United Business Media

Dr.Dobb's DIGEST

The Art and Business of Software Development February, 2010

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

Is There a Digital Sign in Your Future?

By Jonathan Erickson,
Editor In Chief

With an eye towards standards-based, interactive digital signs, Intel and Microsoft have teamed up to deliver an optimized Windows 7-based Windows Embedded Standard 2011 and Intel Core microarchitecture platform for digital signs. Among the platform highlights are:

- Rich immersive user experiences that include panning and zoom, and touch and gesture input to improve web browsing and the overall user experience.
- Video analytics capabilities that allow advertisers to measure the effectiveness of their campaigns through built-in camera technology, which analyzes users' physical attributes and then generates tailored ad displays.
- Remote deployment and management capabilities that minimize onsite maintenance, updates, and repairs.
- Power management through conditional UIs that react to the environment around them.

The demo Intel recently presented simulated a virtual brick-and-mortar store setting where customers could use the multi-touch holographic screen to explore merchandise, find out about promotions, submit feedback on products, read customer reviews, view past purchasing histories, and share what they discovered with their friends via social media and mobile phone integration.

For more information, click http://www.windowsembedded.com/retail to see Microsoft's take, and http://www.intel.com/go/digitalsignage for Intel's take.

But you have to wonder if the world is really ready for a digital signage future. Maybe we should ask drivers in downtown Moscow who recently were entertained, if that's the right word, by a two-minute pornographic video being displayed on a digital highway billboard sign instead of what was supposed to be regular advertising clips. Traffic came to a halt as explicit whatevers appeared on the 9×6-meter display. The sign's owner blamed a hacker. At least they could have added "a hacker with a sense of humor."

Which gives me the opportunity the solicit the unknown programmer for an article on how you go about hacking a digital sign. Send in the article at your earliest convenience. Text only, please.

Return to Table of Contents

Parallel Algorithm Leads To Crypto Breakthrough

Massively parallel algorithm iteratively decrypts fixed-size blocks of data

Pico Computing (http://www.picocomputing.com/) has announced that it has achieved the highest-known benchmark speeds for 56-bit DES decryption, with reported throughput of over 280 billion keys per second achieved using a single, hardware-accelerated server.

"This DES cracking algorithm demonstrates a practical, scalable approach to accelerated cryptography," says David Hulton, Pico Computing Staff Engineer and an expert in code cracking and cryptography. "Previous methods of acceleration using clustered CPUs show increasingly poor results due to non-linear power consumption and escalating system costs as more CPUs are added. Using FPGAs allows us to devote exactly the amount of silicon resources needed to meet performance and cost goals, without incurring significant parallel processing overhead."

Hulton's DES cracking algorithm uses brute force methods to analyze the entire DES 56-bit keyspace. The massively parallel algorithm iteratively decrypts fixed-size blocks of data to find keys that decrypt into ASCII numbers. This technique is often used for recovering the keys of encrypted files containing known types of data. The candidate keys that are found in this way can then be more thoroughly tested to determine which candidate key is correct.

Such brute force attacks are computationally expensive and beyond the reach of software algorithms running on standard servers or PCs, even when equipped with GPU accelerators. According to Hulton, current-generation CPU cores can process approximately 16 million DES key operations per second. A GPU card such as the GTX-295 can be programmed to process approximately 250 million such operations per second.

The 56-bit Data Encryption Standard (DES) is now considered obsolete, having been replaced by newer and more secure Advanced Encryption Standard (AES) encryption methods. Nonetheless, DES continues to serve an important role in cryptographic research, and in the development and auditing of current and future block-based encryption algorithms.

When using a Pico FPGA cluster, however, each FPGA is able to perform 1.6 billion DES operations per second. A cluster of 176 FPGAs, installed into a single server using standard PCI Express slots, is capable of processing more than 280 billion DES operations per second. This means that a key recovery that would take years to perform on a PC, even with GPU acceleration, could be accomplished in less than three days on the FPGA cluster.

"Our research efforts in cryptography and our real-world customer deployments have given us unique insights into parallel computing methods for other domains, including genomics and simulations," added Pico Computing's Robert Trout. "The use of an FPGA cluster greatly reduces the number of CPUs in the system, increases computing efficiency, and allows the system to be scaled up to keep pace with the data being processed."

Return to Table of Contents

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.ddj.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™

What Developers Think

Consider these 7 trends as you craft IT strategy

by Jeffrey Hammond

When it comes to the technologies that software developers use to do their jobs, IT managers and executives don't always have a clear picture of what's going on at the ground level in their development shops. A series of Forrester Research surveys over the past three years have shown that.

We attribute this knowledge gap, which exists regardless of company size or industry, to "technology populism" — the growing trend where tech-savvy workers such as developers make their own decisions about what technologies to use. They help themselves to the software, collaborative tools, and information sources that best fit their needs, with minimal or no support from central IT. Much of this activity flies under management radar.

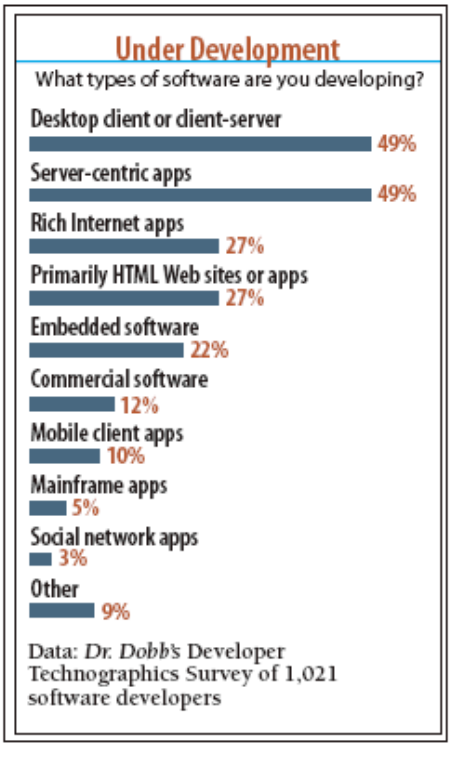


Figure 1: What types of software are you developing?

Developers were among the first users to bring consumer technologies — like instant messaging, mashups, wikis, and mobile devices — to their jobs. They're often the first IT staffers to evaluate and use new software technologies. So it's valuable to track what they're doing.

To better understand what's going on in development shops, Forrester conducted a survey of more than 1,000 platform-agnostic, programming-language-independent *Dr. Dobb's* readers. Collectively, they best represent the software development community. We asked them what types of applications they're writing, how they're writing them, and what they think about the state of application development. Our survey turned up seven trends that could have major implications for your IT strategy.

1. RIAs Are For Real

While client-server and server-based applications are the most common types of software projects that our survey respondents are involved in, there's a third technology to keep an eye on: rich Internet applications (see Figure 1).

RIAs are slowly replacing HTML when it comes to website development because they combine the more-interactive user experience of rich client applications with the location-independence and reduced deployment costs of traditional web applications. Industry stalwarts such as Microsoft, Oracle, and Sun are doing well with this transition, but other vendors, including Adobe and Google, are gaining mind-share as

developers opt for their tools and frameworks, and push to get these vendors added to their companies' stables of platform suppliers.

2. Wide Use Of Open Source

Nearly four out of five developers use open source software for development or application deployment (see Figure 2). That's much higher than the 40% of IT managers and executives who say their companies use open source software.

Developers are incorporating open source software into their application platform infrastructure in various ways, including:

- More Java developers use Apache Tomcat and Red Hat JBoss as their primary application server than either IBM WebSphere or Oracle WebLogic.
- MySOL and PostgreSQL rank third and fourth, respectively, behind Microsoft SQL Server and Oracle as the primary databases used for deployed apps.
- More than one-third of developers use Subversion for source code management; that's almost triple the share of the next most-used SCM tool, Microsoft SourceSafe.

The majority of developers still write code on Windows PCs, but when they deploy apps, almost one in four targets Linux as their primary deployment operating system. And while 5% of developers have moved to a Mac as their primary development machine, most of them are targeting Linux or Windows as a primary deployment operating system, not OS X.

Linux's popularity has grown as more developers build web applications and shift their focus to the server components, especially in the Java/Web space. The economies of scale when deploying to a LAMP stack, or Linux/Apache/Tomcat or Linux/Apache/JBoss combinations, make Linux on the server an attractive proposition. This is especially true in environments making heavy use of virtualization, since developers don't have to stop and ask, "Do I have a license for that image I just created?"

The implication of all this is clear: If you don't have an open source strategy in place, you need one — now!

3. Virtualization, Cloud Use Evolves

Developers appear to understand the benefits of virtualization and are getting rid of the testing server under the desk that used to be a development machine. More than a third (36%) of respondents develop applications on a virtual image that IT manages, and almost as many (28%) deploy to virtual server instances.

But cloud computing is just gaining traction among developers. About 4% of respondents are deploying applications to the cloud. Of those, Amazon's EC2 is the preferred target. That means if you're just getting started on a cloud strategy, relax — you still have time to develop one that builds on the idea of a virtual private cloud, combining the best of your own data center resources with those found in the public cloud.

4. Multilingual Developers Emerge

Developers used to identify themselves by the languages that they used — "I'm a COBOL programmer," "she's a Java developer." But that's changing — less than 15% of the developers we surveyed spend all their time writing in a single language.

The Web has brought about a new type of developer, one who identifies with platforms and runtimes that are tuned to building particular types of applications such as CRUD-style ("Create, Read, Update, Delete") web applications that use tools such as Ruby on Rails and media-centric applications that use Adobe Flash. Multilingual developers create applications for these fit-to-purpose platforms, using no one language more than half the time. They're comfortable working in environments where mixing servers written in Java, .NET, and PHP with clients written in JavaScript and Adobe Flex is practically second nature. One impli-

Figure 2: Which infrastructure tools have you used as part of your development work?

cation of this is that when hiring developers, it's just as important to assess how quickly they can learn a new language as it is to look at the languages they already know.

Understand how your organization is changing, and that you can't hold back tech populism among your developers

5. Young Developers Drawn To Dynamic Languages

When we looked at age demographics, we weren't surprised to see that COBOL developers tended to be the oldest respondents — after all, the language turned 50 last year. It's also no surprise to see that dynamic languages such as Ruby, Python, PHP, and JavaScript are proving most popular with developers in the 45-and-under cohort. Dynamic languages are useful when it comes to assembling components into composite web applications, especially if runtime composition is important.

The implications? As the development staff at a shop turns over, the new generation will push to adopt these dynamic languages. IT managers must ensure that processes and application lifecycle management tools can handle the changes that these new languages bring to the development shop.

6. Agile Processes Resonate

Agile processes are growing ever more popular with developers. Of the 900 developers we surveyed who say they're using a formal methodology, 45% are using an Agile process, with Scrum being the most popular.

Twenty percent of developers using Agile say it's a key part of the success of their projects, compared with only 12% of those using iterative methodologies and 8% of those doing waterfall development. And only 2% of Agile developers feel that their methodology creates significant busywork, compared with 27% of developers doing waterfall development.

Embracing Agile development isn't easy; developers need to change their thinking, processes, and tools to implement practices such as continuous integration and test-driven development. But a fair number of development shops are well along in their transition, and that means IT managers will need to start thinking about how to apply Agile more broadly and measure the results relative to the other development processes they use.

7. Developers As An Untapped Source Of Innovation

When we asked participants to indicate if several statements described them personally, one statistic jumped out: Sixty percent of respondents don't view their craft as a 9-to-5 job — something that they leave at behind at the end of the workday. These developers continue to apply their development skills outside their jobs on side projects and in other businesses and organizations they work with.

Whether it's writing the next great iPhone application, helping out a church or local charity, or working on a startup, these developers are extending the use of their creative powers beyond the confines of their companies. In particular, we're surprised to see that one in five contributes to at least one open source project. This augers well for the future evolution of open source.

Wise development managers should recognize the creative potential of these entrepreneurial developers and think about how it can be redirected to the benefit of their organizations.

What It All Means

The picture that emerges from our survey is one of developers in transition. Software developers are adopting new technologies or techniques including RIAs, virtualization, and Agile development. They're using and contributing to open source projects, and challenging the conventions that underpin the way enterprise software and tools are built and sold. This transition will accelerate as developer tech populism takes hold and drives the adoption of new development approaches related to cloud computing, scale-out architectures that can accommodate change, and mobile web applications.

Here's the ultimate call to action for people who manage developers: Spend more time understanding what your developers are doing both at work and outside of it, and solicit their ideas about how these technologies could speed up development and cut costs for the organization. Above all, understand how your organization is changing, and that you can't hold back tech populism among your developers — they're going to use of all the technology that's available, whether or not it's approved for use inside the firewall. It's best to embrace this phenomenon and channel it in a positive direction.

Jeffrey Hammond is a research analyst focusing on software development for Forrester Research.

Return to Table of Contents

MonoTouch and the iPhone: The GPS and Compass Functions

Building iPhone and iPad apps using C#

by Bryan Costanich

For many developers, building for Apple's iPhone means writing Objective-C with the XCode IDE. This requires a familiarity with many language techniques that have largely been superseded in modern languages like C# and Java. Enter MonoTouch (http://monotouch.net/), a framework that lets you build iPhone and iPad applications using C#. This frees you from dealing with memory management, pointers, etc., and instead lets you focus on the application functionality.

MonoTouch is a commercial product that is part of the larger, open source Mono project (http://mono-project.com/Main_Page), Novell's open-source implementation of the .NET standard created by Microsoft and published as a set of international standards. In this article, I examine the iPhone in MonoTouch by building an app that shows our coordinates in the world, as well as our direction of travel, and which way the phone is pointing.

The iPhone utilizes several hardware and software technologies that let you accomplish this. The underlying technologies are fairly complex, but Apple abstracts most of the complexity away and gives you an easy way to determine location, and so on.

Under the hood, the iPhone utilizes several location technologies:

- **Wi-Fi Positioning Service (WPS).** Available on all iPhones since an early OS update, WPS uses a database lookup of nearby Wi-Fi access point MAC addresses with known locations. It then computes the location based on a proprietary algorithm. In a densely packed urban center, with lots of Wi-Fi access points, the accuracy of WPS is often 20–30 meters.
- **Global Positioning System (GPS).** Available since the iPhone 3G model, GPS uses a system of satellites orbiting the earth that broadcast time and location signals to the earth. GPS receives these signals and then performs triangulation based on the latency of the signals. GPS is much more reliable than WPS in that it will work anywhere in the world where the GPS receiver has an unobstructed view of at least three satellites. The accuracy of GPS is usually within 20 meters since the year 2000 (when the U.S. government stopped degrading location data for civilian use).
- **Compass.** Available since the iPhone 3G, the compass enables the iPhone to know its orientation to the magnetic poles (magnetic heading). Coupled with its location information, it can also tell you its true heading (based on the known magnetic variation of the location).

Having to work with these directly would be pretty tedious, but fortunately for us, all of this functionality is wrapped up in the CoreLocation API. MonoTouch exposes this API via the *MonoTouch.CoreLocation* namespace, and the most important class is *CLLocationManager*.

The general pattern for working with CoreLocation is:

1. Instantiate a *CLLocationManager* object
2. Configure any settings on *CLLocationManager,* such as the accuracy you want.
3. Assign a delegate to handle the location updates
4. Tell *CLLocationManager* to start feeding you location updates
5. Do something interesting with the location information

Let's do just that, but first, a word about delegates in MonoTouch.

Delegates and Prototypes in MonoTouch

Many of the APIs in CocoaTouch (Apple's UI API for iPhone; see http://developer.apple.com/technology/cocoa.html), and consequently MonoTouch, utilize what Apple calls a "prototype" for call-backs. A prototype is similar to *Interfaces* in .NET. It defines a contract whereby the caller knows that certain methods will exist on a class that conforms to the contract or interface. For example, you might define an interface, say *IPerson,* that has the method *Run(int howFast, int howFar)*. The difference is, with Apple's API, the methods are actually optional. If the runtime doesn't find the *Run* method, it doesn't matter; the runtime just won't call it. In MonoTouch, if you want to the runtime to find your method that conforms to their prototype, you "register" it by decorating it with an *Export* attribute:

```
[Export ("Run")]
public void DoSomeRunning(int howFastToRun, int howFarToRun) {... }
```

MonoTouch tries to make this a bit easier by providing strongly typed delegate base classes that already have these methods and their respective *Export* attributes on them. Notice that our method and parameters didn't have the same naming. That's okay, when you register the method all that matters is the data types passed in and returned.

MonoTouch Delegates

You don't have to use strongly typed delegates that MonoTouch gives you. You can write your own custom delegate classes and the decorate them with the **Export** attribute on methods you want to expose to handle call-backs. To do this, simply create your class, mark your methods using the **Export** attribute, and then set your custom class to the **WeakDelegate** property of whatever class expects a delegate. For more information, see monotouch.net/Documentation/API_Design.

To use them, we simply write a class that derives from the appropriate base delegate class, override the methods that we care about, and assign our class to the delegate property of whatever class that needs it.

