" Parameters

First, let's recall the active object example we introduced last time.

We have a GUI thread that must stay responsive, and to keep it ready to handle new messages we have to move "save this document," "print this document," and any other significant work off the responsive thread to run asynchronously somewhere else. One way to do that is to have a background worker thread that handles the saving and print rendering work. We feed the work to the background thread by sending it asynchronous messages that contain the work to be performed; the messages are queued up if the worker thread is already busy, and then executed sequentially on the background worker thread.

The following code expresses the background worker using a *Backgrounder* class that follows the active object pattern. The code we'll show uses C++0x syntax, but can be translated directly into other popular threading environments such as those provided by Java, .NET, and Pthreads (see [1] for a discussion of the pattern and translating the code to other environments).

The *Active* helper member encapsulates a private thread and message queue, and each *Backgrounder* method call simply captures its parameters and its body (both conveniently automated by using lambda function syntax) and *Send* that as an asynchronous message that's fired off to be enqueued and later executed on the private thread:

```
// Baseline example
class Backgrounder {
public:
  void Save( string filename ) { a.Send( [=] {
    // … do the saving work …
  } ); }

  void Print( Data& data ) { a.Send( [=, &data] {
    do {
      // … do the printing work for another piece
of the data …
    } while( /* not done formatting the data */ );
  } ); }

private:
  PrivateData somePrivateStateAcrossCalls;
  Active a;  // encapsulates a private thread, and
};           // pumps method calls as messages
```

The GUI thread instantiates a single *Backgrounder* object (and therefore a single background worker thread), which might be used from the GUI thread as follows:

```
class MyGUI {
public:
  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    backgrounder.Save( filename );
      // this fires off an asynchronous message
  } // and then we return immediately to the caller

  // …

private:
  Backgrounder backgrounder;
};
```

This illustrates the ability to treat asynchronous messages like normal method calls and everything is all very nice, but you might have noticed that the *Save* and *Print* methods (in)conveniently don't have any return value or "out" parameters, don't report their progress or intermediate results, nor communicate back to the caller at all. What should we do if we actually want to see full or partial results?

Option 1: Return a Future (Caller Naturally "Pulls")

First, let's deal with just the return value and output parameters, which should be somehow communicated back to the caller at the end of the asynchronous method. To keep the code simple, we'll focus on the primary return value; output parameters are conceptually just additional return values and can be handled the same way.

For an asynchronous method call, we want to express its return value as an asynchronous result. The default tool to use for an asynchronous value is a "future" (see [2]). To keep the initial example simple, let's say that *Save* just wants to return whether it succeeded or failed, by returning a bool:

```
// Example 1: Return value, using a future
class Backgrounder {
public:
  future<bool> Save( string filename ) {
    // Make a future (to be waited for by the caller)
    // connected to a promise (to be filled in by the callee)
    auto p = make_shared<promise<bool>>();
    future<bool> ret = p->get_future();
    a.Send( [=] {
      // … do the saving work …
      p->set_value( didItSucceed() ? true : false );
    } );
    return ret;
  }
```

(C++0x-specific note: Why are we holding the promise by reference-counted smart pointer? Because promise is a move-only type and C++ lambdas do not yet support move-capture, only capture-by-value and capture-by-reference. One simple solution is to hold the promise by *shared_ptr*, and copy that.)

Now the caller can wait for the "future":

```
future<bool> result = backgrounder.Save( filename );
…
… this code can run concurrently with Save()
…
Use( result.get() );  // block if necessary until result is available
```

This works, and returning a "future" is generally a useful mechanism.

However, notice that waiting for a "future" is inherently a "pull" operation; that is, the caller has to ask for the result when it's needed. For callers who want to find out if the result is ready without blocking if it isn't, "future" types typically provide a status method like *result.is_ready()* for the caller to check without blocking, which he can do in a loop and then sleep in between calls — that's still a form of polling loop, but at least it's better than burning CPU cycles with outright busy-waiting.

So, although the caller isn't forced to busy-wait, the onus is still on him to act to "pull" the value. What can we do if instead the caller wants a "push" notification sent to him proactively when the result is available? Let's consider two ways, which we'll call Options 2 and 3.

