The Microsoft Journal for Developers

msdn® magazine

SEPTEMBER 2009 VOL 24 NO 9

msdn

Microsoft®

Accelerometers ◆ Synchronization Coverage ◆ Parallel Debugging ◆ Event Tracing for Windows
MSDN Magazine
SEPTEMBER 2009 Vol 24 No 9

Microsoft msdn magazine
SEPTEMBER 2009 VOLUME 24 NUMBER 9

LUCINDA ROWLEY Director

EDITORIAL: mmeditor@microsoft.com
HOWARD DIERKING Editor-in-Chief
NANCY MICHELL Senior Editor
TERRENCE DORSEY Senior Editor

WEB SITE
MICHAEL RICHTER Webmaster

CONTRIBUTING EDITORS Don Box, Keith Brown, Dino Esposito, Juval Lowy, Dr. James McCaffrey, Fritz Onion, John Papa, Ted Pattison, Charles Petzold, Jeff Prosise, Jeffrey Richter, John Robbins, Aaron Skonnard, Stephen Toub

MSDN Magazine (ISSN # 1528-4859) is published monthly by TechWeb, a division of United Business Media LLC., 600 Community Drive, Manhasset, NY 11030 516-562-5000. Periodicals Postage Paid at Manhasset, NY and at additional mailing offices. Back issue rates: U.S. $10. All others: $12. Basic one-year subscription rates: U.S. $45. Canada and Mexico $55. Registered for GST as TechWeb, a division of United Business Media LLC., GST No. R13288078, Customer No. 2116057 Canada Post: Publications Mail Agreement #40612608. Canada Returns to be sent to Bleuchip International, P.O. Box 25542, London, ON N6C 6B2. All other foreign orders $70, must be prepaid in U.S. dollars drawn on a U.S. bank. Circulation Department, MSDN Magazine, P.O. Box 1081, Skokie, IL 60076-8081, fax 847-763-9583. Subscribers may call from 8:00 AM to 4:30 PM CST M-F. In the U.S. and Canada 888-847-6188; all others 847-763-9606. U.K. subscribers may call Jill Sutcliffe at Parkway Gordon 01-49-1875-386. Manuscript submissions and all other correspondence should be sent to MSDN Magazine, 6th Floor, 1290 Avenue of the Americas, New York, NY 10104. Copyright © 2009 Microsoft Corporation. All rights reserved; reproduction in part or in whole without permission is prohibited.

POSTMASTER: Send address changes to MSDN Magazine, P.O. Box 1081, Skokie, IL 60076-8081

READERS: Order, renew, or make payment for subscriptions; order back issues; and submit customer service inquiries via the Web at http://msdn.microsoft.com/msdnmag/service.

PUBLISHER:
Jill Thiry jthiry@techweb.com
650-728-7368

United Business Media

BPA WORLDWIDE CONSUMER

ADVERTISING SALES:
Ed Day Western US Accounts/eday@techweb.com/785-838-7547
Brenner Fuller Eastern US Accounts/bfuller@techweb.com/603-746-3057
Michele Hurabiell Key Accounts/mhurabiell@techweb.com/415-378-3540
Wright's Reprints 877-652-5295

ONLINE SERIVCES:
Meagon Marshall Online Accounts & Programs/mmarshall@techweb.com/785-838-7524

MANUFACTURING:
Pete C. Scibilia Production Manager/pscibili@ubm-us.com/516-562-5134

MARKETING:
Jon Guerringue Marketing/jguerringue@techweb.com/516-562-5957
Laura Robison Marketing/lrobison@techweb.com/415-947-6182

AUDIENCE AND SUBSCRIBERS SERVICES:
Karen McAleer Audience Development Director/kmcaleer@techweb.com/516-562-7833
Andrew Athanasiou Audience Development Assistant/aathansiou@techweb.com/516-562-7981
SUBSCRIBER SERVICES: 800-677-2452

TechWeb, a division of United Business Media LLC.–The Global Leader in Business Technology Media
Tony L. Uphoff CEO
Bob Evans SVP and Content Director
Eric Faurot SVP, Live Events Group
Joseph Braue SVP, Light Reading Communications Group
John Siefert VP and Publisher, InformationWeek and TechWeb Network
Scott Vaughan VP, Marketing Services
John Ecke VP, Financial Technology Group
Jill Thiry Publisher, MSDN Magazine and TechNet Magazine
John Dennehy General Manager
Fritz Nelson Executive Producer, TechWeb TV
Scott Popowitz Senior Group Director, Audience Development
Beth Rivera Senior Human Resources Manager

Printed in the USA

EDITOR'S NOTE

Find Your Anchors

I recently returned to work after being gone for four weeks on paternity leave. My job during this time was to assume responsibility for our older daughter, who is now 2 years old, so that my wife could focus more of her attention on caring for the baby. Now, 2-year-olds have fairly little observable pattern or structure in how they want to spend their time. As such, managing the day-to-day feels a lot like stream processing or like managing your workday from your e-mail inbox—you're in a constant state of reacting. I'm sure that there are job roles out there where this pattern of work is considered normal and becomes easier to deal with. However, I found it to be incredibly draining—both physically and mentally.

As I write this, Windows 7 has recently been released to manufacturing and those of you who are MSDN subscribers will likely have already downloaded and installed it by the time you read this. Additionally, Visual Studio 2010 and Office 2010 continue to march toward release; and with the recent announcement of pricing and licensing information on the Windows Azure platform (microsoft.com/azure/pricing.mspx), we can expect to see increased interest in how cloud computing concepts and technologies should fit into the portfolio, from both a technology and a business perspective. Without very deliberate management, you can easily find yourself in a position much like my recent parental leave experience—in a constant state of reaction and working incredibly hard only to wonder what was accomplished at the end of the day.