In the case of *CoreLocation*, this means deriving from the *CLLocationManagerDelegate* base class, and overriding the methods *UpdatedLocation* and *UpdatedHeading*.

Now that we've gotten all that out of the way, let's actually use the CoreLocation API.

A Sample Application

Let's create a MonoTouch application that uses the CoreLocation API. (The complete source and related files for this app are available at i.cmpnet.com/ddj/images/article/2010/code/Example_CoreLocation.zip).

First, open up MonoTouch; it should look something like Figure 1.

Next, create a new iPhone MonoTouch project as in Figure 2. To do this, select "File : New Solution":

Choose "iPhone MonoTouch Project" from the C# templates. Let's name our solution "Example_CoreLocation": see Figure 3.

Figure 1

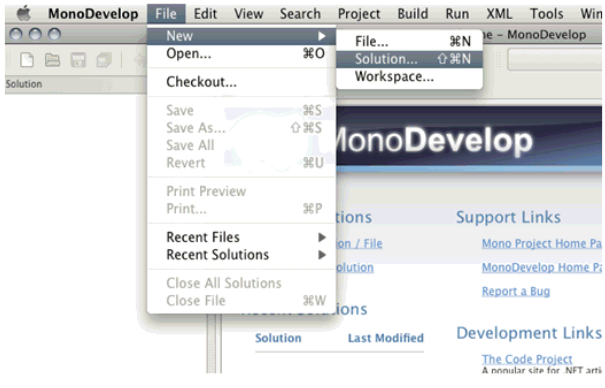


Figure 2

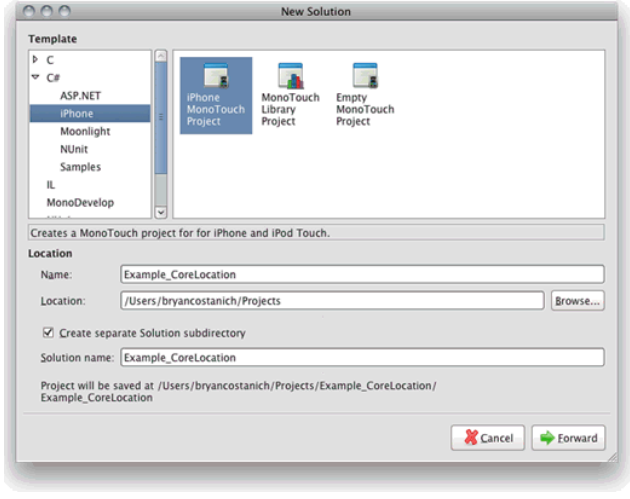


Figure 3

Click "OK" on the Project Features dialogue, as they're not relevant to what we're doing: See Figure 4.

Our project should now look something like Figure 5.

Open the MainWindow.xib file in Interface Builder by double-clicking on it in the Solution Explorer window, you should get something that looks like Figure 6.

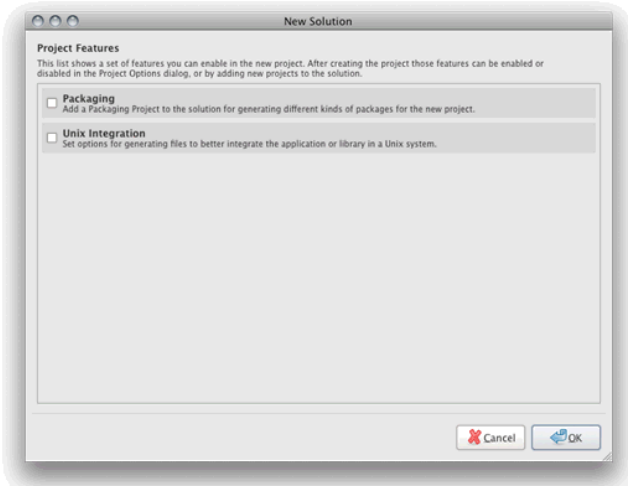


Figure 4

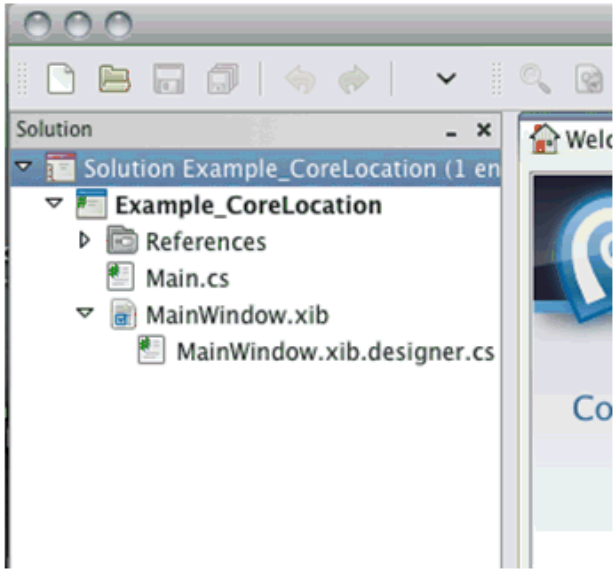


Figure 5

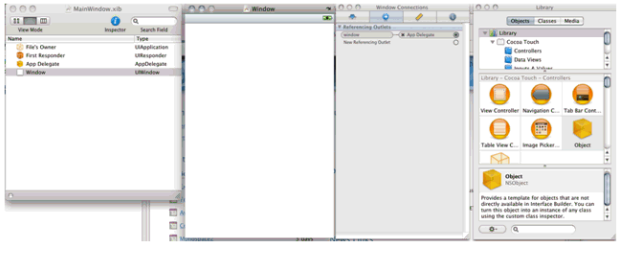


Figure 6

Drag some *UILabels* over to the Window designer and then double-click them to edit their initial text, until you have something that looks like Figure 7.

In the Attributes tab of the Inspector, you can adjust the font and appearance of your labels if you want: see Figure 8.

Next, we need to add Outlets to our *AppDelegate* class for each of the labels that we'll write to. When you add elements to your document in Interface Builder, they're not available to your code until you add Outlets.

In our case, for the simplicity of our example, all our code is going to go into the *AppDelegate* class. Thus we need to add Outlets to it so that our labels are available to it.

To do this, select the Classes Tab in the top of the Library Window, and then select the *AppDelegate* class: see Figure 9.

Make sure the *AppDelegate* class is selected, otherwise your Outlets will get created somewhere else and you'll get build errors when you try to access them.

Next, we're going to add the Outlets by selecting the "Outlets" tab in the lower half of the window and clicking the "+" button.

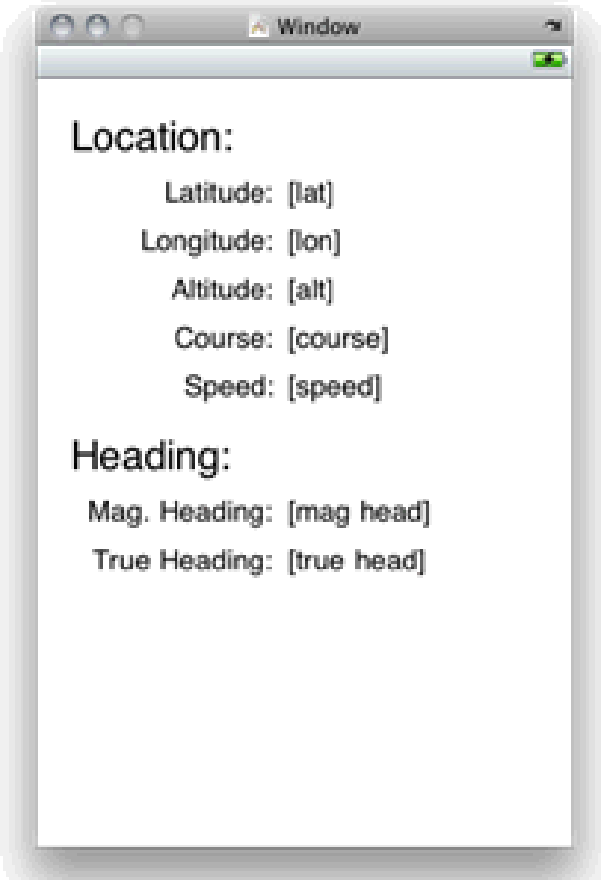


Figure 7

When you click the "+" a new outlet is created and it allows you to edit the name of the Outlet. Let's add the following Outlets:

- txtAltitude
- txtLatitude
- txtLongitude
- txtCourse
- txtSpeed
- txtMagneticHeading
- txtTrueHeading

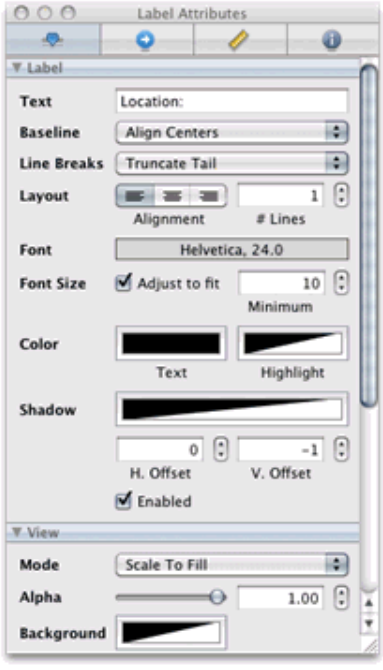


Figure 8

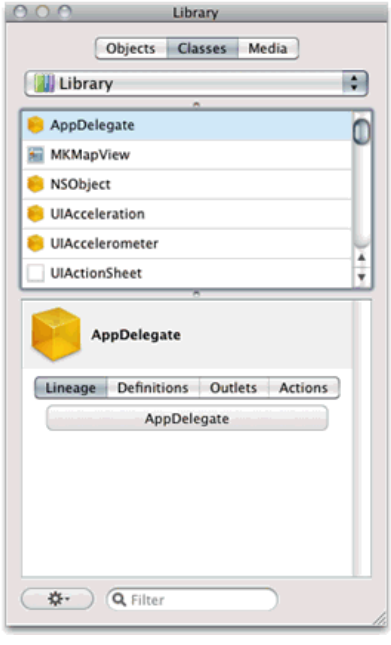


Figure 9

Your Outlets should now look like Figure 10.

Now that we have the Outlets, we actually have to wire them up to their respective controls. To do this, select your App Delegate in the Document Window (see Figure 11).

And then select the Connections tab in the Inspector window (see Figure 12).

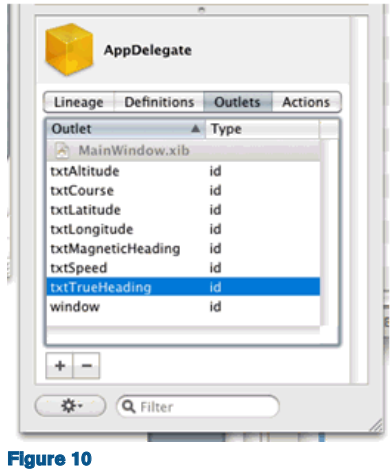


Figure 10

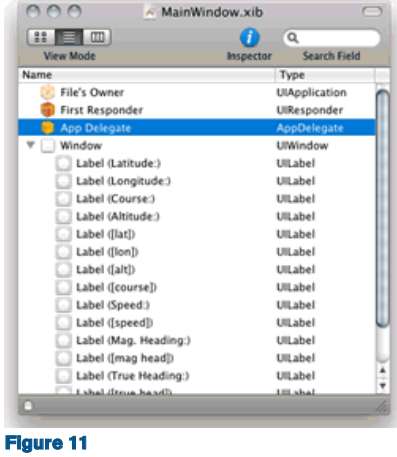


Figure 11

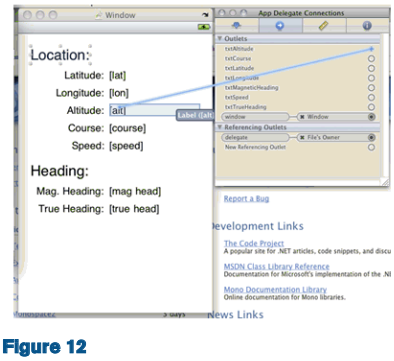


Figure 12

You can see all the Outlets we created earlier. To wire them up, drag from the little circle over to the control on the Document Designer Window, or the Document Window that you want to wire the Outlet up to; see Figure 13.

Continue this until all the outlets are wired up to their appropriate controls; see Figure 14.

Let's run this and make sure it looks right. Save and go back over to MonoDevelop and in the menu, choose

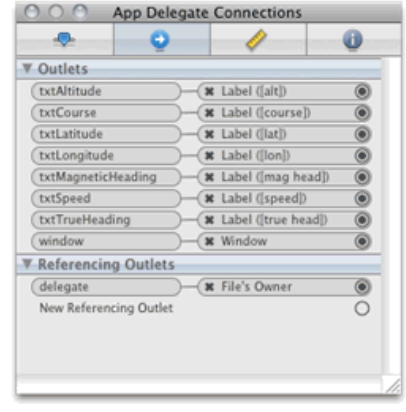


Figure 13

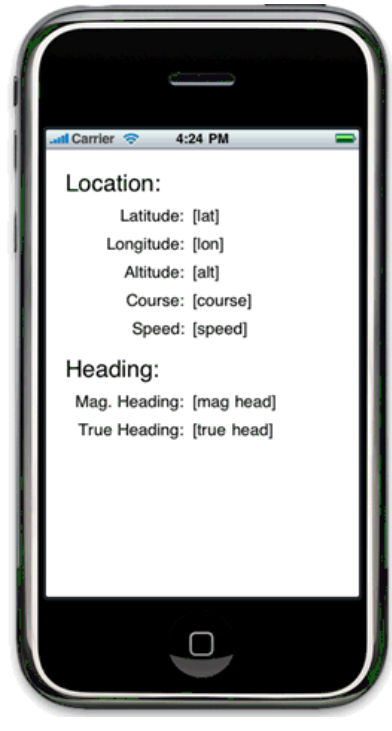


Figure 14

"Run" and then click "Debug." You should have something like Figure 15.

Now that we have our interface created, let's actually do some work.

Open the Main.cs file, and in the *AppDelegate* class, add the following declarations to the class:

```
//—— declare vars
CLLocationManager _iPhoneLocationManager = null;
LocationDelegate _locationDelegate = null;
```

The first declaration is our *CLLocationManager* object. This class is the main point of use for the CoreLocation API.

The second declaration is our *LocationDelegate*. As mentioned before, the *LocationDelegate* is the strongly typed delegate class that will handle location updates.

Next, create the custom *LocationDelegate*. To do this, add the following *LocationDelegate* code to the *AppDelegate* class:

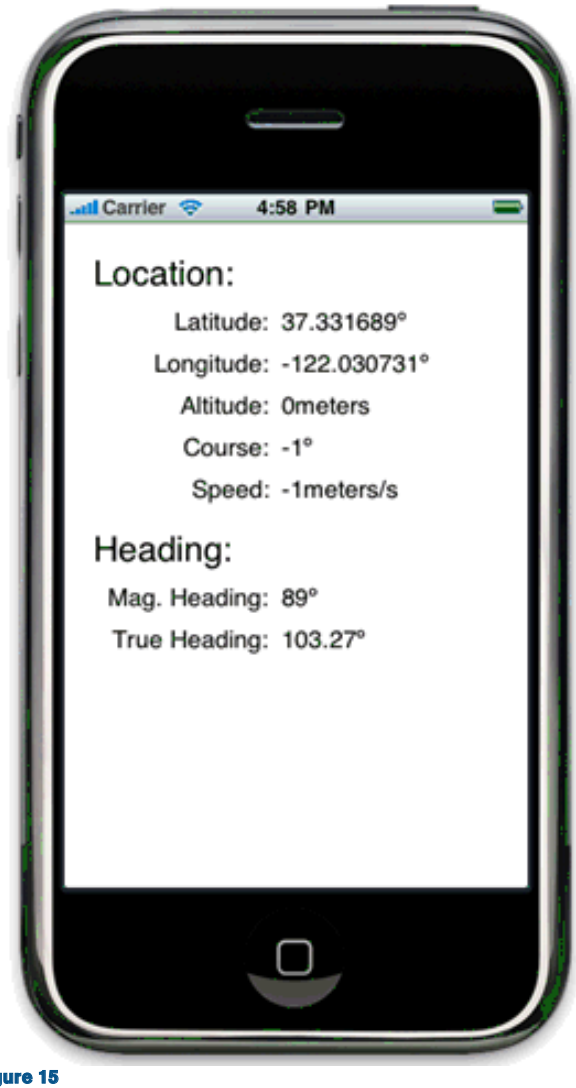


Figure 15

```
//=====================================
public class LocationDelegate : CLLocationManagerDelegate
{
 AppDelegate _app;

 public LocationDelegate (AppDelegate app) : base()
 {
  this._app = app;
 }

 //================================
 public override void UpdatedLocation (CLLocationManager manager
  , CLLocation newLocation, CLLocation oldLocation)
 {
  this._app.txtAltitude.Text = newLocation.Altitude.ToString () +
   "meters";
  this._app.txtLongitude.Text =
   newLocation.Coordinate.Longitude.ToString () + "";
  this._app.txtLatitude.Text = newLocation.Coordinate.Latitude.ToString
    () + "";
  this._app.txtCourse.Text = newLocation.Course.ToString () + "";
  this._app.txtSpeed.Text = newLocation.Speed.ToString () + "meters/s";

 }
 //================================

 //================================
 public override void UpdatedHeading (CLLocationManager manager,
   CLHeading newHeading)
 {
  this._app.txtMagneticHeading.Text =
   newHeading.MagneticHeading.ToString () + "";
  this._app.txtTrueHeading.Text = newHeading.TrueHeading.ToString () +
   "";
 }
 //================================
}
//=====================================
```

Let's take a quick look at this class. It has two methods that we override, *UpdateLocation* and *UpdateHeading*.