Option 2: Return a Future (Caller Converts "Pull" Into "Push")

The "pull" model is great for many uses, but the example caller we saw above is a must-stay-responsive GUI thread. That kind of caller certainly doesn't want to wait for the "future" on any GUI thread method, because responsive threads must not block or stop processing new events and messages. There are workarounds, but they're not ideal: For example, it's possible for the GUI thread to remember there's a "future" and check it on each event that gets processed, and to make sure it sees it soon enough it can generate extra timer events to be woken up just so it can check the "future" —but that seems (and is) a lot harder than it should be.

Given that a responsive thread like a GUI thread is already event-driven, ideally we would like it to be able to receive the result as a new queued event message that it can naturally respond to just like anything else.

Option 2 is to have the caller do the work to convert "pull" to "push." In this model, the callee still returns a "future" as in Option 1, and it's up to the caller turn it into a proactive result. How can the caller do that? One way is to launch a one-off asynchronous operation that just waits for the "future" and then generates a notification from the result. Here's an example:

```
// Example 2(a): Calling code, taking result as a future
// and converting it into an event-driven notification
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    shared_future<bool> result;
    result = backgrounder.Save( filename );
    // queue up a continuation to notify ourselves of the
    // result once it's available
    async( [=] { SendSaveComplete( result.get() ); } );
  }
  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

The statement *SendSaveComplete(result->get());* does two things: First, it executes *result->get()*, which blocks if necessary to wait for the result to be available. Then, and only then, it calls *SendSaveComplete*; in this case, an asynchronous method that when executed ends up calling *OnSaveComplete* and passes the available result. (C++0x-specific note: Like promises, ordinary "futures" are intended to be unique and therefore not copyable, but are move-only, so again we use the *shared_ptr* workaround to enable copying the result into the lambda for later use.)

Option 2 Variant: ContinueWith

As written, the Example 2(a) code has two disadvantages:

- First, it spins up (and ties up) a thread just to keep the asynchronous operation alive, which is potentially unnecessary because the first thing the asynchronous operation does is go idle until something happens.
- Second, it incurs an extra wakeup, because when the "future" result is available, we need to wake up the asynchronous helper operation's thread and continue there.

Some threading platforms offer a "future-like" type that has a *ContinueWith*-style method to avoid this overhead; for example, see .NET's *Task<T>*. [3] The idea is that *ContinueWith* takes a continuation to be executed once the thread that fills the "future" makes the "future" ready, and the continuation can be executed on that same target thread.

Tacking the continuation onto the "future-generating" work itself with *ContinueWith*, rather than having to use yet another fresh *async* operation as in Example 2(a), lets us avoid both of the problems we just listed: We don't have to tie up an extra thread just to tack on some extra work to be done when the "future" is ready, and we don't have to perform a wakeup and context switch because the continuation can immediately run on the thread that fills the "future." For example:

```
// Example 2(b): Calling code, same as 2(a) except using
// ContinueWith method (if available)
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    shared_future<bool> result;
    result = backgrounder.Save( filename );
    // queue up a continuation to notify ourselves of the
    // result once it's available
    result.ContinueWith( [=] {
      SendSaveComplete( result->get() );
    } );
  }
  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

Prefer to use a *ContinueWith* style if it is available in your "futures library."

Option 3: Accept an Event or Callback (to "Push" to Caller)

Both of the alternatives we've just seen let the callee return a "future," which by default delivers a "pull" notification the caller can wait for. So far, we've left it to the caller to turn that "future" into a "push" notification (event or message) if it wants to be proactively notified when the result is available.

What if we want our callee to always offer proactive "push" notifications? The most general way to do that is to accept a callback to be invoked when the result is available:

```
// Example 3: Return value, using a callback
class Backgrounder {
public:
  void Save(
    string filename,
    function<void(bool)> returnCallback
  ) {
    a.Send( [=] {
      // … do the saving work …
      returnCallback( didItSucceed() ? true : false );
    } ); }
```