To be honest, I have some experience with becoming randomized on the technology front as well. One of the great things about my role with the magazine is that it gives me visibility into all of the various development efforts that are happening across Microsoft. The problem with this visibility is that I want to go deep into many of the technologies to which I'm exposed. And what I've found, particularly over the past year, is that—with one exception—I have become good at developing a shallow level understanding of a large number of technologies. And while I believe that being a generalist is a good thing to a point, as the number of metaphorical e-mails increases, it becomes even more necessary to have a solid anchor technology (or a small handful of technologies) where you can go deep. Of equal importance is, I believe, making sure that you are going deep on the right technologies. Thus, you should define and prioritize your anchor technologies in terms of what you want to be known for, and also be aware of what's coming and plan accordingly.

My wakeup call came when I started experimenting with the .NET Services stack and, more recently, the ADO.NET Data Services stack. In both of those endeavors, I very quickly got past the controlled sample applications and ran straight into a fundamental deficiency in my own skillset—Windows Communication Foundation (WCF). As you know, this technology has been around for quite some time. However, for whatever reason, I had simply never given it the focus necessary to really understand how it was used behind the scenes in the cloud and with RESTful services. Without this context, I was handicapped in my code experimentation once I ventured too far off the known path. Personally, I want to grow as an expert in building business intelligence architectures—and this means that my current anchor technology priorities include SQL Server Analysis Services and WCF. As I look into the future, I envision moving further into data visualization and, as such, I expect to add Windows Presentation Foundation to my list.

In the end, it doesn't matter which specific technologies you choose so long as you are deliberate and thoughtful about actually choosing them. Otherwise, as the technology stream's current gets stronger, you risk being swept out into the sea of randomization and buzzwords.

[illegible]

THANKS TO THE FOLLOWING MICROSOFT TECHNICAL EXPERTS FOR THEIR HELP WITH THIS ISSUE: *Tom Ball, Ben Constable, Jim Griesmer, Niklas Gustafsson, Melissa Johnson, Artur Laksberg, Mark Lawrence, Carlos Aguilar Mares, Matthew M. Osborn, Josh Phillips and Matthew Podwysocki.*

Visit us at msdn.microsoft.com/magazine. Questions, comments, or suggestions for *MSDN Magazine*? Send them to the editor: mmeditor@microsoft.com.

Caching, Object-Object Mapping, Blogs and More

Improving Web Application Performance with Distributed Caching

For most data-driven Web applications, each page displays a variety of information from the database. For example, a typical Web page at Amazon.com shows details about a product and includes user reviews, related products, information about the shopper, and so on. Consequently, whenever a Web page is requested, the application must issue a cascade of queries to the database to retrieve the information displayed on the page. This "chatty" behavior works fine for Web applications with light traffic, but it does not scale well.

Caching is one of the most effective tools for reducing load and improving scalability in read-dominated, data-driven Web applications. ASP.NET includes a built-in caching API that uses an in-memory backing store and includes features like time-based expiries and file system and database dependencies. There's also the Caching Application Block in the Enterprise Library, which can be used outside of ASP.NET applications and offers greater flexibility in terms of how and where the cache data is stored. However, both the ASP.NET cache and the Caching Application Block store their cache data locally. This results in suboptimal performance in a Web farm environment because the data cached in one server is not accessible to other servers in the farm. One option is to designate a single server in the farm as the centralized cache server, have it store the only copy of the cache, and share it among the others. However, this approach introduces a single point of failure and a potential bottleneck.

Distributed caching overcomes the shortcomings of having several localized caches or resorting to a single, centralized cache store. In a nutshell, a distributed cache either replicates or partitions the cache store across multiple servers, providing a more efficient caching strategy in a Web farm environment.

A variety of distributed caching tools are available. One of the most popular is **memcached** (version 1.2.8), a free, open-source option created by Danga Interactive and used on high-profile sites like LiveJournal, Wikipedia, and SourceForge. Memcached has its roots in the LAMP stack (Linux, Apache, MySQL, PHP), but there are community-created Windows ports and .NET libraries available, along with open-source custom provider classes for integration with ASP.NET's session state. Microsoft is busy working on its own distributed caching library, code-named "Velocity," which at the time of this writing is available as a community technology preview. And there are also commercial distributed caching tools, such as **ScaleOut StateServer** and **ScaleOut SessionServer** by **ScaleOut Software**, and **NCache** by **Alachisoft**. (NCache was reviewed in the October 2007 issue: msdn.microsoft.com/en-us/magazine/cc163343.aspx.)

To get started with any distributed caching tool, you must first define the distributed cache's topology. With memcached, you simply start the memcached service or application on those computers that will store cache data, specifying parameters like the cache size via command-line switches. Velocity and most commercial offerings provide both command-line access and graphical user interfaces for creating and managing the topology.

Figure 1 **Typical "Get" Request**

```
Function GetUserProfile(UserID)
    UserProfile = Cache.Get(
      "UserProfile" + UserID)

    If UserProfile = NULL Then
        UserProfile = _
          Database.GetUserProfile(UserID)
        Cache.Add("UserProfile" +
          UserID, UserProfile)
    End If

    Return UserProfile
End Function
```

The patterns for reading from and writing to the distributed cache are no different from those used with ASP.NET's built-in caching API. Both types of caches act as a giant hashtable, where each item in the cache is referenced by a unique string. The pseudocode in **Figure 1** shows how data is read from the cache. When the "Get" statement is executed, the distributed cache library determines where the cached item exists in the topology and retrieves the data. Note that the client application cannot assume that the data exists in the cache because it may have expired, been removed by user code or because of a dependency, or been evicted because of memory constraints. Instead, the client application must always check whether data was returned from the cache. If the item is not found, then it must be re-retrieved and re-added to the cache.