The *UpdateLocation* method is called when new location information is received, and from it we get things like location coordinates, altitude, and if we're moving, a course and heading.

The *UpdateHeading* method is called when new heading information is received (which comes from the compass), and from it we get our magnetic and true headings.

The class constructor takes an *AppDelegate* parameter, so that we can reference our outlets within our *LocationDelegate*. This is mostly for example purposes. You could instead raise an event in here and wrap your delegate up into another class if you wanted a more ".NETie" pattern.

Now that we have our delegate class, let's add the following code to our *FinishedLaunching* method, before it returns out:

```
//—— initialize our location manager and callback handler
this._iPhoneLocationManager = new CLLocationManager ();
this._locationDelegate = new LocationDelegate (this);
this._iPhoneLocationManager.Delegate = this._locationDelegate;

//—— start updating our location, et. al.
this._iPhoneLocationManager.StartUpdatingLocation ();
this._iPhoneLocationManager.StartUpdatingHeading ();
```

And then add the following using directives to the top of the Main.cs file:

```
using MonoTouch.CoreLocation;
```

This code initializes our *CLLocationManager* and *LocationDelegate* objects, and then assigns our *LocationDelegate* to the *CLLocationManager*. Next, it tells the *CLLocationManager* to begin updating our location and heading info.

Our entire Main.cs file should now look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using MonoTouch.Foundation;
using MonoTouch.UIKit;
using MonoTouch.CoreLocation;

namespace Example_CoreLocation
{
 public class Application
 {
  static void Main (string[] args)
  {
   UIApplication.Main (args);
  }
 }

 // The name AppDelegate is referenced in the MainWindow.xib file.
 public partial class AppDelegate : UIApplicationDelegate
 {
  //---- declare vars
  CLLocationManager _iPhoneLocationManager = null;
  LocationDelegate _locationDelegate = null;

  // This method is invoked when the application has loaded its UI
and its ready to run
  public override bool FinishedLaunching (UIApplication app,
NSDictionary options)
  {
  // If you have defined a view, add it here:
  // window.AddSubview (navigationController.View);
    window.MakeKeyAndVisible ();

//---- initialize our location manager and callback handler
   this._iPhoneLocationManager = new CLLocationManager ();
 this._locationDelegate = new LocationDelegate (this);
   this._iPhoneLocationManager.Delegate = this._locationDelegate;

 //---- start updating our location, et. al.
   this._iPhoneLocationManager.StartUpdatingLocation ();
 this._iPhoneLocationManager.StartUpdatingHeading ();
   return true;
  }

 // This method is required in iPhoneOS 3.0
  public override void OnActivated (UIApplication application)
  {
  }

 //===============================
  public class LocationDelegate : CLLocationManagerDelegate
  {
   AppDelegate _app;
   public LocationDelegate (AppDelegate app) : base()
   {
       this._app = app;
   }

 //===============================
  public override void UpdatedLocation (CLLocationManager manager
    , CLLocation newLocation, CLLocation oldLocation)
  {
  this._app.txtAltitude.Text = newLocation.Altitude.ToString () +
"meters";
  this._app.txtLongitude.Text =
newLocation.Coordinate.Longitude.ToString () + "";
  this._app.txtLatitude.Text =
newLocation.Coordinate.Latitude.ToString () + "";
  this._app.txtCourse.Text = newLocation.Course.ToString () + "";
  this._app.txtSpeed.Text = newLocation.Speed.ToString () +
"meters/s";

  }
   //===============================

   //===============================
 public override void UpdatedHeading (CLLocationManager manager,
CLHeading newHeading)
 {
 this._app.txtMagneticHeading.Text =
newHeading.MagneticHeading.ToString () + "";
  this._app.txtTrueHeading.Text = newHeading.TrueHeading.ToString ()
+ "";
 }
   //===============================
 }
 //===============================
 }
}
```

Now, if we run this in the simulator we should get something like Figure 16.

It takes a second or two for the *CLLocationManager* to acquire location and heading information and then it should get updated to the screen.

Now, because the simulator doesn't actually have access to location and heading information, it gives you simulated data, which returns the lat/long of Apple's headquarters in Cupertino, California, and made up data for the rest.

If you were to deploy this to an actual device however, you would get actual location information.

Congratulations, you now know how to use the CoreLocation API in MonoTouch! Now that we have this working, let's look at a few more considerations when using it.

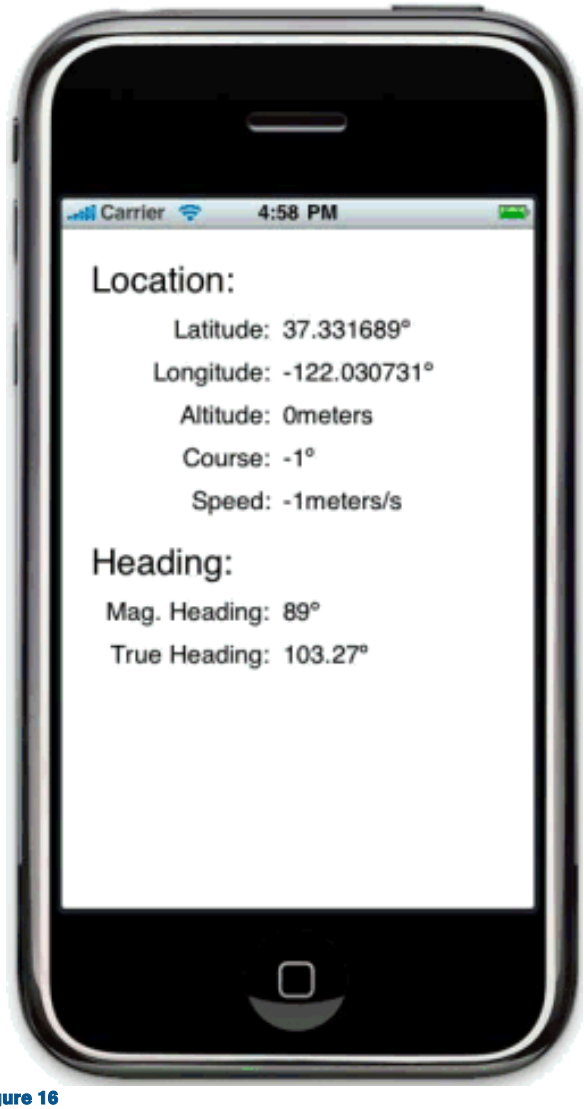


Figure 16

Other Features and Considerations

Battery Drain. Using the CoreLocation API causes considerable battery drain on an iPhone. Because of this, it's important to use it as little as possible. There are several ways to reduce usage and therefore battery drain:

- Only Check When Needed. By calling *StartUpdatingLocation* and *StartUpdatingHeading* only when you need the information, and calling *StopUpdatingLocation* and *StopUpdatingHeading* when you're finished with the data, the iPhone can actually turn off the underlying hardware that makes these calls. This makes the battery last significantly longer.
- Set Your Accuracy. By default, the location data given by the iPhone will strive to be as accurate as possible. This may mean trying to poll more satellites, or resolving more Wi-Fi access points, etc. This also can cause significant battery drain. *CLLocationManager* has a property called *DesiredAccuracy* that can be set to your desired accuracy of location data. It expects a setting in meters, so setting it to 1000 will give it kilometer level accuracy. Setting to –1 will force it to be as accurate as possible.

Update Threshold. By default, location and heading data will be updated constantly. You can set a threshold level for how often you want the location and heading to be updated. *CLLocationManager* exposes the *DistanceFilter* and *HeadingFilter* properties, to set the location and heading update threshold, respectively.

- *CLLocationManager.DistanceFilter*. The value, in meters, of how far the device has to move laterally before a location update is sent. A value of *–1* will cause it to update continuously.
- *CLLocationManager.HeadingFilter*. The value, in degrees, of how far the device has to rotate before a heading update is sent. A value of *–1* will cause it to update continuously.

Capabilities. The iPhone user can turn off location services altogether from the settings panel on the iPhone. Some users do this for privacy or battery life reasons. Additionally, the compass was introduced in the iPhone 3G model. The *CLLocationManager* exposes two properties, *LocationServicesEnabled* and *HeadingAvailable,* to give you information:

- *CLLocationManager.LocationServicesEnabled*. If false, the user has this turned off, and if your application attempts to access location information via *StartUpdatingLocation*, the user will be given a prompt to allow the application to use location services.
- *CLLocationManager.HeadingAvailable*. This is false on the versions of the iPhone that do not have a compass built in. If it's false, then you won't be able to get any meaningful heading data.

Return to Table of Contents

C Snippet #2

Rounding floating-point values

by Bob Stout

Q: HOW DO I... round floating-point values in C?

A: This header file Snippet contains macros to round floating-point values in two different manners. The *iround()* macro rounds the value to the nearest integer. The *fround()* macro rounds the value to a specified number of decimal points.

```
/* ** rounding macros by Dave Knapp */
/* round to integer */
#define iround(x) floor((x)+0.5)
/* round number n to d decimal points */
#define fround(n,d) (floor((n)/pow(.1,(d))+.5)*pow(.1,(d)))
```

Update to C Snippet #2

```
/************************************************/
/**** round.h ****/
/*
** rounding macros by Dave Knapp, Thad Smith, Jon Strayer, & Bob Stout
*/

#ifndef ROUND__H
#define ROUND__H

#include <math.h>

#if defined(__cplusplus) && __cplusplus

/*
** Safe C++ inline versions
*/

/* round to integer */

template<class T> inline int iround(T x)
{
   // This needs to be modified if passing long doubles return (int)floor(double(x) + 0.5);
}

/* round number n to d decimal points */

inline double fround(double n, unsigned d)
{
 return floor(n * pow(10., d) + .5) / pow(10., d);
}

#else // C

/*
** NOTE: These C macro versions are unsafe since arguments are referenced
** more than once. ** ** Avoid using these with expression arguments to be safe.
*/

/*
** round to integer
*/

#define iround(x) ((int)floor((double)(x) + 0.5))

/*
** round number n to d decimal points
*/

#define fround(n,d) (floor((n)*pow(10.,(d))+.5)/pow(10.,(d)))

#endif // C/C++

#endif /* ROUND__H */

/************************************************/
/**** rndtst.c ****/
```

```
/*
**  Demo/test main() function for ROUND.H
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "round.h"

int main(void)
{
       float fx[] = {-5/3.0, -4/3.0, -2/3.0, -1/3.0, 1/3.0, 2/3.0, 4/3.0, 5/3.0};
       double dx[] = {-5/3.0, -4/3.0, -2/3.0, -1/3.0, 1/3.0, 2/3.0, 4/3.0, 5/3.0};
       double phi = (1 + sqrt(5.0)) / 2;
       int i;

       for (i = 0; i < 8; ++i)
              printf("%d: float: iround(%.6g) = %d\n", i+1, fx[i], iround(fx[i]));
       for (i = 0; i < 8; ++i)
              printf("%d: double: iround(%.12g) = %d\n", i+1, dx[i], iround(dx[i]));

       for (i = 1; i < 5; i++)
       {
              int x = i * 2;

              printf("fround(phi, %d) = %.9g\n", x, fround(phi, x));
       }
       return EXIT_SUCCESS;
}



/*********************************************/
/**** rndtst.out ****/

1: float: iround(-1.66667) = -2
2: float: iround(-1.33333) = -1
3: float: iround(-0.666667) = -1
4: float: iround(-0.333333) = 0
5: float: iround(0.333333) = 0
6: float: iround(0.666667) = 1
7: float: iround(1.33333) = 1
8: float: iround(1.66667) = 2
1: double: iround(-1.66666666667) = -2
2: double: iround(-1.33333333333) = -1
3: double: iround(-0.666666666667) = -1
4: double: iround(-0.333333333333) = 0
5: double: iround(0.333333333333) = 0
6: double: iround(0.666666666667) = 1
7: double: iround(1.33333333333) = 1
8: double: iround(1.66666666667) = 2
fround(phi, 2) = 1.62
fround(phi, 4) = 1.618
fround(phi, 6) = 1.618034
```

More C Snippets

- Determining a file's size (http://www.drdobbs.com/cpp/219100141)
- Rounding floating-point values (http://www.drdobbs.com/cpp/219200409)
- Sorting an array of strings (http://www.drdobbs.com/cpp/219500313)
- Computing the wind chill factor (http://www.drdobbs.com/cpp/219700315)
- Timers and default actions (http://www.drdobbs.com/cpp/220001077)
- Getting a string of arbitrary length (http://www.drdobbs.com/cpp/219200409)

All the code in C Snippets is either public domain or freeware and may therefore freely be used by the C programming community without restrictions. In most cases, if the original author is someone other than myself, he or she will be identified. Thanks to all who have contributed to this collection over the years. I hope *Dr. Dobb's* readers will find these useful.

—Bob Stout

rbs@snippets.org

Return to Table of Contents

JDBC Fast Connection Failover with Oracle RAC

Configuring and testing with Tomcat and the Spring framework

by Michael Pilone

It wasn't too long ago when it would be common to see a notice on your favorite website that it is "Currently Down for Scheduled Maintenance". However, with the push toward all data in the cloud and more web "applications" over the past few years, users are demanding constant availability and access to data and resources. Just look at the Internet outcries when Twitter or Google gmail goes down for more than a few minutes and you can understand the importance of building highly available systems. In this article, I present examples of how to configure and test Java web service and enterprise applications to leverage Oracle Real Application Clusters (RAC), Oracle Universal Connection Pool (UCP), and Fast Connection Failover (FCF) to provide high availability.

Oracle RAC (http://www.oracle.com/technology/products/database/clustering/index.html) is a clustering solution that allows you to present multiple interconnected Oracle databases as one server to the end user or applications. Oracle RAC uses Oracle Clusterware (http://www.oracle.com/technology/products/database/clusterware/index.html) to link together multiple servers to create one large single database. Normally, there is a one-to-one relationship of an Oracle database to an instance when running on a single server; however in a RAC environment, there is a one-to-many relationship where a single database is simultaneously hosted on multiple instances. The obvious advantage is that any one of the instances could fail while the database remains available on the other instances in the cluster. There are other advantages to RAC, including performance scaling and workload distribution, however a full discussion of RAC is beyond the scope of this article. For more information, refer to the Oracle Real Application Clusters Administration and Deployment Guide (http://download.oracle.com/docs/cd/E11882_01/rac.112/e10718/toc.htm).

With Oracle RAC in place, database client applications can leverage a number of high availability features including:

- Fast Connection Failover (FCF): Allows a client application to be immediately notified of a planned or unplanned database service outage by subscribing to Fast Application Notification (FAN) events.
- Runtime Connection Load-Balancing: Uses the Oracle RAC Load Balancing Advisory events to distribute work appropriately across the cluster nodes and to quickly react to changes in cluster configuration, overworked nodes, or hangs.
- Connection Affinity (11g recommended/required): Routes connections to the same database instance based on previous connections to an instance to limit performance impacts of switching between instances. RAC supports web session and transaction-based affinity for different client scenarios.

These features are enabled in a Java application through the use of the Oracle Universal Connection Pool or UCP (http://download-west.oracle.com/docs/cd/B28359_01/java.111/e10788/title.htm) and, other than configuration, require no special support code.

Oracle Server Environment

Before any high availability (HA) work can be done on the client side, a few key Oracle server components must be in place. This article is not meant to be a guide to installing and configuring Oracle RAC (there are many books on the subject) but there are a few key features that must be enabled by your database administrator (DBA).

Obviously, the RAC environment must have at least two nodes available for testing. Your DBA will give you the hostnames for these nodes, but you can assume that they will be something like "sales1node1" and "sales1node2". These are the two hosts that your Java client will connect to and therefore need to be running the standard Oracle listener process. Your DBA should be able to tell you the port number that the listeners are bound to (usually 1521).

Oracle databases are represented to clients as a service. A single database can have multiple services associated with it, which are defined by service names, such as sales.foo.com or finance.foo.com. The service names should not be confused with the listener host names even though a service name may look like a host or domain name. In a RAC environment, different instances in the cluster can offer the same service to clients, which allows for failover if one instance goes down, distribution of load based on the capabilities of the hardware, and service migration for planned downtime. Therefore, it is important that the service name and not the instance name (i.e., Oracle System Identifier (SID)) is used when configuring the client. Figure 1 shows how two different instances (sales 1 and sales 2) can both offer the same service (sales.foo.com) while only one instance (sales 2) offers a second service (finance.foo.com).

The Oracle RAC Load Balancing Advisory must be configured with service-level goals for each service for which load balancing is enabled. Enabling the advisory allows the RAC nodes to advise clients on how to distribute work across the cluster to ensure optimal use of resources and to avoid slow, hung, or down nodes. The Oracle Real Application Clusters Administration and Deployment Guide contains the details for configuring the advisory but the basic process is to execute a SQL statement that associates a load-balancing goal with a service such as:

```
EXECUTE DBMS_SERVICE.MODIFY_SERVICE (service_name => 'sales' -,
goal => DBMS_SERVICE.GOAL_THROUGHPUT -, clb_goal =>
DBMS_SERVICE.CLB_GOAL_SHORT);
```

The final server side piece is the setup of the Oracle Notification Service (ONS), which propagates Fast Application Notification (FAN) events between the Oracle RAC database and the JDBC connection pool. On the client side, ONS can be configured in a local or remote configuration with remote being preferred because it eliminates the need for a local ONS server process on the client machine. To support remote connections to the ONS server, the DBA must modify the ons.config file on the database servers to set the remoteport and the list of ONS nodes. Your DBA should be able to tell you the remote port to use when configuring remote ONS support on the client side.

Now that the Oracle server side setup is complete and you obtained all the needed information from a DBA, you can test the FCF and ONS setups and configure the JDBC clients.

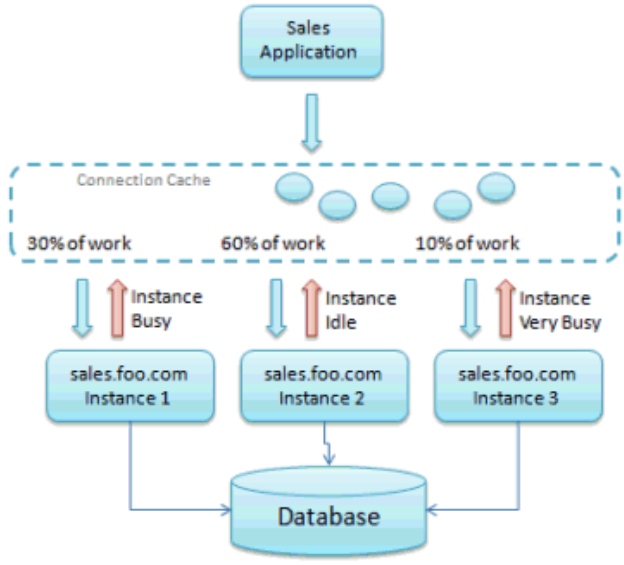


Figure 1: Oracle RAC attempts to distribute the load by sending Runtime Load-Balance Advisory FAN events to clients. The UCP will rebalance the pooled connections based on the advisory events.

JDBC Client Environment

To begin the client-side testing and configuration, you need to gather some configuration information and required libraries. From the previous Oracle environment setup, you should already know:

- The hosts and ports for the RAC node listeners
- The service name(s) of the target database
- The hosts and ports for the ONS servers

To use the ONS service on the client side, Oracle's ONS library (ons.jar) must be obtained from Oracle; available directly from the Oracle JDBC Driver website (www.oracle.com/technology/software/tech/java/sqlj_jdbc/htdocs/jdbc_112010.html) or it can be copied from a thick client or server installation. The library implements all the client-server communications to subscribe to the ONS servers and receive FAN events.

Most Tomcat- or Spring-based applications currently use the Apache Database Connection Pooling (DBCP) library to create and manage a connection pool. While robust and well supported, the DBCP library is a generic connection pool that is not aware of vendor-specific extensions such as FCF or ONS. Oracle therefore provides the Universal Connection Pool (UCP) which serves as a generic connection pool that implements the standard *javax.sql.DataSource* interface while supporting the Oracle-specific features for HA. Prior to Oracle 11g, Oracle recommended using Implicit Connection Caching (ICC) in the *oracle.jdbc.pool.OracleDataSource*; however, according to the Oracle 11g JDBC documentation:

In this release the Oracle Implicit Connection Cache feature is deprecated. Users are strongly encouraged to use the new Universal Connection Pool instead. The UCP has all of the features of the ICC, plus much more. The UCP is available in a separate jar file, ucp.jar.

Therefore, any new application should use UCP rather than starting with ICC. The 11g JDBC drivers are backward compatible with 10g databases so even in instances where the database is only 10g, the client should consider using UCP with an 11g driver. The UCP library (ucp.jar) and the 11g JDBC driver (ojdbc6.jar) can be downloaded directly from Oracle's website (http://www.oracle.com/technology/index.html).

The final component for client side setup is the JDBC URL used to connect to the server. When working with a single database instance, the standard thin JDBC driver URL took the form *jdbc:oracle:thin:@<host>:<port>:<SID>*, which specifies the single host, port, and SID of the instance to which to connect. To support HA, the client needs to be able to talk to multiple RAC nodes in the cluster in the event that any node goes down. To support this, the thin JDBC driver allows for the use of a connect descriptor similar to what is found in the tnsnames.ora configuration file on the Oracle Server. Your DBA may be able to provide you with a connect descriptor, you can use the *tnsping* command on the server, or you can write one from scratch such as:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=ON)(AD
DRESS=(PROTOCOL=TCP)(HOST=sales1node1.foo.com)(PORT=1521))(ADDRES
S=(PROTOCOL=TCP)(HOST=sales1node2.foo.com)
(PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=sales.foo.com)))
```

The key is to include multiple instance host names in the descriptor so the connection pool can contact both instances at runtime.

Testing the Setup

Testing the setup with a simple test client can verify that all the configuration pieces are in place for both the Oracle server environment and the JDBC client environment before trying to configure a complete application or container. Direct access to all of the hosts and service ports from the client machine is required because the listener may redirect the client to another host based on load and availability. Access can be checked using the *telnet* command on the client machine and verifying that a server answers for each request:

telnet <ons host 1><port>
telnet <ons host 2><port>
telnet <listener host 1><port>
telnet <listener host 2> <port>

With the ports accessible, you can run a simple test client to verify the configuration information and to get a better understanding of how ONS, FCF, and UCP are working together. Listing 1 contains a JDBC client that performs some basic JDBC interactions and prints the UCP log information.

Load Balancing In the Connect Descriptor

Although the connect descriptor presented here contains the options LOAD_BALANCE=ON, the UCP, FCF, and ONS components must be configured correctly to use the runtime load-balance advisory. The LOAD_BALANCE option simply tells the client to randomly choose an available listener from the address list when making new connections. However the runtime load-balance advisory will take precedence over this simple client-side load-balancing approach if advisory events are received.

```
LISTING ONE
import static java.lang.String.*;

import java.io.*;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.sql.*;
import java.util.*;
import java.util.logging.*;

import oracle.jdbc.pool.OracleDataSource;
import oracle.ucp.admin.UniversalConnectionPoolManager;
import oracle.ucp.admin.UniversalConnectionPoolManagerImpl;
import oracle.ucp.jdbc.*;

/**
 * <p>
 * A test class that provides some simple JDBC functionality to test the
Oracle
 * Universal Connection Pool, Fast Connection Failover, and the Oracle
 * Notification Service. You may need to modify the DB_* constants to
configure
 * the test driver for your DB.
 * </p>
 * <p>
 * To run the test, execute:<br/>
 * java -cp ons.jar:ucp.jar:ojdbc6.jar:. UcpTest
 * </p>
 * <p>
 * The commands include:
 * <ul>
 * <li>exit: close all connections and exit</li>
 * <li>get: get a connection from the pool</li>
 * <li>print: print current connections and connection distribution</li>
 * <li>stats: print some basic connection pool stats</li>
 * <li>commit: commit the current transaction on a connection</li>
 * <li>close: close (release) a connection and return it to the pool</li>
 * </ul>
 * </p>
 *
 * @author mpilone
 */
public class UcpTest
{
  /**
   * The user to connect as. This user must have select access to
v$instance.
   */
  private static final String DB_USER = "bsmith";

  /**
   * The password for the user.
   */
  private static final String DB_PASSWORD = "secret";

  /**
   * The user to connect as for gather statistics information such as
connection
   * distribution. This user must have select access to gv$session. It is
   * recommended that you don't use the same user as DB_USER so that the
   * connection distribution count is accurate. It is safe to use
   * "sys as sysdba" for this user if desired.
   */
```

```
  private static final String DB_STATS_USER = "sys as sysdba";

  /**
   * The password for the stats user.
   */
  private static final String DB_STATS_PASSWORD = "dba_secret";
  /**
   * The URL to the DB using a connection descriptor.
   */
  private static final String DB_URL =
      "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=ON)"
          + "(ADDRESS=(PROTOCOL=TCP)"
          + "(HOST=sales1node1.foo.com)(PORT=1521))"
          + "(ADDRESS=(PROTOCOL=TCP)"
          + "(HOST=sales1node2.foo.com)(PORT=1521)))"
          + "(CONNECT_DATA=(SERVICE_NAME=sales.foo.com)))";

  /**
   * The ONS node configuration which uses the remoteport of the ONS
remote
   * nodes.
   */
  private static final String ONS_CONFIG =
      "nodes=sales1node1.foo.com:6151,sales1node2.foo.com:6151";

  /**
   * Main entry point to the application. No arguments are supported.
   *
   * @param args
   *           the command line arguments
   */
  public static void main(String[] args) throws Exception
  {
    new UcpTest().go();
  }
  /**
   * Starts the test client and prompts for input.
   *
   * @throws Exception
   */
  private void go() throws Exception
  {
    // The list of active connections
    List<Connection> conns = new ArrayList<Connection>();

    // Create the universal connection pool
    PoolDataSource ds = initConnectionPool();

    // Setup logging so we can see the FAN events coming from ONS
    Handler fh = new FileHandler("ucptest.log");
    fh.setLevel(Level.FINE);
    fh.setFormatter(new SimpleFormatter());

    UniversalConnectionPoolManager mgr =

UniversalConnectionPoolManagerImpl.getUniversalConnectionPoolManager();
    mgr.setLogLevel(Level.FINE);
    Logger log = Logger.getLogger("oracle.ucp");
    log.setLevel(Level.FINE);
    log.addHandler(fh);
    log.setUseParentHandlers(false);

    BufferedReader sysin = new BufferedReader(new
InputStreamReader(System.in));

    while (true)
    {
      System.out
          .println("\nWaiting for command [exit, get, print, stats,
commit, close]:");
      String command = sysin.readLine();
      System.out.println();

      // Handle the specific command
      if (command.equalsIgnoreCase("exit"))
      {
        for (Connection conn : conns)
        {
          conn.close();
        }
        System.exit(0);
      }
      else if (command.equalsIgnoreCase("get"))
      {
        Connection conn = ds.getConnection();
        conn.setAutoCommit(false);
        conns.add(conn);
        printConnections(conns);
      }
      else if (command.equalsIgnoreCase("print"))
      {
        printConnections(conns);
      }
      else if (command.equalsIgnoreCase("stats"))
      {
        JDBCConnectionPoolStatistics stats = ds.getStatistics();

        if (stats != null)
        {
          printStat("AbandonedConnectionsCount", stats);
          printStat("AvailableConnectionsCount", stats);
          printStat("AverageBorrowedConnectionsCount", stats);
          printStat("AverageConnectionWaitTime", stats);
          printStat("BorrowedConnectionsCount", stats);
          printStat("ConnectionsClosedCount", stats);
          printStat("ConnectionsCreatedCount", stats);
          printStat("RemainingPoolCapacityCount", stats);
          printStat("TotalConnectionsCount", stats);
        }
        printConnectionDistribution();
      }
      else if (command.equalsIgnoreCase("commit"))
      {
        int index = readInt("Connection number: ", sysin);
        if (index >= 0 && index < conns.size())
        {
          Connection conn = conns.get(index);
          conn.commit();
        }
      }
      else if (command.equalsIgnoreCase("close"))
      {
        int index = readInt("Connection number: ", sysin);
        if (index >= 0 && index < conns.size())
        {
          Connection conn = conns.remove(index);
          conn.close();
        }
        printConnections(conns);
      }
    }
  }

  /**
   * Creates the connection to the DB as the statistics user.
   *
   * @return a new connection to the database
   */
  private Connection initStatsConnection() throws SQLException
  {
    OracleDataSource ds = new OracleDataSource();
    ds.setUser(DB_STATS_USER);
    ds.setPassword(DB_STATS_PASSWORD);
    ds.setURL(DB_URL);

    return ds.getConnection();
  }

  /**
   * Creates the UCP with FCF enabled.
   *
   * @return the UCP
   * @throws SQLException
   */
  private PoolDataSource initConnectionPool() throws SQLException
  {
    PoolDataSource ds = PoolDataSourceFactory.getPoolDataSource();
    ds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
    ds.setUser(DB_USER);
    ds.setPassword(DB_PASSWORD);
    ds.setURL(DB_URL);
    ds.setMinPoolSize(5);
    ds.setMaxPoolSize(25);
    ds.setInactiveConnectionTimeout(20);
    ds.setConnectionWaitTimeout(20);
    ds.setPropertyCycle(60);
    ds.setFastConnectionFailoverEnabled(true);
    ds.setONSConfiguration(ONS_CONFIG);

    return ds;
  }

  /**
   * Prints a statistic given the name of the statistic. This method uses
   * reflection for simplicity.
   *
   * @param statName
   *           the name of the statistic to print
   * @param stats
   *           the statistics from the connection pool
   * @throws NoSuchMethodException
   * @throws SecurityException
   * @throws InvocationTargetException
   * @throws IllegalAccessException
```

```
    * @throws IllegalArgumentException
    */
  private void printStat(String statName, JDBCConnectionPoolStatistics
stats)
      throws Exception
  {
    Method m = stats.getClass().getMethod("get" + statName);
    Object value = m.invoke(stats);

    System.out.println(format("%-35s%s", statName, value));
  }

  /**
   * Reads an integer from standard input.
   *
   * @param sysin
   *           standard input
   * @return the integer read
   * @throws IOException
   *            if there is an error reading
   */
  private int readInt(String prompt, BufferedReader sysin) throws
IOException
  {
    System.out.print(prompt);
    int index = Integer.valueOf(sysin.readLine());
    return index;
  }

  /**
   * Prints the list of connections and connection distribution.
   *
   * @param conns
   *           the list of active connections
   */
  private void printConnections(List<Connection> conns)
  {
    // Print the total number of connections actively held.
    System.out.println("Total held connections: " + conns.size());

    // Iterate over the connections and execute a query to determine
    // which instance the connection is talking to.
    int index = 0;
    for (Iterator<Connection> iterator = conns.iterator();
iterator.hasNext();)
    {
      Connection conn = iterator.next();
      try
      {
        Statement stmt = conn.createStatement();

        // Get the instance name that is serving this connection.
        ResultSet rs =
            stmt.executeQuery("select instance_name from v$instance");
        if (rs.next())
        {
          System.out.println(format("[%s] Instance name [%s]", index++,
rs
              .getString(1)));
        }

        rs.close();
        stmt.close();
      }
      catch (SQLException ex)
      {
        System.out.println(format("Failed to query connection. "
            + "It will be removed from the list. Error [%s]",
ex.getMessage()));

        iterator.remove();
      }
    }
  }

  /**
   * Prints the distribution of connections across the RAC nodes.
   *
   * @throws SQLException
   */
  private void printConnectionDistribution() throws SQLException
  {
    Connection statsConn = initStatsConnection();

    // Get the distribution of connections across the nodes, including
    // connections just sitting in the cache.
    System.out.println();
    System.out.println("Connection distribution");
    System.out.println("Instance ID    Connection Count");

    Statement stmt = statsConn.createStatement();
    ResultSet rs =
        stmt
            .executeQuery(format("select inst_id, count(1) from
gv$session "
                + "where type='USER' and username = '%s' group by
inst_id",
                DB_USER));

    while (rs.next())
    {
      System.out.println(format("%6s%18s", rs.getString(1),
rs.getString(2)));
    }

    rs.close();
    stmt.close();
    statsConn.close();
  }
}
```

Setting Up the Test Privileges

There are a couple of DBA support items needed before you can run these FCF tests. The JDBC client application requires access to the v$instance and the gv$session tables, which may require special privileges depending on the database user the client is configured to use. Your DBA should be able to setup access to these data dictionaries for your user rather than allowing you to use a more privileged user such as SYS.

With the client, you can watch the ONS events received, obtain connections from the RAC nodes, print information about the active connections, and selectively close connections. The client is launched from the console with the command: *java "cp ons.jar:ucp.jar:ojdbc6.jar:. UcpTest.*

With the client running, you should see events that indicate load-balancing information from the RAC server. This information is published as part of the Runtime Load Balance Advisory that you configured earlier. The detailed information in the events, which won't be visible in the logs, contains a percentage of the load that each instance should be given. This information is used by the connection pool to allocate work and to rebalance idle connections. Figure 2 shows a conceptual view of the load-balance advisory information.