This is especially useful if the caller is itself an active object and gives a callback that is one of its own (asynchronous) methods. For example, this might be used from a GUI thread as follows:

```
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    // pass a continuation to be called to give
    // us the result once it's available
    shared_future<bool> result;
    result = backgrounder.Save( filename,
      [=] { SendSaveComplete( result->get() ); } );
  }

  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

Since Example 3 uses a callback, it's worth mentioning a drawback common to all callback styles, namely: The callback runs on the callee's thread. In the aforementioned code that's not a problem because all the callback does is launch an asynchronous message event that gets queued up for the caller. But remember, it's always a good idea to do as little work as possible in the callee, and just firing off an asynchronous method call and immediately returning is a good practice for callbacks.

Getting Multiple or Interim Results

All of the aforementioned options deal well with return values and output parameters. Finally, however, what if we want to get multiple notifications before the final results, such as partial computation results, updated status such as progress updates, and so on?

We have two main options:

- Provide an explicit message queue or channel back to the caller, which can enqueue multiple results.
- Accept a callback to invoke repeatedly to pass multiple results back to the caller.

Example 4 will again use the callback approach. If the caller is itself an active object and the callback it provides is one of its own (asynchronous) methods, we've really combined the two paths and

> It's always a good idea to do as little work as possible in the callee, such as just firing off an asynchronous call and immediately returning

done both bullets at the same time. (Note: Here we're focusing on the interim progress via the *statusCallback*; for the return value, we'll again just use a "future" as in Examples 1 and 2.)

```
// Example 4: Returning partial results/status
class Backgrounder {
public:
  // Print() puts print result into spooler, returns one of:
  //    Error (failed, couldn't process or send to spooler)
  //    Printing (sent to spooler and already actively printing)
  //    Queued (sent to spooler but not yet actively printing)
  future<PrintStatus>
  Print(
    Data& data,
    function<void(PrintInfo)> statusCallback
  ) {
    auto p = make_shared<promise<PrintStatus>>();
    future<PrintStatus> ret = p->get_future();
    a.Send( [=, &data] {
      PrintInfo info;
      while( /* not done formatting the data */ ) {
        info.SetPercentDone( /*…*/ );
        statusCallBack( info );   // interim status
        // … do the printing work for another piece of the data …
      } while( /* not done formatting the data */ );
      p->set_value( /* … */ );    // set final result
      info.SetPercentDone( 100 );
      statusCallBack( info );     // final interim status
    } );
    return ret;
  }
```

This might be used in the GUI thread example as follows:

```
class MyGUI {
public:
  // …

  // When the user clicks [Print]
  void OnPrintClick() {
    // …
    // … turn on printing icon, etc. …
    // …
    // pass a continuation to be called to give
    // us the result once it's available
    shared_future<PrintStatus> result;
    result = backgrounder.Print( theData,
      [=]( PrintInfo pi ) { SendPrintInfo( pi, result ); } );
  }

void OnPrintInfo(
  PrintInfo pi,
  shared_future<PrintStatus> result
) {
  // … update print progress bar to
  //     pi.GetPercentDone(), etc. …
  // if this is the last notification
  // (100% done, or result is ready)
  if( result.is_ready() ) {
    // … turn off printing icon, etc. …
  }
}
```

Summary

To express return values and "out" parameters from an asynchronous function, including an active object method, either:

- Return a "future" to invoke that the caller can "pull" the result from (Example 1) or convert it to a "push" (Examples 2(a) and 2(b), and prefer to use *ContinueWith* where available); or
- Accept a callback to invoke to "push" the result to the caller when ready (Example 3).

To return multiple partial results, such as partial computations or even just "percent done" progress information, also use a callback (Example 4).

Whenever you provide callbacks, remember that they are running in the callee's context, so we want to keep them as short and noninvasive as possible. One good practice is to have the callback just fire off asynchronous messages or method calls and return immediately.

On Deck

Besides moving work off to a background thread, what else could we use an active object for? We'll consider an example next time, but for now, here's a question for you to ponder: How might you use an active object to replace a mutex on some shared state? Think about ways you might approach that problem, and we'll consider an example in my next column.

References

[1] H. Sutter. "Prefer Using Active Objects Instead of Naked Threads," *Dr. Dobb's Digest*, June 2010, www.drdobbs.com/high-performance-computing/225700095.

[2] H. Sutter. "Prefer Futures to Baked-In 'Async APIs,'" *Dr. Dobb's Digest*, January 2010, www.drdobbs.com/high-performance-computing/222301165.

[3] *Task.ContinueWith* Method (*Action<Task>*); *MSDN*; http://msdn.microsoft.com/en-us/default.aspx.

— Herb Sutter is a bestselling author and consultant on software development topics, and a software architect at Microsoft. He can be contacted at www.gotw.ca.

Return to Table of Contents