Whenever data is updated, any references to that data in the cache become

Send your questions and comments for Scott to toolsmm@microsoft.com.

All prices confirmed at press time and are subject to change. The opinions expressed in this column are solely those of the author and do not necessarily reflect the opinions at Microsoft.

Figure 2 **Typical "Update" Request**

```
Function UpdateUserProfile(UserProfile)
    Database.UpdateserProfile(UserProfile)

    Cache.Update("UserProfile" +
      UserProfile.UserID, UserProfile)
End Function
```

outdated. To prevent showing stale data, all cached references must be removed or updated. **Figure 2** contains pseudocode that would run when a user visiting the site updates her profile. This method not only updates the database for each instance, but also updates the associated cache item with the new data. Other techniques for maintaining fresh data in the cache include expiries and cache dependencies.

Caching is an essential component in building a scalable, data-driven Web application. For large, heavily trafficked Web sites that use a Web farm, consider using a distributed cache to maximize performance. Tools like memcached, Velocity and others provide an easy-to-use API for working with the cache and encapsulate the low-level details of maintaining, updating and accessing a distributed cache.

memcached: danga.com/memcached

Velocity: msdn.microsoft.com/en-us/data/cc655792.aspx

Blogs of Note

Most of the technical blogs I subscribe to focus on the technologies I use on a day-to-day basis, including ASP.NET, AJAX, Web design and so forth. But I also make a point to find and read blogs from experts in other fields. To me, an expert is a person who has a wealth of knowledge and, more important, real-world experience and can share this wisdom in a way that's clear and meaningful—even to developers who are not well-versed in the technology.

Udi Dahan fits this description. Dahan is a speaker, trainer, and consultant on software architecture and design of distributed systems and has worked on numerous large-scale, service-oriented applications for enterprises. He shares his insights on his blog, on online resource sites and in *MSDN Magazine* (see msdn.microsoft.com/en-us/magazine/dd569749.aspx and msdn.microsoft.com/en-us/magazine/cc663023.aspx).

If you haven't visited Dahan's blog before, start with the "First time here?" page (udidahan.com/first-time-here), where you'll find his most popular articles, blog posts, interviews and webcasts on service-oriented architecture (SOA), domain models and smart client applications. Also check out the Articles section, udidahan.com/articles, which contains links to Dahan's published content. And when you're ready to implement a SOA, check out nServiceBus, Dahan's free, open-source messaging framework for .NET applications.

nServiceBus: nservicebus.com

Udi Dahan's Blog: udidahan.com/?blog=true

A Helpful Utility for Object-Object Mapping

An object-oriented application architecture models the real-world problem domain through the use of objects and through object-oriented principles like inheritance, modularity and encapsulation. A point-of-sale application would have classes like Employee, Customer, OrderItem and Product. **Figure 3** shows how these classes might be composed to model a Sale.

Udi Dahan - The Software Simplist; a blog on Service Oriented Architecture and Enterprise Softw - Internet Explorer provided by
http://www.udidahan.com/?blog=true
Google
Favorites
Udi Dahan - The Software Simplist; a blog on Ser...
Page
Safety
Tools
Udi Dahan – The Software Simplist
Enterprise Development Expert & SOA Specialist
Home
Blog
Consulting
Training
Articles
Speaking
About
Contact
First time here?
Click here for my best content.

The Fallacy Of ReUse

Posted in Architecture | Autonomous Services | EDA | OO | Web Services

This industry is pre-occupied with reuse.

Udi Dahan's Blog: **udidahan.com/?blog=true**

Figure 3 **Parts of the Sale Object**

```
public class Sale {
    public Employee SalesPerson { get; set; }
    public Customer Customer { get; set; }
    public IEnumerable<OrderItem> Items {
      get; set; }
}

public class OrderItem {
    public Product Product { get; set; }
    public int Quantity { get; set; }
    public decimal Discount { get; set; }
}
```

While this model works well within the application domain, it may be a less-than-ideal model for moving data outside of the domain. For example, imagine that our application included a Windows Communication Foundation (WCF) service that exposed sales data to a business partner. While the service could return a collection of Sale objects, those objects might contain more data than we care to expose. The Employee object or Customer object in the Sale object might include sensitive information, like Social Security numbers or payment details. The Products that comprise the OrderItems might include unimportant details that unnecessarily inflate the size of the transmitted payload.

A common technique for overcoming these issues is to define Data Transfer Objects (DTOs), such as SalesDTO, EmployeeDTO, CustomerDTO and so on. These DTOs would contain the precise set of properties to share. The code for the service would internally work with the domain models—Sale, Employee and so forth—but before returning the data, it would create the corresponding DTOs and populate them with the appropriate properties from the domain objects.

Writing the domain object to DTO mapping code is tedious. If you find yourself

Figure 4 **Product and ProductDTO Classes**

```
public class Product {
    public Guid ProductId { get; set; }
    public string ProductName { get; set; }
    public decimal UnitPrice { get; set; }
    public bool CanBeDiscounted { get; set; }
}

public class ProductDTO {
    public Guid ProductId { get; set; }
    public string ProductName { get; set; }
}
```

routinely writing such object-to-object mapping code, check out **AutoMapper** version 0.3.1. AutoMapper is a free, open-source utility that can automatically map one object onto another with as little as two lines of code.

To start, call the Mapper.CreateMap method and specify the source and destination types like so: Mapper.CreateMap<*SourceType*, *DestinationType*>(). This creates a MappingExpression object that defines the mapping between the two object types. If there are nested types (as there are in **Figure 3**), you would call CreateMap once for each type that needs to be mapped.