The advisory events in the log will look like this:

```
<DATE> oracle.ucp.jdbc.oracle.ONSRuntimeLBEventHandlerThread run
FINE: Runtime Load Balancing event triggered
```

The first scenario to test is a planned outage where the Oracle service is shutdown gracefully, allowing clients to complete their current transactions, close connections, and migrate to another instance. The UCP will handle all of this assuming it gets the DOWN FAN event from the server. The test case steps are:

1. Get connections until the client has a connection from both RAC nodes
2. Ask the DBA to stop the service using the *srvctl* command on one of the RAC nodes
3. Verify that a DOWN FAN event was received in the logs such as: <DATE> oracle.ucp.jdbc.oracle.ONSDatabaseEventHandlerThread run FINE: event triggered: Service name: search.echo.nasa.gov, Instance

name: rac1db2, Unique name: rac1db, Host name: rac1node2, Status: down, Cardinality: 0, Reason: user, Event type: database/event/service

4. Print the connections to verify that they are all still valid
5. Close the connections (release to the pool) to the node with the stopped service and watch as the UCP immediately closes the connections to the downed instance
6. Get a couple new connections and verify that they all go to the only up instance
7. Ask the DBA to start the downed service using the *srvctl* command
8. Verify that an UP FAN event was received in the logs such as: oracle.ucp.jdbc.oracle.ONSDatabaseEventHandlerThread run FINE: event triggered: Service name: search.echo.nasa.gov, Instance name: rac1db2, Unique name: rac1db, Host name: rac1node2, Status: up, Cardinality: 2, Reason: user, Event type: database/event/service
9. Close a few connections and verify that the UCP rebalances the pooled connections across the nodes based on the load-balance advisory.

In an operational environment, once the service is shutdown the DBA can monitor the active sessions manually or execute the SQLPlus command shutdown NORMAL to shutdown the instance once all of the connections have been released by the UCP.

The second scenario to test is an unplanned outage where the Oracle service is shutdown abnormally such as a machine crash or network failure. The UCP will immediately mark the active connections as no longer valid (preventing a client lag or hang) and will open new connections only to the remaining instance. In this scenario, it is up to the client to handle a *SQLException* raised by the connection. The test case steps are:

1. Get connections until the client has a connection from both RAC nodes

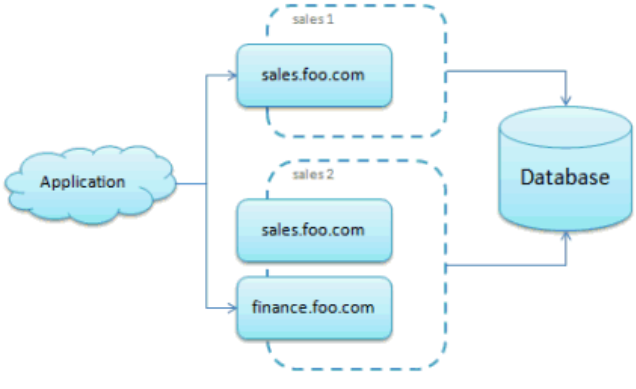


Figure 2: A single Oracle instance can host one or more services. Clients can use any instance that offers the service needed.

2. Ask the DBA to stop an instance using the SQLPlus command shutdown IMMEDIATE
3. Verify that a FAILURE FAN event was received in the logs such as: <DATE> oracle.ucp.jdbc.oracle.ONSDatabaseEventHandlerThread run FINE: event triggered: Service name: search.echo.nasa.gov, Instance name: rac1db2, Unique name: rac1db, Host name: rac1node2, Status: down, Cardinality: 0, Reason: failure, Event type: database/event/service
4. Print the connections to verify that connections to the failed instance are terminated
5. Get a couple new connections and verify that they all go to the only up instance
6. Ask the DBA to start the downed instance and service
7. Verify that an UP FAN event was received in the logs
8. Close a few connections and verify that the UCP rebalances the pooled connections across the nodes based on the load-balance advisory

Obviously, unplanned outages are not preferred, but UCP does attempt to quickly close connections to downed services and redirect new connections to the remaining instances.

At this point, you should be able to verify the configuration and functionality of your RAC setup and see the FAN events in the log of the test client. If the FCF functionality is not working, contact your DBA to verify the configuration and test again before continuing. It is easier to debug configuration problems in this simple scenario than in a larger application or container.

Configuring a Tomcat Data Source

Tomcat data source configuration is done through a *Resource* element in the Context configuration. Tomcat has built-in support for database resources using the Apache DBCP implementation; however, a custom resource factory can be used to setup the Oracle UCP implementation. Oracle makes this configuration easy by providing an implemention of the JNDI *javax.naming.spi.ObjectFactory* interface via the *oracle.ucp.jdbc.PoolDataSourceImpl* class, which allows Tomcat to instantiate and configure the UCP data source.

The attributes used on the *Resource* element will be passed to the *PoolDataSourceImpl* during construction to setup the connection pool. The Javadocs for the *oracle.ucp.jdbc.PoolDataSource* define the various configuration strings as constants for reference. Listing 2 shows a sample configuration. The UCP data source will be available to deployed applications just like a normal Tomcat DBCP configured data source and requires no changes to the application code.

```
LISTING TWO
<Resource name="salesDatasource"
          auth="Container"
          factory="oracle.ucp.jdbc.PoolDataSourceImpl"
```

```
          type="oracle.ucp.jdbc.PoolDataSource"
          description="FCF Sales Datasource"

connectionFactoryClassName="oracle.jdbc.pool.OracleDataSource"
          connectionWaitTimeout="30"
          minPoolSize="5"
          maxPoolSize="25"
          inactiveConnectionTimeout="20"
          timeoutCheckInterval="60"
          fastConnectionFailoverEnabled="true"

onsConfiguration="nodes=sales1node1.foo.com:6151,sales1node2.foo.
com:6151"
          user="bsmith"
          password="secret"

url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=
sales1node1.foo.com)(PORT=1521))(ADDRESS=(PROTOCOL=TCP)(HOST=sale
s1node2.foo.com)
(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=sales.foo.com)))"
          connectionPoolName="salesConnectionPool"
          validateConnectionOnBorrow="true"
          sqlForValidateConnection="select 1 from DUAL"
/>
```

Configuring a Spring Data Source

Spring data source configuration is normally done through a bean definition in an application context XML file. Most applications again use the Apache DBCP implementation but the UCP implementation can be used by configuring a factory bean with the *oracle.ucp.jdbc.PoolDataSourceFactory* class and setting properties on the resulting *oracle.ucp.jdbc.PoolDataSource*.

You should be aware that a Spring-configured UCP is not destroyed when the bean context is shutdown, which is a problem if the Spring application is redeployed in a container such as Tomcat. The recommended way to correct this behavior is to use the destroy-method attribute of the Spring bean definition; however the *oracle.ucp.jdbc.PoolDataSource* doesn't expose any suitable target method. Listing 3 presents a custom bean post processor that identifies data source instances and properly removes them from the UCP manager when an application context is shutdown. In a standalone Spring application, this functionality isn't required because the UCP will use Java finalizers to perform the pool cleanup.

LISTING THREE

```
package org.mpilone.jdbc;

import java.util.*;

import oracle.ucp.UniversalConnectionPoolException;
import oracle.ucp.admin.UniversalConnectionPoolManager;
import oracle.ucp.admin.UniversalConnectionPoolManagerImpl;
import oracle.ucp.jdbc.PoolDataSource;

import org.springframework.beans.BeansException;
import org.springframework.beans.FatalBeanException;
import org.springframework.beans.factory.config.DestructionAwareBeanPostProcessor;

/**
 * <p>
 * A simple bean post processor that identified {@link PoolDataSource}s which
 * should be shutdown in the Universal Connection Pool (UCP) manager. The data
 * sources will be shutdown when the application context is shutdown. By default
 * this implementation destroys all instances of {@link PoolDataSource}s found
 * in the context. To use this class, simply create an instance in the bean
 * configuration.
 * </p>
 *
 * @author mpilone
 */
public class PoolDataSourceBeanPostProcessor implements
    DestructionAwareBeanPostProcessor
{
  /**
   * The data sources to target when performing the destroy post processing.
   */
  private List<PoolDataSource> mTargetDataSources =
      new ArrayList<PoolDataSource>();

  /*
   * (non-Javadoc)
   * @see
   * org.springframework.beans.factory.config.DestructionAwareBeanPostProcessor
   * #postProcessBeforeDestruction(java.lang.Object, java.lang.String)
   */
  @Override
  public void postProcessBeforeDestruction(Object bean, String beanName)
      throws BeansException
  {
    if (PoolDataSource.class.isAssignableFrom(bean.getClass()))
    {
      try
      {
        UniversalConnectionPoolManager mgr =
            UniversalConnectionPoolManagerImpl
                .getUniversalConnectionPoolManager();

        PoolDataSource ds = (PoolDataSource) bean;
        String poolName = ds.getConnectionPoolName();

        if (isTargeted(poolName))
        {
          mgr.stopConnectionPool(poolName);
          mgr.destroyConnectionPool(poolName);
        }
      }
      catch (UniversalConnectionPoolException ex)
      {
        throw new FatalBeanException("Unable to destroy UCP data source", ex);
      }
    }
  }

  /**
   * Checks to see if the given pool is targeted by this post processor.
   *
   * @param poolName
   *          the name of the pool
   * @return true if the pool is targeted, false otherwise
   */
  private boolean isTargeted(String poolName)
  {
    if (poolName == null)
    {
      return false;
    }

    // If we have specific data sources, only return
    // true if one of the targets matches the pool name.
    boolean targeted = mTargetDataSources.isEmpty();
    if (!targeted)
    {
      for (Iterator<PoolDataSource> iter = mTargetDataSources.iterator(); iter
          .hasNext() &&
          !targeted;)
      {
        PoolDataSource ds = iter.next();
        targeted = ds.getConnectionPoolName().equals(poolName);
      }
    }

    return targeted;
  }

  /*
   * (non-Javadoc)
   * @seeorg.springframework.beans.factory.config.BeanPostProcessor#
   * postProcessAfterInitialization(java.lang.Object, java.lang.String)
   */
  @Override
  public Object postProcessAfterInitialization(Object bean, String beanName)
      throws BeansException
  {
    return bean;
  }

  /*
   * (non-Javadoc)
```

```
   * @seeorg.springframework.beans.factory.config.BeanPostProcessor#
   * postProcessBeforeInitialization(java.lang.Object, java.lang.String)
   */
  @Override
  public Object postProcessBeforeInitialization(Object bean, String beanName)
      throws BeansException
  {
    return bean;
  }

  /**
   * Gets the data sources to target when performing the destroy post processing
   * or an empty list if all are targeted.
   *
   * @return the data sources to target or an empty list
   */
  public List<PoolDataSource> getTargetDataSources()
  {
    return mTargetDataSources;
  }

  /**
   * Sets the the data sources to target when performing the destroy post
   * processing. If not set, all {@link PoolDataSource} instances will be
   * targeted.
   *
   * @param targetDataSources
   *           the data sources to target
   */
  public void setTargetDataSources(List<PoolDataSource> targetDataSources)
  {
    mTargetDataSources = targetDataSources;
  }
}
```

Listing 4 shows the Spring bean configuration of the *PoolDataSource* and includes the *PoolDataSourceBeanPostProcessor* to support clean context shutdowns. Just like in the Tomcat case, the UCP data source will now be available to deployed applications as a configured data source and requires no changes to the application code.

```
LISTING FOUR
<bean class="org.mpilone.jdbc.PoolDataSourceBeanPostProcessor" />

<bean id="salesDataSource"
class="oracle.ucp.jdbc.PoolDataSourceFactory" factory-
method="getPoolDataSource">
          <property name="URL"
value="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=sale
s1node1.foo.com)(PORT=1521))(ADDRESS=(PROTOCOL=TCP)(HOST=sales1node2.fo
o.com)  (PORT=1521))(CONNECT_DATA=(SERVICE_NAME=sales.foo.com)))"/>
          <property name="user" value="bsmith"/>
          <property name="password" value="secret"/>
          <property name="connectionFactoryClassName"
value="oracle.jdbc.pool.OracleDataSource"/>
          <property name="connectionPoolName"
value="salesConnectionPool"/>
          <property name="connectionWaitTimeout" value="30"/>
          <property name="minPoolSize" value="5"/>
          <property name="maxPoolSize" value="25"/>
          <property name="inactiveConnectionTimeout" value="20"/>
          <property name="timeoutCheckInterval" value="60"/>
          <property name="fastConnectionFailoverEnabled"
value="true"/>
          <property name="ONSConfiguration"
value="nodes=sales1node1.foo.com:6151,sales1node2.foo.com:6151"/>
          <property name="validateConnectionOnBorrow" value="true"/>
          <property name="SQLForValidateConnection" value="select 1
from DUAL"/>
</bean>
```

Limitations and Notes

Fast Connection Failover allows the UCP to quickly terminate connections to a failed database instance; however, it does not automatically resubmit the original query, which forces the client to deal with the SQLException. This limitation requires that applications deal with the exception in a graceful manner or deliver the error to the user. There are some third-party solutions such as Spring Source's Enterprise package that provide data source wrappers that handle the node failure exception and automatically resubmit the last query to the next node. These solutions may be worthwhile if a transient error for a user in a transaction is unacceptable in the event of a RAC node failure.

The UCP documentation states that it is compatible with both 10g and 11g Oracle drivers as well as third-party drivers (assuming you don't want FCF support). However with FCF enabled, there appears to be a JVM security violation when deployed in Tomcat. Therefore, it is recommended that you use the latest 11g driver with UCP if possible.

It is important to note that the discussion so far has been about the Oracle thin JDBC driver. This driver is usually preferred by application developers because it is cross platform and has no external dependencies. However, some applications require the high-performance, native C-language based Oracle Call Interface (OCI) driver. This driver is compatible with FCF and can alternatively use Transparent Application Failover (TAF), which operates at a lower level than FCF and can automatically resubmit SELECT queries in the event of a node failure. However for most applications, the ease of deployment of the thin driver with full FCF support will outweigh any benefits offered by the OCI driver.

Highly Available JDBC

Oracle RAC is a great foundation for an HA architecture and with the new Universal Connection Pool features, Java clients can immediately take advantage of RAC with only simple data source configuration changes. In this article you saw how to configure the key components of RAC on the server side, test the configuration from the client side, and configure Tomcat and Spring to leverage the RAC features. Oracle has done an excellent job of writing lightweight, JDBC drivers that make it an excellent choice for large-scale Java applications. Give RAC a try and hit your five 9s of uptime with your current applications.

—Michael Pilone is a senior software engineer for Vangent where he is currently working on Java-based web service technologies and large scale geographical data indexing and searching.

Return to Table of Contents

Your Own MP3 Duration Calculator

Select MP3 files from a directory and calculate the total duration of the selected files on-the-fly

by Gigi Sayfan

My wife recently completed certification as a spinning instructor — you know, riding real hard on a stationary bike — and began teaching classes. It turns out that music is very important for spinning, and instructors have to prepare CDs with proper tracks for warm-up, main work-out, and cool down. Each part is measured and you have to find the right tracks with the proper combined length. Of course, you could do this manually by looking at the duration of each track and try to add them together, but this approach is tedious and error-prone. A more serious problem is that the duration of MP3 files is recorded as an ID3 tag in the file header. There is no guarantee that this number is correct (and often it is not). This situation is simply screaming for a software solution.

The solution I present in this article — the "MP3 Duration Calculator" — lets you select MP3 files from a directory and calculate the total duration of the selected files on-the-fly. The complete source code and related files are available at http://i.cmpnet.com/ddj/images/article/2010/code/MP3DurationCalculator_v1.zip. To calculate the duration of each track, the software actually opens the file in a media player and gets an accurate duration number. Figure 1 shows the folder selection dialog where you choose the folder that contains the MP3 files.

Figure 2 shows the main window of the MP3 Duration Converter.

The program is a WPF application implemented in C# over the .NET 3.5 framework. It demonstrates a few interesting aspects such as:

- Dynamically loading MP3 files
- Working with the *MediaElement* component
- Asynchronous execution of long operations while keeping the UI responsive
- On-the-fly resizing of GridView columns (more obscure than it should be)
- Data binding and custom conversion
- Application state persistence

An Introduction to WPF

WPF (Windows Presentation Foundation) is the user interface/display system of Windows. It is built on top of DirectX and can take advantage of hardware acceleration. It is more advanced than previous UI APIs (including the .NET Windows Forms), all based on GDI/GDI+ and the User32 API. WPF offers a declarative UI (XAML), animation, multi-media support, rich text model, web-like layout model, browser-like page-based navigation, rich drawing model (including 3D) based on primitives and not pixels, totally customizable

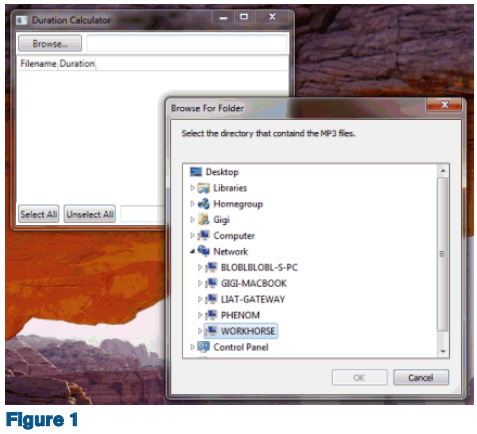


Figure 1

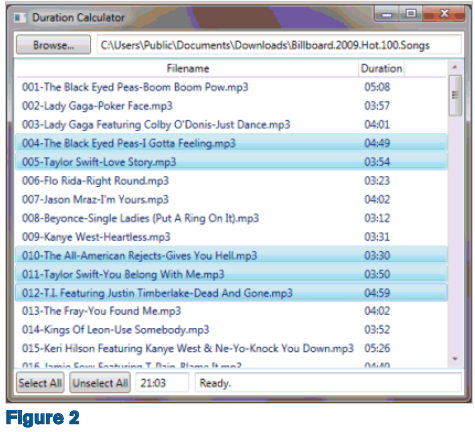


Figure 2

look-and-feel for controls (via templates and styling), and more.

Silverlight, Microsoft's rich web client platform, is based on the same design and is almost compatible, which really helps if you want to multi-target your application to the desktop and the browser. At this writing, WPF is not supported by Mono (and probably won't be in the near future), so if you need to target multiple platforms your best bet is still the Windows Forms API.

XAML

XAML, short for "eXtensible Application Markup Language," is an XML dialect for instantiating .NET objects and hooking them up. It can be used for many purposes, but currently its primary use is to define WPF user interfaces. It provides the following capabilities:

- Full namespace support
- Resource definition
- Animation definition
- Control template definition
- Hooking up event handlers
- Data binding
- Design time integration with Visual Studio 2008

This is a rich feature set for a UI description markup language. It should be noted that XAML is not required and you can code any WPF construct programmatically. However, many things are easier with XAML, especially if you use Visual Studio as your IDE.

MP3 Duration Calculator Design

The design is straightforward. There are four classes:

- *App* (the application class)
- *TrackDurations* (engine)
- *DurationConverter* (a helper class)
- *MainWindow* (GUI).

The *App* and *MainWindow* classes are code-behind classes for the corresponding XAML files: App.xaml and MainWindow.xaml.

The *App* Class

The *App* class is generated by Visual Studio if you use it (I do). The code is split across two files: One of them is called App.g.cs and you can find it in the Debug/obj folder, but don't modify it. The other file is called App.xaml.cs and you can find it in the project folder. In this file you can add application-level event handlers like *OnStartup()* and *OnExit()*. In this case it is pretty trivial:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Windows;

namespace MP3DurationCalculator
{
  /// <summary>
  /// Interaction logic for App.xaml
  /// </summary>
  public partial class App : Application
  {
  }
}
```

The two files are merged to a single class by virtue of the partial class feature of C#. In addition to the code files, there is also the application XAML file called App.xaml. This file contains a couple of namespaces, and identifies the application class (*MP3DurationCalculator.App* and the URI of the main window's XAML file. WPF takes it from here and knows how to instantiate the *App* and display the main window.

```
<Application x:Class="MP3DurationCalculator.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

The *TrackDurations* Class

The *TrackDurations* class is the workhorse that does the heavy lifting of accurately measuring the duration of each track. It utilizes the useful *MediaElement* class, which wraps the Windows Media Player engine (At least the Microsoft Windows Media Player 10 OCX for media playback).

Before diving into the *TrackDurations* class, let me introduce the *TrackInfo* class. This is basically a pair of properties: *Name* (a string) and *Duration*. A collection of *TrackInfo* objects is the main data structure that travels from the *TrackDurations* class to the *MainWindow* class. The fact that *Name* and *Duration* are properties allows the UI to bind to them directly. More about it later:

```
class TrackInfo
{
  public string Name { get; set; }
  public TimeSpan Duration { get; set; }
}
```

The *TrackDurations* class uses a delegate to notify the UI whenever it discovers the duration of another track. After processing all the tracks, it sends a null object to let the UI know it's done. Here is the signature:

```
delegate void TrackDurationsDelegate(TrackInfo ti);
```

A delegate is really a callback function. In this case, a *TrackInfo* object is sent with each call.

TrackDurations implements the *IDisposable* interface, which allows explicitly disposing of resources by calling its *Dispose()* method:

```
class TrackDurations : IDisposable
{
 ...
}
```

The class keeps references to a media element and the delegate and stores a list of files in a stack:

```
MediaElement _me;
TrackDurationsDelegate _delegate;
Stack<string> _files;
```

The media element is not managed by *TrackDurations* (it doesn't create or own it). It is passed in the constructor. The reason for this design is that a *TrackDurations* object is instantiated every time a new folder needs to be scanned, but a single media element can be used because it is relatively expansive to instantiate one. Also, there is never more than one scan going on at the same time, so there is no need to worry about conflicts.

The constructor accepts a media element, a delegate, and a list of file names. It stores the list of files in a stack that can be popped. It sets the *LoadedBehavior* of the media element to "Manual", which means programmatic control on play/pause/stop, attaches two event handlers for *MediaOpened* and *MediaFailed*, sets the *Source* property to the first file in the list, and tells the media element to play it:

```
public TrackDurations(
  MediaElement me,
  IEnumerable<string> files,
  TrackDurationsDelegate d)
{
  Debug.Assert(me != null);
  Debug.Assert(d != null);
  _delegate = d;
  _files = new Stack<string>(files);
  Debug.Assert(_files.Count >  0);
  _me = me;
  _me.LoadedBehavior = MediaState.Manual;
  _me.MediaOpened += _onMediaOpened;
  _me.MediaFailed += _onMediaFailed;
  _me.Source = new System.Uri(_files.Peek());
  _me.Play();
}
```

The result of all these operations is that the media element will open the first file in order to play it. If the open operation succeeds, the *_onMediaOpened()* event handler will be called; otherwise, *_onMediaFailed()* will be called.

The *Dispose()* method simply detaches the two event handlers:

```
public void Dispose()
{
  _me.MediaOpened -= _onMediaOpened;
  _me.MediaFailed -= _onMediaFailed;
}
```

The *_getNextTrack()* method is used by both event handlers to move on after the current track has been handled. It reads the next file from the stack (processed files are removed by the event handlers) and tries to play it. If there are no more files, the delegate is called with a null object to signal the end of processing:

```
void _getNextTrack()
{
  if (_files.Count == 0)
  {
    _delegate(null);
  }
  else
  {
    // Get the next file
    _me.Source = new System.Uri(_files.Peek());
    _me.Play();
  }
}
```

When the media element opens successfully the current file as a result of a *Play()* call the *MediaOpened* event is fired. That causes the *_onMediaOpened()* event handler to be called. At this point, the correct duration is available as the *NaturalDuration* property of the media element. The event handler pauses the media element to avoid accidental playing, creates a new *TrackInfo* object, stops the media element completely, invokes the delegate, and finally gets the next track.

```
private void _onMediaOpened(object sender, RoutedEventArgs
e)
{
  _me.Pause();
  Debug.Assert(_me.NaturalDuration.HasTimeSpan);
  TimeSpan duration = _me.NaturalDuration.TimeSpan;
  var ti = new TrackInfo
  {
    Name = System.IO.Path.GetFileName(_files.Pop()),
    Duration = duration
  };

  _me.Stop();
  _delegate(ti);

  _getNextTrack();
}
```

If for some reason,the media element failed to open the track, the *MediaFailed* event is fired and the *_onMediaFailed()* event handler is called. All it does is pop the bad file from the files list and get the next track:

```
private void _onMediaFailed(object sender, RoutedEventArgs e)
{
  // Get rid of the bad file
  _files.Pop();
  _getNextTrack();
}
```

The *MainWindow* Class

The *MainWindow* class is responsible for displaying the main window and also functions as the controller. In a larger system I would probably split it into multiple files and maybe use some application framework, but in such a small utility I felt okay with just managing everything in a single class. The UI is defined declaratively using XAML in *MainWindow.xaml* and the controller code is in the code-behind class *MainWindow.xaml.cs*.

Let's start with the UI. The root element is naturally the <*Window*>. It contains the usual XML namespaces for XAML and the local namespace for the *MP3DurationCalculator*. It also contains the

title of the window and its initial dimensions. There is also a *<Window.Resources>* subelement that contains resources that can be shared by all elements in the window. In this case it's the *DurationConverter* class. I'll talk more about it later. Here is the XAML for *<Window>* element:

```
<Window x:Class="MP3DurationCalculator.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:MP3DurationCalculator"
  Title="Duration Calculator" Height="293" Width="376">
  <Window.Resources>
    <local:DurationConverter
x:Key="DurationConverter"></local:DurationConverter>
  </Window.Resources>
  ...
  </Window>
```

The *Window* element corresponds to the WPF *Window* class, which is a content container. That means it can hold just one item. That sounds pretty limited, but in practice this single item can be a layout container that itself can hold many items. This is how WPF Layout works. You nest containers within containers to get the desired layout. The most common and flexible layout container is the grid. WPF Layout is a big topic that warrants its own article (or more). In this article, I try to get away with the minimal amount of explanations necessary to understand the layout of the MP3 Duration Converter. The grid element contains two elements: a media element and a dock panel. The media element is not displayed because we use it only to play audio. The dock panel contains the rest of the UI. Here is a collapsed view of the grid:

```
<Grid>
  <MediaElement
       Height="0"
       Width="0"
       Name="mediaElement"
       LoadedBehavior="Manual" />
  <DockPanel LastChildFill="True">
    ...
  </DockPanel>
</Grid>
```

The dock panel is another layout container, which contains three stripes arranged vertically. The top stripe contains a browse button for selecting a folder and a text box to display the selected folder name (see Figure 3).The middle stripe contains a list view that displays the MP3 files in the selected folder (see Figure 4).

The bottom stripe contains a couple of buttons for selecting all or none of the files and a text box to display the total duration of the selected songs (see Figure 5).

The top stripe is a dock panel that has several attributes. The *DockPanel.Dock* attribute is actually an attached attribute and it applies to the outer dock panel (the one that contains all three stripes). It has the value *Top*, which means that the top strip will be docked to the top part of the outer dock panel. The height is defined to be 30 and the vertical alignment is set to *Stretch*, which means that the top stripe will stretch to fit the entire width of the outer dock panel. The *LastChildFill* attribute is set to *True*, which means that the last child laid out will fill the remaining area. The top stripe contains two controls, a button, and a text box. Both have attached *DockPanel.Dock* properties that apply to the top stripe itself. The button is docked to the left and the text box is docked to the right.

In addition, since the top stripe has a *LastChildFill* ="*True*" the text box will stretch to fill the remaining area of the top stripe after the button is laid out. The margin attribute of the button and text box ensures that there will be some spacing between them, so they are not squished together. The button has an event handler called *Click* that calls the *btnSelectDir_Click* method when the button is clicked. This method is defined in the code-behind class. Here is XAML for the top stripe:

```
<DockPanel
 DockPanel.Dock="Top"
 LastChildFill="True"
 Height="30"
 Margin="0" VerticalAlignment="Stretch"
>
 <Button DockPanel.Dock="Left"  Height="22" Name="btnSelectDir"
   Width="84" Margin="3"
   Click="btnSelectDir_Click">Browse...</Button>
 <TextBox Height="22" Name="tbTargetFolder" MinWidth="258"
   Margin="3" TextChanged="tbTargetFolder_TextChanged"></TextBox>
```

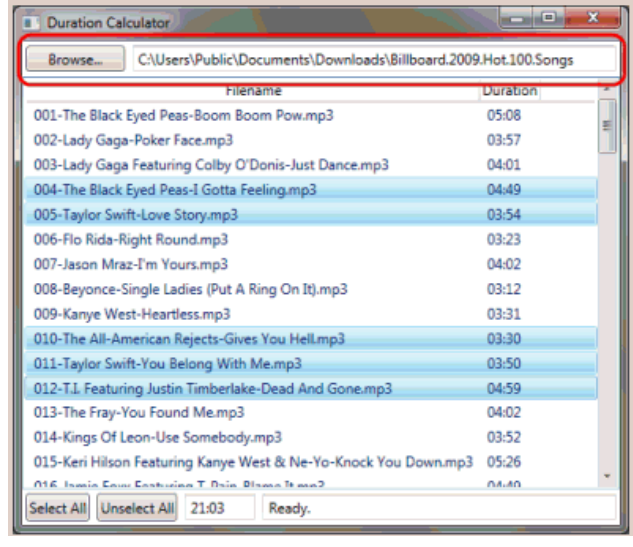


Figure 3

Figure 4

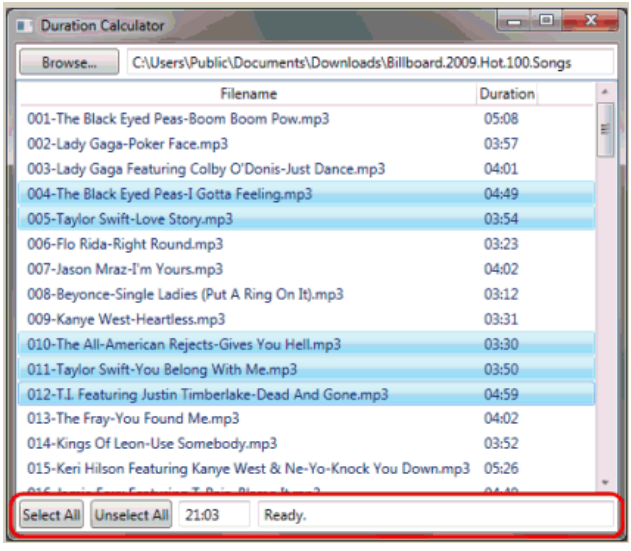


Figure 5

```
</DockPanel>
```

The second stripe is actually the bottom stripe and not the middle stripe. The reason for this supposed inconsistency is that I want the middle stripe to fill the remaining area of the outer stripe after the top and bottom have been laid out (according to the *LastChildFill* attribute), so it must be laid out last. I actually find it a little disorienting. I would prefer if you could simply set one of the *DockPanel.Dock* children to *Fill* instead of making sure it appears last in the XAML file. Anyway, that's the choice the WPF designers made, so the second stripe is the bottom stripe. It is very similar to the top stripe. It contains two buttons for selecting all files or unselecting all files and two text boxes. The *total* text box contains the total duration of all the selected files and the *status* text box shows various status messages. Named elements like *total* and *status* can be accessed using their name in the code-behind class. The text of the "*total*" text box is data bound to the *total.Text* property of the *MainWindow* object. More on that later. Here is the XAML for the bottom stripe:

```
<DockPanel
   DockPanel.Dock="Bottom"
   LastChildFill="True"
   Height="30"
   Margin="0" VerticalAlignment="Stretch" >
   <Button DockPanel.Dock="Left" Margin="3"
     Click="SelectAll_Click">Select All</Button>
   <Button DockPanel.Dock="Left" Margin="3"
     Click="UnselectAll_Click">Unselect All</Button>
   <TextBox DockPanel.Dock="Left" Height="22" Name="total"
     Width="60" Margin="3"
   Text="{Binding Path=Self, Converter={StaticResource
     DurationConverter}}"></TextBox>
   <TextBox Height="22" Name="status" MinWidth="100" Margin="3" />
  </DockPanel>
```

The middle stripe is a list view. It has no *DockPanel.Dock* attached property because it is the last child and thus just fills the area left between the top stripe and the bottom stripe. When the Window is resized, the top and bottom stripe keep their fixed height and the list view is resize to accommodate the new height of the window. The list view has a name ("files") because it is accessed programmatically from the *MainWindow* class. It also has an event handler for the *SelectionChanged* event. The list view also has a view property, which is a grid view in this case (the list view supports several view types including custom views). The grid view has two columns, which are bound to the *Name* and *Duration* properties of its item object. The items that populate the list must be objects that have *Name* and *Duration* properties. The *Duration* property is converted using the *DurationConverter* to a more display-friendly format. Here is the XAML for the list view:

```
<ListView MinHeight="223" Name="files" MinWidth="355"
  Background="White" SelectionChanged="files_SelectionChanged">
<ListView.View>
  <GridView>
   <GridView.Columns>
     <GridViewColumn
     Header="Filename"
     DisplayMemberBinding="{Binding Path=Name}"
     Width="Auto"
     />
     <GridViewColumn
     Header="Duration"
      DisplayMemberBinding="{Binding Path=Duration,
        Converter={StaticResource DurationConverter}}"
     Width="Auto"
     />
   </GridView.Columns>
  </GridView>
</ListView.View>
</ListView>
```

Let's move on to the *MainWindow* code-behind class. This class oversees the following actions: browse for a new folder that contains MP3 files, collect track info about every MP3 file, select files from the current folder, calculate and display the total duration of all the selected files.