After creating the mapping, call Mapper.Map and pass it the source object: Mapper.Map<*SourceType*, *DestinationType*>(*sourceObject*). The Map method returns an instance of the destination object with its properties assigned to the corresponding members in the source.

Figures 4 and 5 illustrate an end-to-end example. **Figure 4** defines the two objects in this example: Product and ProductDTO. **Figure 5** shows code from our WCF service. Here we have a Product object that we need to map onto a ProductDTO object to return to the client. Note how this mapping is performed by AutoMapper's Mapper class with just two lines of code.

Figure 5 **AutoMapper's Mapper.Map Method Creates a New ProductDTO Object Based on the Specified Product**

```
public ProductDTO GetProduct(Guid productId) {
    Product product = Product.Load(productId);

    // Map Product onto a ProductDTO object
    Mapper.CreateMap<Product, ProductDTO>();
    ProductDTO productDto =
      Mapper.Map<Product, ProductDTO>(product);

    return productDto;
}
```

AutoMapper can also map between collections of one type to another, such as mapping a List of Product objects to an array of ProductDTO objects.

In the real world, it's not always possible to have the property names or property types neatly line up between the source and destination object types, but that's no problem for AutoMapper. With one line of code, you can project a property (or properties) in the source type to a differently named property in the destination type. If the mapped property types do not align, AutoMapper can automatically convert the source property type to the destination property type if there is a suitable type converter in the .NET Framework. If no such type converter exists, you can create a custom type converter.

Price: Free, open source

codeplex.com/AutoMapper

SCOTT MITCHELL, *author of numerous books and founder of 4GuysFromRolla.com, is an MVP who has been working with Microsoft Web technologies since 1998. Mitchell is an independent consultant, trainer and writer. Reach him at Mitchell@4guysfromrolla.com or via his blog at ScottOnWriting.net.*

What's New in the .NET Framework 4 Base Class Library

Just about everyone who uses Microsoft .NET uses the Base Class Libraries (BCL).When we make the BCL better, almost every managed developer benefits. This column will focus on the new additions to the BCL in .NET 4 beta 1.

Three of the additions have already been covered in previous articles—I'll start with a brief review of these:

- Support for code contracts
- Parallel extensions (tasks, concurrent collections, and coordination data structures)
- Support for tuples

Then, in the main part of the article, I'll talk about three new additions we haven't written about yet:

- File IO improvements
- Support for memory mapped files
- A sorted set collection

There isn't room to describe all of the new BCL improvements in this article, but you can read about them in upcoming posts on the BCL team blog at blogs.msdn.com/bclteam. These include:

- Support for arbitrarily large integers
- Generic variance annotations on interfaces and delegates
- Support for accessing 32-bit and 64-bit registry views and creating volatile registry keys
- Globalization data updates
- Improved System.Resourcesresource lookup fallback logic
- Compression improvements

We're also planning to include some additional features and improvements for beta 2 that you will be able to read about on the BCL team blog around the time when beta 2 ships.

Code Contracts

One of the major new features added to the BCL in the .NET Framework 4 is code contracts. This new library provides a language-agnostic way to specify pre-conditions, post-conditions and object invariants in your code. You'll find more information on code contracts in Melitta Andersen's August 2009 *MSDN Magazine* CLR Inside Out column. You should also take a look at the Code Contracts DevLabs site at msdn.microsoft.com/en-us/devlabs/dd491992.aspx and on the BCL team blog at blogs.msdn.com/bclteam.

Parallel Extensions

As multi-core processors have become more important on the client, and massively parallel servers have become more widespread, it's more important than ever to make it easy for programmers to use all of these processors. Another major new addition to the BCL in .NET 4 is the Parallel Extensions (PFX) feature that is being delivered by the Parallel Computing Platform team. PFX include the Task Parallel Library (TPL), Coordination Data Structures, Concurrent Collections and Parallel LINQ (PLINQ)—all of which will make it easier to write code that can take advantage of multi-core machines. More background on PFX can be found in Stephen Toub and Hazim Shafi's article "Improved Support For Parallelism In The Next Version Of Visual Studio" in the October 2008 issue of *MSDN Magazine*, available online at msdn.microsoft.com/en-us/magazine/cc817396.aspx. The PFX team blog at blogs.msdn.com/pfxteam is also a great source of information on PFX.

By providing tuples in the BCL, we are helping to better facilitate language interoperability.

Tuples

Another addition to the BCL in .NET 4 is support for tuples,which are similar to anonymous classes that you can create on the fly. A tuple is a data structure used in many functional and dynamic languages, such as F# and Iron Python. By providing common tuple types in the BCL, we are helping to better facilitate language interoperability. Many programmers find tuples to be convenient, particularly for returning multiple values from methods—so even C# or Visual Basic developers will find them useful. Matt Ellis's July 2009 *MSDN Magazine* CLR Inside Out column discusses the new support for tuples in .NET 4.

This column is based on a prerelease version of Microsoft .NET Framework 4. Details are subject to change.

Send your questions and comments to clrinout@microsoft.com.

File IO Improvements

One of the new additions to .NET 4 that we haven't written about in detail is the new methods on System.IO.File for reading and writing text files. Since .NET 2.0, if you needed to read the lines of a text file, you could call the File.ReadAllLines method, which returns a string array of all the lines in the file. The following code uses File.ReadAllLines to read the lines of a text file and writes the length of the line along with the line itself to the console:

```
string[] lines = File.ReadAllLines("file.txt");
foreach (var line in lines) {
  Console.WriteLine("Length={0}, Line={1}", line.Length, line);
}
```

You no longer have to wait until all of the lines are read into memory before you can iterate through lines.