The class works with the following member variables:

- *_folderBrowserDialog*. This is a *System.Windows.Forms* component used to display a dialog for browsing the filesystem and selecting a folder. WPF doesn't have its own component, but using the *Windows.Forms* component is just as easy.
- *_trackDurations*. This is an instance of our very own *TrackDurations* class described earlier.
- *_results*. This is an observable collection of *TrackInfo* objects received from the *TrackDurations* class whenever the selection of MP3 files is changed.
- *_fileCount*. This is just an integer that says how many files are in the current selected directory.
- *_maxLength*. This is an integer that measures the length of the longest track name. It is used to properly resize the list view columns to make sure track names are not truncated.

```
FolderBrowserDialog _folderBrowserDialog;
TrackDurations _trackDurations;
ObservableCollection<TrackInfo> _results;
int _fileCount = 0;
double _maxLength = 0;
```

In addition to these member variables defined in the code, the *MainWindow* class also has additional member variables, which are all the named elements in the XAML file like the *files* list view, and the *total* and *status* text boxes.

The constructor calls the mandatory *InitializeComponent()* method that reads the XAML and builds all the UI and then instantiates the folder browser dialog and initializes its properties so it starts browsing from the *Downloads* directory (folder) under the current user's desktop directory. It also instantiates *_results* to an

empty collection of *TrackInfo* objects. The most important action is binding the 'files' list view to the *_results* object by assigning *_results* to *files.ItemsSource*. This ensures that the files list view will always display the contents of the *_results* object.

```
public MainWindow()
{
  InitializeComponent();
  _folderBrowserDialog = new FolderBrowserDialog();
  _folderBrowserDialog.Description =
    "Select the directory that containd the MP3 files.";
  // Do not allow the user to create new files via the
  // FolderBrowserDialog.
  _folderBrowserDialog.ShowNewFolderButton = false;
  _folderBrowserDialog.RootFolder =
     Environment.SpecialFolder.DesktopDirectory;
  var dt = Environment.GetFolderPath
    (Environment.SpecialFolder.DesktopDirectory);
  var start = System.IO.Path.Combine(dt, "Downloads");
  _folderBrowserDialog.SelectedPath = start;
  _results = new ObservableCollection();
  files.ItemsSource = _results;
}
```

The action is usually triggered by selecting a folder to browse. When you click the 'Browse...' button, the *BrowseFolderDialog* pops up and lets you select a target directory. If the user selected a folder and clicked 'OK' the text of the *tbTargetFolder* text box is set to the selected path. If the user clicked 'Cancel' nothing happens.

```
void btnSelectDir_Click(object sender, RoutedEventArgs e)
{
  DialogResult r = _folderBrowserDialog.ShowDialog();
  if (r == System.Windows.Forms.DialogResult.OK)
  {
    tbTargetFolder.Text = this._folderBrowserDialog.SelectedPath;
  }
}
```

Another way to initiate the action is to directly type a folder path into the *tbTargetFolder* text box. In both cases, the *TextChanged* event of the text box will fire and corresponding event handler will be called. It will check if the text constitutes a valid folder path and if so calls the *collectTrackInfo()* method.

```
private void tbTargetFolder_TextChanged(object sender,
                                        TextChangedEventArgs e)
{
  if (Directory.Exists(tbTargetFolder.Text))
    collectTrackInfo(tbTargetFolder.Text);
}
```

The *collectTrackInfo()* method is pretty central so I'll explain it in detail. First of all, it disables the browse button and the target folder text box to ensure that the user doesn't try to go to a different folder while the collection is in progress. This prevents a whole class of race condition and synchronization issues.

```
void collectTrackInfo(string targetFolder)
{
  btnSelectDir.IsEnabled = false;
  tbTargetFolder.IsEnabled = false;
```

The next part is getting all the MP3 files in the target folder. I used a LINQ expression that reads almost like English: "From the files in the target folder select all the files whose extension is ".mp3":

```
var mp3_files = from f in Directory.GetFiles(targetFolder)
                where System.IO.Path.GetExtension(f) == ".mp3"
                select f;
```

The collection of MP3 files is returned in the reverse order for some reason, so I reverse them back.

```
mp3_files = mp3_files.Reverse();
```

Now, the *_fileCount* member variable is updated and the *_results* collection is cleared:

```
_fileCount = mp3_files.Count();
_results.Clear();
```

If *_fileCount* is 0 it means no MP3 files were found and there is no need to collect any track information. The *status* text box is updated and the browse button and the target folder text box are enabled.

```
if (_fileCount == 0)
{
  status.Text = "No MP3 files in this folder.";
  btnSelectDir.IsEnabled = true;
  tbTargetFolder.IsEnabled = true;
}
```

If *_fileCount* is greater than 0, then a new instance of *_trackDurations* is created and receives the media element, the collection of MP3 files in the target folder, and the *onTrackInfo()* callback.

```
else
  _trackDurations = new TrackDurations(mediaElement,
                                       mp3_files,
                                       onTrackInfo);
```

The *onTrackInfo()* callback is called by *TrackDurations* every time the information about one of the tracks is collected and once more in the end (with a null *TrackInfo*). If *ti* (the *TrackInfo* object) is null, it means we are done with the current directory. The *_maxLength* variable is reset to 0, the *_trackDurations* object is disposed of, the status text box displays "Ready." and the selection controls are enabled again.

```
void onTrackInfo(TrackInfo ti)
{
  if (ti == null)
  {
    _maxLength = 0;
    _trackDurations.Dispose();
    status.Text = "Ready.";
    btnSelectDir.IsEnabled = true;
    tbTargetFolder.IsEnabled = true;
  }
```

If *ti* is not null, it means a new *TrackInfo* object was received asynchronously. First of all, the *TrackInfo* object is added to the *_results* collection. As you recall (or not), the *_results* collection is data-bound to the list view, so just adding it to *_results* make the new track info show up in the list view.

```
else
{
  _results.Add(ti);
```

The next step is to make sure the new file name fits in the first list view column. This is a little cumbersome and involves first creating a *FormattedText* object with the proper font of the list view and then checking if the size of this formatted text object is greater than the current *_maxWidth*. If it is greater, it becomes the new *_maxWidth* and the width of first column of the list view is set to the new *_maxWidth*.

```
  // Make sure the new filename fits in the column
  var ft = new FormattedText(
    ti.Name,
    CultureInfo.GetCultureInfo("en-us"),
```

```
            System.Windows.FlowDirection.LeftToRight,
            new Typeface(files.FontFamily,
                         files.FontStyle,
                         files.FontWeight,
                         files.FontStretch),
            files.FontSize,
            Brushes.Black);

          if (ft.Width > _maxLength)
          {
            _maxLength = ft.Width;
            var gv = (GridView)files.View;
            var gvc = gv.Columns[0];
            var curWidth = gvc.Width;

            // Reset to a specific width before auto-sizing
            gvc.Width = _maxLength;
            // This causes auto-sizing
            gvc.Width = Double.NaN;

          }
```

The last part of the *onTrackInfo()* method is updating the status line with a current count of track info objects out of the total number of MP3 files.

```
          // Update the status line
          var st = String.Format("Collecting track info {0}/{1} ...",
                                 _results.Count,
                                 _fileCount);
          status.Text = st;
        }
      }
```

After all the information has been collected, the user may select files in the list view using the standard Windows selection conventions (click/space to select, ctrl+click/space to toggle selection, and shift+click/space to extend selection). Whenever the selected files change, the *SelectionChanged* event is fired and the event handler calculates the total duration of all the currently selected files and updates 'total' text box.

```
    private void files_SelectionChanged(object sender,

SelectionChangedEventArgs e)
    {
      var tp = new TimeSpan();
      foreach (var f in files.SelectedItems)
      {
        tp += ((TrackInfo)f).Duration;
      }

      var d = new DateTime(tp.Ticks);
      string format = "mm:ss";
      if (tp.Hours > 0)
        format = "hh:mm:ss";

      total.Text = d.ToString(format);
    }
```

There are two buttons called 'Select All' and 'Unselect All' that are hooked to corresponding event handlers and simply select or unselect all the files in list view when clicked. This results of course in a *SelectionChanged* event handled by the file's *_SelectionChanged* event handler described earlier.

```
    private void SelectAll_Click(object sender, RoutedEventArgs e)
    {
      files.SelectAll();
    }
    private void UnselectAll_Click(object sender, RoutedEventArgs e)
    {
      files.UnselectAll();
    }
```

The *DurationConverter* Class

There is one last small class that demonstrates an interesting feature of WPF data binding called "data conversion." When binding a data source to a target UI element, you may often want to format it a little differently. You can do it of course by creating a new class that consumes the original data object, formats it, and then binds the UI element to the new object. But WPF provides a standardized solution in the form of the *IValueConverter* interface. To use it you implement a value converter class that implements the interface (a *Convert()* and *ConvertBack()* methods), decorate it with the *ValueConversion* attribute and you are good to go. The *ConvertBack()* method is useful in two-way (or write-only) binding scenarios where you want to update the source object when you modify the value in the UI. The *DurationConverter* class is used for one-way binding only, so the *ConvertBack()* method just throws a *NotImplementedException*. The *Convert()* method takes a *TimeSpan* value and converts it to a string that displays minutes and seconds (if less than an hour) or hours, minutes, and seconds. This is more human-readable and there is no need for a finer resolution for the purpose of the MP3 Duration Converter:

```
  [ValueConversion(typeof(TimeSpan), typeof(string))]
  public class DurationConverter : IValueConverter
  {
    public object Convert(object value,
                          Type targetType,
                          object parameter,
                          CultureInfo culture)
    {
      var duration = (TimeSpan)value;
      var d = new DateTime(duration.Ticks);
      string format = "mm:ss";
      if (duration.Hours > 0)
        format = "hh:mm:ss";

      return d.ToString(format);
    }

    public object ConvertBack(object value, Type targetType,
object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
  }
```

The *DurationConverter* is used in the XAML for the text property of the total text box:

```
  <TextBox DockPanel.Dock="Left"
           Height="22"
           Name="total"
           Width="60"
           Margin="3"
           Text="{Binding Path=Self,
           Converter={StaticResource DurationConverter}}">
  </TextBox>
```

Conclusion

This article showcased the MP3 duration converter and demonstrated various WPF techniques and practices that cover a lot of ground: UI, asynchronous programming, data binding, and multimedia control. In the next article, I will tell you what happend when I released the program to my wife (hint: not a lot of gratitude).

— Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

Return to Table of Contents

Proving Correctness of an OS Kernel

This proof surpasses by far what other formal-verification projects have achieved

by Gernot Heiser

Earlier this year, Open Kernel Labs (http://www.ok-labs.com/) and its research partner, NICTA (Australia's Information and Communications Technology Research Centre of Excellence; http://www.nicta.com.au/), announced completion of groundbreaking long-term research and development providing formal mathematical proof of the correctness of a general-purpose operating-system microkernel and virtualization platform.

The project centered on the need to assure extremely high levels of reliability, safety, and security in mission-critical domains that include aerospace, national security, and transportation. The correctness proof will significantly reduce the effort required for certification under formal security or safety evaluation regimes, such as Common Criteria (http://www.commoncriteriaportal.org/), and paves the way for deploying trustworthy virtualization technology for business-critical applications in mobile telephony, business intelligence, and mobile financial transactions.

Formal Verification: Key to Secure and Reliable Software

Existing certification regimes center on software processes, testing, and conformance to specifications of models of software. By contrast, the formal verification project actually proved the correctness of the code itself, using formal logic and programmatic theorem checking — the first time this has been achieved for a general-purpose operating system (OS) kernel or hypervisor. The verification eliminated a wide range of exploitable errors in the kernel, including design flaws and code-based errors like buffer overflows, null pointer dereference and other pointer errors, memory leaks and arithmetic overflows, and exceptions.

The project entailed more than five years of work by NICTA and researchers from the University of New South Wales on behalf of OK Labs. The joint team verified 7,500 lines of source code, proving over 10,000 intermediate theorems in over 200,000 lines of formal proof. The verified code base, which underlies the OK:Verified platform (project name seL4; http://ertos.nicta.com.au/research/sel4/), is a newly developed OS microkernel. It is derived from the globally developed and deployed open source L4 project and is designed for use as a hypervisor that can support multiple virtual machines.

The code constitutes a third-generation microkernel and is influenced by the Extremely Reliable Operating System (EROS) project (http://www.eros-os.org/). It features abstractions for virtual address spaces, threads, interprocess communication (IPC), and, like EROS but unlike most commercial systems, capabilities for access control. Initial development and all verification work was performed on an ARMv6-based platform, with a subsequent port of the kernel (so far without proof) to the x86 architecture.

The proof was machine-checked and is one of the largest proofs ever completed. Formal proofs for specific properties have been conducted for smaller kernels and code bases, but what the team achieved constitutes a general, functional correctness proof, which has never before been realized for software of this complexity or size.

To reach this milestone, the team invented new techniques in formal machine-checked proof methods, made advances in the mathematical understanding of real world programming languages, and developed new methodologies for rapid prototyping of OS kernels. This work went beyond the usual checks for the absence of certain specific errors. Instead, it verified full compliance with the system specification. The project has yielded not only a verified microkernel but

also a body of techniques that can be used to develop and verify other pieces and types of software.

Smaller is Better

Large code size and complexity are the archenemies of software reliability. Experience shows that well-engineered software tends to exhibit at least two to five bugs per thousand lines of code (kLoC), and where there are bugs, there are, most certainly, security or safety vulnerabilities. And, as many mobile/wireless devices now field hundreds of thousands, even millions, of lines of code, they therefore may have literally thousands of bugs with concomitant vulnerabilities.

The security and reliability of a computer system can only be as good as that of the underlying OS kernel. The kernel, defined as the part of the system executing in the most privileged mode of the processor, has unlimited hardware access. Therefore, any fault in kernel implementation has the potential to undermine the correct operation of the entire system.

The existence of bugs in any sizeable code base is inevitable. As a consequence, when security or reliability is paramount, the usual approach is to reduce the amount of privileged code, in order to minimize the exposure to bugs.

The best way to manage this complexity and accompanying reliability challenges is to architect systems to limit the size of security- or safety-critical components. The trusted computing base (TCB) comprises the part of the system that can violate security policies. If the TCB is small enough, it becomes feasible to engineer it to the highest standards.

TCB reduction is a primary motivation behind security kernels and separation kernels, the MILS (multiple independent levels of security) approach, the use of small hypervisors as a minimal trusted base, as well as systems that require the use of type-safe languages for all code except some "dirty" core. It is the reason why Common Criteria, at the strictest evaluation level, requires the system under evaluation to have a "simple" design.

Easily said, but not as easily done. Rich functionality (like GUIs and sophisticated application programs) is generally implemented on top of powerful, function-rich OSes like Windows, Linux, or Android. These are themselves large and complex, and consequently full of bugs. But the OS is inherently part of the TCB of whatever program is running on top.

One simple and elegant solution to the challenge of TCB reduction is based on virtualization: A system is partitioned into several virtual machines (VMs), each containing a separate subsystem. For a typical mobile device, one VM would host a rich OS supporting the user-facing software, another a real-time OS (RTOS) supporting the wireless communication software, and a third VM would contain and protect security- or safety-critical software (possibly running directly on the virtual hardware without any OS). In such a setup, the TCB of the critical subsystem would consist of little more than the hypervisor that implements the virtual machines. The security and safety of the system would then hinge on the trustworthiness of a relatively modest-sized hypervisor. By adapting the hypervisor to the specific requirements of mobile systems (and the similar requirements of other consumer electronic devices), with careful design and implementation of a minimum set of primitives, a total TCB size of approximately 10 kLoC is achievable.

The number of bugs in such a small code base can be reduced to a small number by using good software-engineering techniques, and one can hope that none of them constitute vulnerabilities. However, with human lives and national security at stake, we should take a more pessimistic approach than hoping for the best, and assume that there are critical bugs unless proven otherwise.

Formal Verification

Other than exhaustive testing, formal verification is the only known way to guarantee a system free of bugs. This approach is not well known and seldom used, because it is considered expensive and only applicable to very small bodies of code. However, a small microkernel is amenable to formal verification, as the project demonstrated.

The technique used for formal verification is a refinement proof using interactive theorem proving. Refinement shows a formal correspondence between a high-level "abstract" and a low-level "concrete" model. Refinement proves that all possible behaviors of the concrete model are captured by the abstract one.

In this case, the abstract model is the specification of kernel behavior, essentially its ABI, expressed in a language called "higher-order logic" (HOL). The concrete model is the kernel's implementation — the C code processed by a C compiler to produce an executable kernel. Refinement then proves that the kernel truly behaves according to its specification, i.e., it is functionally correct.

Functional correctness means that implementation always strictly follows the high-level abstract specification of kernel behavior. Provable functional correctness encompasses traditional design and implementation safety properties such that the kernel will never crash, and will never perform an unsafe operation. Functional correctness also enables prediction of precisely how a kernel will behave in every possible situation. Such assurances rule out any unexpected behavior. One implication is that it becomes impossible to inject arbitrary code into the kernel as happens with popular "stack-smashing" attacks — our kernel is provably immune to such attacks.

At the concrete level, the translation from C into HOL is correctness-critical; the team took great care to model the C subset semantics precisely, treating C semantics, types, and memory model exactly as the standard prescribes, e.g., with architecture-dependent word size, padding of structs, type unsafe casting of pointers, and arithmetic on addresses. The actual proof was conducted with and

checked by an interactive theorem prover called Isabelle/HOL (http://www4.informatik.tu-muenchen.de/~nipkow/LNCS2283/). Interactive theorem proving requires human intervention and creativity to construct and guide the proof. However, it has the advantage of not being constrained to specific properties or finite, feasible state spaces, unlike more automated methods of verification, such as static analysis or model checking.

Kernel Design for Verification

To facilitate verification, kernel design should minimize the complexity of its components. Ideally, kernel code (and associated proofs) would consist of simple statements that rely on explicit local state, with simple invariants. These smaller elements could then be integrated into larger elements that avoid exposing underlying local elements. Unfortunately, OS kernels are not usually so structured and instead feature interdependent subsystems, as is the case for a small, high-performance microkernel, such as the various members of the L4 microkernel family. By removing everything from the kernel that can securely be implemented in user space, the microkernel is the "essence of OS messiness," and its components are highly interconnected.

A major design goal of OK:Verified was suitability for real-world use, requiring performance comparable to the fastest existing microkernels. Therefore, the kernel design methodology aims to minimize proof complexity without compromising performance.

OS developers tend to take a bottom-up approach to kernel design. They strive for high performance by managing hardware efficiently, leading to designs governed by low-level details. In contrast, formal methods practitioners tend toward top-down design, as proof tractability is determined by system complexity.

A top-down approach results in designs based on simple models with a high degree of abstraction from hardware. As a compromise that blends both views, we adopted an approach based around an intermediate target, readily accessible to both OS developers and formal methods practitioners. We used the functional programming language Haskell as a tool for OS developers, while at the same time providing an artifact for automatic translation into the theorem-proving tool.

This resulted in a high-level prototype of the kernel, implemented in Haskell, (a general-purpose, purely functional programming language, with non-strict semantics and strong static typing) that is itself an executable program. The prototype required design and implementation of algorithms that manage the low-level hardware details, and as such shares many implementation details of the real kernel.

To execute the Haskell prototype in a reasonably realistic setting, the team linked it with software derived from QEMU, a portable processor emulator that relies on dynamic binary translation for speed (http://wiki.qemu.org/Main_Page). Normal user-level execution is enabled by the emulator, while traps are passed to the kernel model, which computes the result of the trap. The prototype modifies the user-level state of the emulator to appear as if a real kernel had executed in privileged mode.

This arrangement provides a prototyping environment that enables low-level design evaluation from both the user program and kernel perspective, including low-level physical and virtual memory management. It also provides a realistic execution environment that is binary-compatible with the real kernel. Employing the standard ARM tool chain for developing software and executing it on the Haskell kernel had the nice side effect of letting the team port software to the new kernel and get experience with its ABI before the "real" kernel existed.

Although the Haskell prototype provides an executable model and implementation of the final design, it is not itself the final production kernel. We manually reimplemented the model in the C programming language for several reasons. First, the Haskell runtime is a significant body of code (much bigger than our kernel), which would be hard to verify for correctness. Second, the Haskell runtime relies on garbage collection, which is unsuitable for real-time environments.

Additionally, using C enables optimization of the low-level implementation for performance. Although an automated translation from Haskell to C would have simplified verification, the team would have lost most opportunities to optimize the kernel, required for adequate microkernel performance. Instead, we could build a kernel that performs on par with the fastest kernels.

Other elements of the kernel design, such as incorporating global variables, memory management, concurrency, and I/O are fully described in "seL4: Formal Verification of an OS Kernel" (http://www.ok-labs.com/whitepapers/sample/sel4-formal-verification-of-an-os-kernel). The kernel implementation used a slightly restricted subset of the C language to ease verification. Specifically, we outlawed *goto* statements, fall-through in *switch* statements, and the use of functions with side effects in expressions — all features good programmers avoid anyway. The most serious restriction was prohibiting passing of pointers to stack variables as function arguments (but pointers to global variables are allowed).

Verifying the Verification

Often raised is the question of proof correctness: How can we be certain that a proof is correct? After all, a large percentage of proofs published in the mathematical literature are faulty. Could we also have introduced such faults in one of our proofs?

Faulty proofs are generally traditional "pen-and-paper" proofs, and are prone to human error, such as typos, oversights, or incorrect or incomplete applications of mathematical transformations. No such errors are possible in a theorem prover; it will only apply a transformation if permitted by the underlying mathematical logic.

The correctness of the proof then depends on the mathematical logic itself, and the prover's core, its proof verifier. Both are independent of the project, and have been used for countless proofs

across a wide range of problems. The logic and verification approach itself is the result of more than 30 years of research in theorem proving, and theorem proving can now achieve a degree of trustworthiness for a formal, machine-checked proof that far surpasses confidence levels relied upon in engineering or mathematics for our daily survival (e.g., in most safety-critical or life-critical systems). Furthermore, the Isabelle prover allows extracting a prover-independent representation of the proof, which can be fed into an independently developed proof verifier. Such a tool can be relatively simple, implemented in a few kLoC, and could be further used to increase the level of confidence in the proof, for example, as part of a formal security evaluation.

The proof shows that the kernel cannot deadlock, enter into infinite loops, take exceptions, or otherwise crash

The remaining assumptions we made are the correctness of the compiler, a few small sequences of assembly code in the kernel, and the hardware. Each of them, if faulty, could lead to an incorrect operation of the kernel, even if its implementation is correct. Unavoidably, if underlying hardware operates incorrectly, it is impossible to offer guarantees about the operation of the software.

Dependence on the compiler can be removed by the use of a verified compiler, proven to produce correct code. An optimizing verified compiler for the ARM architecture was recently developed by researchers at the INRIA lab in France. Compiling our kernel with the INRIA compiler will eliminate the C compiler as a source of incorrect implementation.

Trusting assembler code in the kernel is a temporary restriction. The team understands how to verify these assembly sequences, and will do so in the near future.

Cost of Verification

Much of the cost involved in our project constituted a one-time expenditure. The experience gained can further reduce the cost of repeating the exercise on a similar object. We estimate that we could now design, implement, and verify similar systems at a cost that is a tiny fraction of the cost of high-assurance software. Cost calculations included kernel design (which itself includes many innovative features, beyond the scope of this article), implementation, development of proof tools, techniques and libraries, and the proof itself. This costs runs about an order of magnitude less than typical development, quality-assurance, and evaluation cost of software evaluated at the highest levels of Common Criteria (which provides much weaker assurance than our proof).

Our approach is thereby very attractive, and opens the possibility of deploying truly bulletproof software in areas where high assurance has been considered far too expensive.

Conclusion

What the result means is that OK Labs and its partners have completed a rigorous mathematical proof that kernel implementation (i.e., C code compiled into the kernel image that executes on a target platform) is consistent with its formal specification (a mathematical representation of the kernel ABI). The proof assures that all possible behaviors of the kernel are a subset of what is allowed by the specification. Such assurance makes a very strong statement, much stronger than ever achieved for a general-purpose OS kernel.

In particular, the proof explicitly rules out the possibility of typical bugs, such as typos, buffer overflows, and null-pointer dereferences. The proof shows that the kernel cannot deadlock, enter into infinite loops, take exceptions, or otherwise crash. It also shows that no code injection into the kernel is possible. The kernel can only "misbehave" if underlying assumptions are invalidated (such as a hardware fault or a compiler bug).

This proof surpasses by far what other formal-verification projects have achieved. For example, there have been a small number of systems subjected to similar processes, but with verification of a model of the kernel, rather than the kernel code itself. This approach leaves a gap to be bridged by informal (and thus imprecise and not completely reliable) correspondence arguments between model and code.

The proof also surpasses the capability of static analysis techniques, which automatically analyze programs to detect possible bugs, using a mathematical technique called model checking. Such techniques can only check for specific properties, and cannot produce a general statement of complete adherence to the specification as performed by team.

The team not only analyzed specific aspects of the kernel, such as safe execution, but also provided a full specification and proof for precise kernel behavior. We have shown the correctness of a very detailed, low-level design of OK:Verified and have formally verified its C implementation. We assumed the correctness of the compiler, assembly code, boot code, management of caches, and the hardware; we proved everything else.

In addition, we have created a methodology for rapid kernel design and implementation that is a fusion of traditional OSes and formal methods techniques. We found that our verification focus improved the design and was surprisingly often not in conflict with achieving performance.

— Gernot Heiser is CTO and founder of Open Kernel Labs.

Return to Table of Contents

Debug It! Book Review

Reviewed by Mike Riley

Debug It!: Find, Repair, and Prevent Bugs in Your Code
Paul Butcher
The Pragmatic Bookshelf
$34.95

A recent article I read regarding the decline of the use of specific debugging tools sparked my interest in the latest software bug-hunting best practices and trends. The article's hypothesis was that debugging tools were becoming less important due to the increase in test harnessing at a project's outset. Read on to see if *Debug It!* author Paul Butcher agrees.

Debug It!: Find, Repair, and Prevent Bugs in Your Code follows the same advice format as other Pragmatic Bookshelf titles similar to *The Passionate Programmer* and *Land the Tech Job You Love*. Instead of listing large blocks of code and debugging output, the book extols the exceptional qualities of being an effective software debugging expert.

The book is divided into three parts: The Heart of the Problem (dealing with identifying, reproducing, diagnosing, fixing, and preventing future bugs), The Bigger Picture (bug tracking and prioritization), and Debug-Fu (dealing with special cases like backwards compatibility, debugging concurrencies, setting up the ideal debugging environment, using assertions, and anti-patterns). The author provides a clear, conversational style to each topic, keeping what could have been a very dry presentation lively and engaging. His frequent stories of his own situations also made me relate more closely to his situation and pay more attention to his outcome and advice. Thoughts of, "Oh yeah, I've been there," were followed by "I wonder how he dealt with that problem," and occasionally punctuated by "Good idea. I will have to try that." While some of the advice was obvious (at least to readers like me who have been writing code in some form over the last 25 years), others (especially in the Debug-Fu section) were relevant and timely. For example, the brief sections on concurrency and the Heisenbugs (inspired by the Heisenberg Uncertainty Principle, since it's "a bug that 'goes away' the instant you start looking for it") were insightful and much appreciated.

It's clear that the author's years of debugging experience were superbly distilled into this collection of high-impact advice. While the debate of test harness versus debugging tools was not discussed, there's no doubt that a combination of testing and leveraging the debugging knowledge imparted by this book will yield positive results.

Return to Table of Contents

Top 5 Tips for Getting a Jump on IT Spending for 2010

by Siamak Farah

1. **The OS Is Irrelevant!**

As the battle of operating systems (OS) wages on between Apple and Microsoft, many businesses feel caught in the middle, unclear about which system to choose. Once a side has been chosen, there is still the ever-present (and recurring) dilemma over which version to choose — not to mention the potential nightmare of migration! (e.g., Should we migrate from XP to Windows 7; What pitfalls, if any, might we encounter?, etc...). Consider, instead, going OS neutral. With the growing popularity of Web-delivered software (also referred to as Software as a Service — or SaaS), companies can relieve themselves of a tremendous headache by relying on experts who deliver always-up-to-date applications via a simple Web browser. This path allows employers to avoid worries over software updates, plus, you have the added benefit of being able to "take your desktop with you" (as you can login to your desktop from any computer in the world with a browser and Web access).

2. Ditch the Servers

Perhaps the most significant line item of any IT budget is the costs of hardware (servers); And the hidden cost associated with this occurs when the IT department is pressured to estimate the right size. Assuming a large growth path, many servers must be ordered in advanced to be ready to support the growth. Should downsizing be in the cards, then one needs to plan on decommissioning servers which are hard to dispose of, as they often are worth a fraction of their purchase price. SaaS takes the guesswork out of your budget. In the same fashion that one does not think about the cell phone infrastructure and just orders or decommissions cell phones based on the number of employees, IT managers, can always have the right amount of server power and be poised for growth with SaaS providers.

3. Give Your Employees a (Virtual) Key to the Office

The average American now works longer hours than even our overseas counterparts. If your company makes use of next-generation SaaS tools, your employees can have anytime/anywhere access to their desktop, allowing them to work remotely and during off-hours if that is what is necessary to get the job done. We've found that by making remote access to all aspects of the work environment easy for our employees, they have become infinitely more efficient — many log-in to check for urgent issues before starting their morning commute and check-in again in the evening — from home. This type of employee dedication can help propel a company from being just a player in their industry to being "the player."

4. To Thy Own Client Be True

Okay, perhaps that isn't how the saying actually goes however the sentiment is valid. In this day of aggressive competition, it's important to use every tool and advantage you can afford to keep in touch with your clients (and have a reliable means for including personal details and generating automated follow-up reminders). CRM (customer relationship management) software is not new, however the leaders in this industry charge more than a pretty penny for their tools. Consider one of the "optimally-sized" versions (such as StreetSmart's Web-based CRM) that can be literally a fraction the cost and which offer the core functionality you need. Don't be left without such a valuable tool just because you've heard CRM software can be too price prohibitive.

5. Automate Your Protection

Every industry has its own set of compliance rules and best business practices. However, many companies overlook one of the most basic — yet most crucial — practices: email archiving. This simple step can offer tremendous piece of mind and protection. Investigate automatic email archiving software which has the potential to serve as the most affordable business insurance you have ever had. Such software works invisibly in the background to back-up all employee email, protecting your company from accidental or intentional email deletion.

— Siamak Farah is CEO of InfoStreet (www.infostreet.com/).

Return to Table of Contents

What the iPad Really Is

It's all about the user

When Steve Jobs demo'd the iPad, I thought I understood what he was presenting, and I was surprised at some of the reactions. I thought Jobs was pretty clear, and yet most people seemed to be misunderstanding, stubbornly trying to fit this device into a category that is wasn't intended for.

My friend Daniel Steinberg got it. He pointed out that, instead of watching the screen on the right side of the stage during that boring display of visiting web sites and sending email, you needed to be looking at the left side of the stage, where Steve was sitting in a comfortable chair playing the role of a user, an information consumer. It wasn't a demo of what the device does, it was a demo of the user experience.

But I wasn't satisfied. There was a phrase that the device put in my mind, a phrase that I wasn't seeing in the blogs and articles that followed the press event. The phrase was, "The iPad is a consumer device."

So I Googled "iPad consumer device," and immediately saw my thesis spelled out clearly on D. Sivakumar's blog.

This is a device for consuming information. Not for creating it. Apple has decided that the era of the personal computer is over. It was a fun 30-year run, but the idea that we all need a single omnipurpose information processing device has had its day. A tool that is optimized for developing software (or for creating video or music or other media content) is not the optimal tool for consuming information: movies, books, magazines, music, web content, email, tweets, chat, voicemail. Apple is betting that we, a sufficient number of us, will decide that we have sufficient need for an information consumer device in addition to whatever other devices we think we need.

An information consumer device needs some means for entering data, because even consuming information today is interactive. But it doesn't need a writer's keyboard, just an email user's keyboard. It doesn't need a lot of things that we think a personal computer needs, because it isn't one.

Anyway, that's my interpretation (and more or less D. Sivakumar's, I guess). My guess is that Apple is right, and will sell millions of these this year. I plan to buy one. So that's one right there.

Speaking of guesses, I recently asked you to guess what Apple was going to announce at that highly anticipated event. I limited the guessing game to a few fairly objective and unambiguous (I thought) questions. I asked:

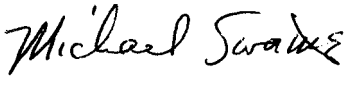
- Price: Less than $500, $500-699, $700-999, four figures.
- Processor: ARM, Intel, something else.
- Operating System: Basically the iPhone OS, basically the Mac OS, something significantly different.
- Screen Size: 7-inch diagonal, 10-inch diagonal, something else.
- DK: Full SDK available on delivery or not.
- Name: Something involving Slate, something involving Tablet, something involving Book, something else.
- Ship Date: Before March 1, in March, after April 1.
- Number sold in 2010: Your guess.

So how did *Dr. Dobb's* readers do?

Since Apple announced a range of prices and two ship dates, I had to be generous in scoring those answers. I accepted either 9 or 10 inches for screen size, and nearly all of you got that right. You also did well on price, ship date, and SDK availability. Not a single one of you identified the CPU, although a minority answered the question correctly by saying "something else." Apple's answers are at http://www.apple.com/ipad/specs/.

Dogorzaly wins a no-prize for answering just one of the questions and getting it exactly right: he or she correctly predicted that it would be called the iPad. Robin Kimzy precisely identified the price that Steve put up on the screen at the demo: $499. And Daryl Desmond was the only participant to go six for six (I didn't score the last question since we don't know about 2010 sales yet).

Congratulations to all our no-prize winners!



By Michael Swaine

— Michael Swaine is a senior contributing editor to Dr. Dobb's *and editor of PragPub.*

Return to Table of Contents