Unfortunately, there is a subtle issue with this code. The issue stems from the fact that ReadAllLines returns an array. Before ReadAllLines can return, it must read all lines and allocate an array to return. This isn't too bad for relatively small files, but it can be problematic for large text files that have millions of lines. Imagine opening a text file of the phone book or a 900-page novel. ReadAllLines will block until all lines are loaded into memory. Not only is this an inefficient use of memory, but it also delays the processing of the lines, because you can't access the first line until all lines have been read into memory.

To work around the issue, you could use a TextReader to open the file and then read the lines of the file into memory, one line at a time. This works, but it's not as simple as calling File.ReadAllLines:

```
using (TextReader reader = new StreamReader("file.txt")) {
    string line;
    while ((line = reader.ReadLine()) != null) {
      Console.WriteLine("Length={0}, Line={1}", line.Length, line);
    }
}
```

In .NET 4, we've added a new method to File named ReadLines (as opposed to ReadAllLines) that returns IEnumerable<string> instead of string[]. This new method is much more efficient because it does not load all of the lines into memory at once; instead, it reads the lines one at a time. The following code uses File.ReadLines to efficiently read the lines of a file and is just as easy to use as the less efficientFile.ReadAllLines:

```
IEnumerable<string> lines = File.ReadLines(@"verylargefile.txt");
foreach (var line in lines) {
  Console.WriteLine("Length={0}, Line={1}", line.Length, line);
}
```

Note that the call to File.ReadLines returns immediately. You no longer have to wait until all of the lines are read into memory before you can iterate through the lines. The iteration of the foreach loop actually drives the reading of the file. Not only does this significantly improve the perceived performance of the code, because you can start processing the lines as they're being read, it's also much more efficient because the lines are being read one at a time. Using File.ReadLines has an added benefit in that it allows you to break out of the loop early if necessary, without wasting time reading additional lines you don't care about.

We've also added new overloads to File.WriteAllLines that take an IEnumerable<string> parameter similar to the existing overloads that take a string[] parameter. And we added a new method called AppendAllLines that takes an IEnumerable<string> for appending lines to a file. These new methods allow you to easily write or append lines to a file without having to pass in an array. This means if you have a collection of strings, you can pass it directly to these methods without having to first convert it into a string array.

There are other places in the BCL that would benefit from the use of IEnumerable<T> over arrays. Take the file system enumeration APIs, for example. In previous versions of the framework, to get the files in a directory, you would call a method like DirectoryInfo.GetFiles, which returns an array of FileInfo objects. You could then iterate through the FileInfo objects to get information about the files, such as the Name and Length of each file. The code to do that might look like this:

```
DirectoryInfo directory = new DirectoryInfo(@"\\share\symbols");
FileInfo[] files = directory.GetFiles();
foreach (var file in files) {
  Console.WriteLine("Name={0}, Length={1}", file.Name, file.Length);
}
```

There are two issues with this code. The first issue shouldn't surprise you; just as with File.ReadAllLines, it stems from the fact that GetFiles returns an array. Before GetFiles can return, it must retrieve the complete list of files in the directory from the file system and then allocate an array to return. This means you have to wait for all files to be retrieved before getting the first results and this is an inefficient use of memory. If the directory contains a million files, you have to wait until all 1,000,000 files have been retrieved and a million-length array has been allocated.

The second issue with the above code is a bit more subtle. FileInfo instances are created by passing a file path to FileInfo's constructor. Properties on FileInfo such as Length and CreationTime are initialized the first time one of the properties is accessed. When a property is first accessed, FileInfo.Refresh is called which calls into the operating system to retrieve the file's properties from the file system. This avoids a call to retrieve the data if the properties are never used, and when they are used, it helps to ensure the data isn't stale when first accessed. This works great for one-off instances of FileInfo, but it can be problematic when enumerating the contents of a directory because it means that additional calls to the file system will be made to get the file properties. These additional calls can hinder performance when looping through the results. This is especially problematic if you are enumerating the contents of a remote file share, because it means an additional round-trip call to the remote machine across the network.

In .NET 4, we have addressed both of these issues. To address the first issue, we have added new methods on Directory and DirectoryInfo that return IEnumerable<T> instead of arrays.

Like File.ReadLines, these new IEnumerable<T>-based methods are much more efficient than the older array-based equivalents. Consider the following code that has been updated to use .NET 4's DirectoryInfo.EnumerateFiles method instead of DirectoryInfo.GetFiles:

```
DirectoryInfo directory = new DirectoryInfo(@"\\share\symbols");
IEnumerable<FileInfo> files = directory.EnumerateFiles();
foreach (var file in files) {
  Console.WriteLine("Name={0}, Length={1}", file.Name, file.Length);
}
```

Unlike GetFiles, EnumerateFiles does not have to block until all of the files are retrieved, nor does it have to allocate an array. Instead, it returns immediately, and you can process each file as it is returned from the file system.

To address the second issue, DirectoryInfo now makes use of data that the operating system already provides from the file system during enumeration. The underlying Win32 functions that DirectoryInfo calls to get the contents of the file system during enumeration actually include data about each file, such as the length and creation time. We now use this data when initializing the FileInfo and DirectoryInfo instances returned from both the older array-based and new IEnumerable<T>-basedmethods on DirectoryInfo. This means that in the preceding code, there are no additional underlying calls to the file system to retrieve the length of the file when file.Length is called, since this data has already been initialized.

Together, the new IEnumerable<T>-based methods on File and Directory enable some interesting scenarios. Consider the following code:

```
var errorlines =
    from file in Directory.EnumerateFiles(@"C:\logs", "*.log")
    from line in File.ReadLines(file)
    where line.StartsWith("Error:", StringComparison.OrdinalIgnoreCase)
    select string.Format("File={0}, Line={1}", file, line);

File.WriteAllLines(@"C:\errorlines.log", errorlines);
```

This uses the new methods on Directory and File, along with LINQ, to efficiently find files that have a .log extension, and specifically lines in those files that start with "Error:". The query then projects the results into a new sequence of strings, with each string formatted to show the path to the file and the error line. Finally, File.WriteAllLines is used to write the error lines to a new file named "errorlines.log," without having to convert error lines into an array. The great thing about this code is that it is very efficient. At no time have we pulled the entire list of files into memory, nor have we pulled in the entire contents of a file into memory. No matter if C:\logs contains 10 files or a million files, and no matter if the files contain 10 lines or a million lines, the above code is just as efficient, using a minimal amount of memory.

Memory-Mapped Files

Support for memory-mapped files is another new feature in the .NET Framework 4. Memory-mapped files can be used to edit large files or create shared memory for inter-process communication (IPC). Memory-mapped files allow you to map a file into the address space of a process. Once mapped, an application can simply read or write to memory to access or modify the contents of the file. Since the file is accessed through the operating system's memory manager, the file is automatically partitioned into a number of pages that are paged in and paged out of memory as needed. This makes working with large files easier, as you don't have to handle the memory management yourself. It also allows for complete random access to a file without the need for seeking.

Memory-mapped files can be created without a backing file. Such memory-mapped files are backed by the system paging file (only if one exists and the contents needs to be paged out of memory). Memory-mapped files can be shared across multiple processes, which means they're a great way to create shared memory for inter-process communication. Each mapping can have a name associated with it that other processes can use for opening the same memory-mapped file.

To use memory-mapped files, you must first create a MemoryMappedFile instance using one of the following static factory methods on the System.IO.MemoryMappedFiles.MemoryMappedFile class:

- CreateFromFile
- CreateNew
- CreateOrOpen
- OpenExisting

After this, you can create one or more views that actually maps the file into the process's address space. Each view can map all or part of the memory mapped file, and views can overlap.

Using more than one view may be necessary if the file is greater than the size of the process's logical memory space available for mapping (2GB on a 32-bit machine). You can create a view by calling either the CreateViewStream or CreateViewAccessor methods on the MemoryMappedFile object. CreateViewStream returns an instance of MemoryMappedFileViewStream, which inherits from System.IO.UnmanagedMemoryStream. This can be used like any other Stream in the framework. CreateViewAccessor, on the other hand, returns an instance of MemoryMappedFileViewAccessor, which inherits from the new System.IO.UnmanagedMemoryAccessorclass. UnmanagedMemoryAccessor enables

New Methods

on System.IO.File

- public static IEnumerable<string> ReadLines(string path)
- public static void WriteAllLines(string path, IEnumerable<string> contents)
- public static void AppendAllLines(string path, IEnumerable<string> contents)

on System.IO.Directory

- public static IEnumerable<string> EnumerateDirectories(string path)
- public static IEnumerable<string> EnumerateFiles(string path)
- public static IEnumerable<string> EnumerateFileSystemEntries(string path)

on System.IO.DirectoryInfo

- publicIEnumerable<DirectoryInfo> EnumerateDirectories()
- publicIEnumerable<FileInfo> EnumerateFiles()
- publicIEnumerable<FileSystemInfo> EnumerateFileSystemInfos()

Figure 1 **Process 1**

```
using (varmmf = MemoryMappedFile.CreateNew("mymappedfile", 1000))
  using (var stream = mmf.CreateViewStream()) {
    var writer = new BinaryWriter(stream);
    writer.Write("Hello World!");

    varstartInfo = new ProcessStartInfo("process2.exe");
    startInfo.UseShellExecute = false;
    Process.Start(startInfo).WaitForExit();
  }
```

Figure 2 **Process 2**

```
using (varmmf = MemoryMappedFile.OpenExisting("mymappedfile"))
  using (var stream = mmf.CreateViewStream()) {
    var reader = new BinaryReader(stream);
    Console.WriteLine(reader.ReadString());
  }
```

random access, whereas UnmanagedMemoryStream enables sequential access.

The following sample shows how to use memory-mapped files to create shared memory for IPC. Process 1, as shown in **Figure 1**, creates a new MemoryMappedFile instance using the CreateNew method specifying the name of the memory mapped file along with the capacity in bytes. This will create a memory-mapped file backed by the system paging file. Note that internally, the specified capacity is rounded up to the next multiple of the system's page size (if you're curious you can get the system page size from Environment.SystemPageSize, which is new in .NET 4). Next, a view stream is created using CreateViewStream and "Hello Word!" is written to the stream using an instance of BinaryWriter. Then the second process is started.

Process 2, as shown in **Figure 2**, opens the existing memory mapped file using the OpenExisting method, specifying the appropriate name of the memory mapped file. From there, a view stream is created and the string is read using an instance of BinaryReader.

SortedSet<T>

In addition to the new collections in System.Collections.Concurrent (part of PFX), the .NET Framework 4 includes a new set collection in System.Collections.Generic, called SortedSet<T>. Like HashSet<T>, which was added in .NET 3.5, SortedSet<T> is a collection of unique elements, but unlike HashSet<T>, SortedSet<T> keeps the elements in sorted order.

SortedSet<T> is implemented using a self-balancing red-black tree that gives a performance complexity of O(log n) for insert, delete, and lookup. HashSet<T>, on the other hand, provides slightly better performance of O(1) for insert, delete and lookup. If you just need a general purpose set, in most cases you should use HashSet<T>. However, if you need to keep the elements in sorted order, get the subset of elements in a particular range, or get the min or max element, you'll want to use SortedSet<T>. The following code demonstrates the use of SortedSet<T> with integers:

```
var set1 = new SortedSet<int>() { 2, 5, 6, 2, 1, 4, 8 };
bool first = true;
foreach (var i in set1) {
    if (first) {
        first = false;
    }
    else {
        Console.Write(",");
    }
    Console.Write(i);
}
// Output: 1,2,4,5,6,8
```

The set is created and initialized using C#'s collection initializer syntax. Note that the integers are added to the set in no particular order. Also note that 2 is added twice. It should be no surprise that when looping through set1's elements, we see that the integers are in sorted order and that the set contains only one 2. Like HashSet<T>, SortedSet<T>'s Add method has a return type of bool that can be used to determine whether the item was successfully added (true) or if it wasn't added because the set already contains the item (false).

Figure 3 shows how to get the max and min elements within the set and get a subset of elements in a particular range.

Figure 3 **Getting Max, Min and Subset View of Elements**

```
var set1 = new SortedSet<int>() { 2, 5, 6, 2, 1, 4, 8 };

Console.WriteLine("Min: {0}", set1.Min);
Console.WriteLine("Max: {0}", set1.Max);

var subset1 = set1.GetViewBetween(2, 6);
Console.Write("Subset View: ");
bool first = true;
foreach (var i in subset1) {
    if (first) {
        first = false;
    }
    else {
        Console.Write(",");
    }
Console.Write(i);
}

// Output:
// Min: 1
// Max: 8
// Subset View: 2,4,5,6
```

The GetViewBetween method returns a view of the original set. This means that any changes made to the view will be reflected in the original. For example, if 3 is added to subset1 in the code above, it's really added to set1. Note that you cannot add items to a view outside of the specified bounds. For example, attempting to add a 9 to subset1 in the code above will result in an ArgumentException because the view is between 2 and 6.

Try It Out

The new BCL features discussed in this column are a sampling of the new functionality available in the .NET Framework 4. These features are available in preview form as part of the .NET Framework 4 beta 1, which is available for download along with Visual Studio 2010 beta 1 at msdn.microsoft.com/en-us/netframework/dd819232.aspx. Download the beta, try out the new functionality, and let us know what you think at connect.microsoft.com/VisualStudio/content/content.aspx?ContentID=12362. Be sure to also keep an eye on the BCL team blog for upcoming posts on some of the other BCL additions and an announcement on what's new in beta 2. ■

JUSTIN VAN PATTEN *is a program manager on the CLR team at Microsoft, where he works on the Base Class Libraries. You can reach him via the BCL team blog at blogs.msdn.com/bclteam.*

DINO ESPOSITO

Exploring ASP.NET 4.0—Web Forms and Beyond

ASP.NET is a stable and mature platform for building rich and powerful Web applications, so it's hard to imagine a new set of compelling features being added to it. But last fall, with the release of Service Pack 1 for ASP.NET 3.5, Microsoft refined the platform's built-in AJAX support and enhanced its productivity by shipping Dynamic Data controls, a new framework of components specifically designed to address the needs of data-driven and data-entry applications.

In parallel, Microsoft developed a brand-new, alternative programming model called ASP.NET MVC. Unlike the classic Web Forms model, ASP.NET MVC helps developers create Web applications in accordance with a widely recognized design pattern: the Model View Controller.

Today, the overall ASP.NET platform is made up of a few distinct components: Web Forms, ASP.NET MVC, Dynamic Data controls and ASP.NET AJAX. The upcoming ASP.NET 4.0 platform has the same foundation as the latest 3.5 SP1 version, but it provides further refinement in the areas of Web Forms, Dynamic Data controls and, last but not least, ASP.NET AJAX.

In this article, I'll take a look at what's new and improved in the Web Forms model. In future columns, I'll address the Dynamic Data control platform as a whole and explore in-depth the developments in the ASP.NET AJAX environment.

ASP.NET Web Forms 4.0 at a Glance

The key words to describe what's new in the overall ASP.NET 4.0 platform are "more control." ASP.NET 4.0 is neither a revolutionary change nor a refactoring of its existing architecture. It consists, instead, of a good number of small-scale changes that together provide developers with much more control of certain frequently used features of the existing framework.

For example, ASP.NET 4.0 Web Forms give developers more control over viewstate management, generation of IDs in the context of data-bound controls, and HTML generated by some template-based controls. In addition, you'll find new families of pluggable components for features that weren't supporting the provider model in earlier versions of ASP.NET and a finer control over the linking of external script files through the ScriptManager control. Let's start with viewstate management.

This column is based on a prerelease version of ASP.NET 4.0. Details are subject to change.

Send your questions and comments for Esposito to cutting@microsoft.com.

More Control Over the Viewstate

I say nothing new by stating that the viewstate has been one of the most controversial features of ASP.NET since the advent of the platform. Too many developers are still convinced that the viewstate is a waste of bandwidth and an unacceptable burden for each and every ASP.NET page. Nearly the same set of developers eagerly welcomed ASP.NET MVC because of its complete absence of viewstate. Recently, while teaching an ASP.NET MVC class, I discussed a master/detail scenario in which the user could select a customer from a list to see more details. I populated the list during the loading of the page, as expected. Next, while handling the selection-changed event, I showed how to fill in the customer's details. However, to have the list available for another selection, I also had to explicitly repopulate it.

Students promptly noted the extra work required to fill in the list at every server action. Couldn't this be automatically filled as in Web Forms? Well, in ASP.NET Web Forms you don't need to refill data-bound controls over a postback just because of the viewstate.

In short, the viewstate is not there only to reduce your bandwidth. The viewstate is functional to the Web Forms model, as it caches some of the content for the controls in the page. Next, the ASP.NET infrastructure takes care of reading that information from the viewstate to restore the last known good state for each control within the page.

As widely known, but also largely overlooked, the viewstate is an optional feature. The viewstate support is turned on for each page by default, but developers have a Boolean property available to change the default setting and do without it. The property is named EnableViewState and is defined on the System.Web.UI.Control class. It should be noted that the System.Web.UI.Page class inherits from the Control class. As far as the viewstate is concerned, an individual control and the page are one and the same.

For any ASP.NET page, turning off the viewstate couldn't be easier. You set EnableViewState to false either declaratively or programmatically during the page's life cycle, as follows:

```
void Page_Load(object sender, EventArgs e)
{
    // Disable viewstate for the page
    // and ALL of its child controls
    this.EnableViewState = false;
    ...
}
```

The EnableViewState property indicates whether the control is allowed to cache its own UI-related state. Be reminded that the viewstate setting in ASP.NET has a hierarchical nature, which means that if the viewstate is enabled on the parent control, it cannot be disabled on any of its child controls. The following code has no effect on the behavior of the text box if the viewstate is enabled at the page or container level:

```
protected void Page_Load(object sender, EventArgs e)
{
  TextBox1.EnableViewState = false;
}
```

The IsViewStateEnabled property—a protected property indeed—reports about the current state of the viewstate for a control. But what does all of this mean to developers?

If the viewstate is enabled on the page (which is the default setting), you have no means to keep the state of individual controls off the storage. To gain some control over it in ASP.NET 3.5, you need to disable the viewstate at the page level and then re-enable it where needed, but also keep in mind the hierarchical nature of it. Any container control that has the viewstate enabled will inevitably push its setting down to the list of its children. This fact leads to somewhat of a paradox: it is possible for the same control to have the property IsViewStateEnabled set to true and the property EnableViewState set to false!

The viewstate is a fundamental piece of the ASP.NET Web Forms architecture, and dropping it entirely from the platform in the name of a performance gain is arguably not the best option. Years of experience has proved that a more sustainable option is having the viewstate disabled by default on the page. Even better than that is the change coming up with ASP.NET 4.0: enabling viewstate control over individual controls.

In ASP.NET 4.0, the System.Web.UI.Control class exposes a new property named ViewStateMode:

```
public virtual ViewStateMode ViewStateMode { get; set; }
```

The property takes its feasible values from the ViewStateMode enumerated type, shown in **Figure 1**.

Figure 1 **The ViewStateMode Enumeration**

Value	Description
Inherit	Inherits the value of ViewStateMode property from the parent control.
Enabled	Enables viewstate for this control even if the parent control has viewstate disabled.
Disabled	Disables view state for this control even if the parent control has viewstate enabled.

To preserve compatibility, the default value of the property is Inherit. However, this property gives developers the possibility of controlling the viewstate setting for individual controls, regardless of the viewstate option of any parent page or containers.

Only a few of the controls in ASP.NET pages really need viewstate. For example, all text boxes you use to simply gather text don't need viewstate at all. If Text is the only property you use on the control, then the value of the Text property would be preserved by cross-page postbacks because of posted values. Any value stored in the viewstate, in fact, will be regularly overwritten by the posted value. In this case, the viewstate is really unnecessary. But there's more to notice. Any side properties that are set to a given non-default value on creation, but remain unchanged during postbacks (such as ReadOnly, BackColor and so on), have no need to go to the viewstate. For example, a Button control that retains the same caption all the time doesn't need the viewstate at all. Up until ASP.NET 4.0, turning off the viewstate for individual controls was problematic. Things change in the next version. This is the key takeaway of the ViewStateMode property.

More Control over Auto-Generated IDs

In ASP.NET pages, using the same ID for two server controls isn't allowed. If this happens, the page won't compile—period. In HTML, though, it is possible for two or more elements to share the same ID. In this case, when you make a search for an element via document.getElementById, you'll simply get an array of DOM elements. What about nested ASP.NET controls?

Most data-bound, template-based controls generate their output by repeating an HTML template for every data-bound item. This means that any child control defined with a unique ID in the template is being repeated multiple times. The original ID can't just be unique. For this reason, since the beginning, the ASP.NET team defined an algorithm to ensure that every HTML element emitted by an ASP.NET control could be given a unique ID. The ID resulted from the concatenation of the control ID with the ID of the naming container. Furthermore, in case of repeated controls (i.e., templates), a numeric index was added for disambiguation. For years, nobody really cared about the readability of the auto-generated ID, and strings like the following became fairly common:

```
ctl00$ContentPlaceHolder1$GridView11$TextBox1
```

Looking at this, the first issue that may come to mind is the potential length of the string, which, repeated for several elements, makes the download larger. More important, such an approach is problematic from a client script perspective. Predicting the ID of a given control you want to script from the client is difficult, and leads straight to hard-to-read code. In the first place, you need to know the detailed name being generated for a given HTML element that is buried in the folds of a grid or any other naming container controls. Second, this name is subject to change as you rename one of the server controls along the hierarchy. A frequently used trick is the following:

```
var btn = document.getElementById("<% =Button1.ClientID %>");
```

The trick consists in using ASP.NET code blocks to insert in the HTML source code being generated for the real client ID of a given control. When master pages or template-based controls are used, the trick is a real lifesaver, because in this case, the naming container of a plain control ends up being quite a complex hierarchy of controls that developers often leave unnamed. The actual name of the control, therefore, is subject to a few autogenerated IDs.

In addition to using code blocks, ASP.NET 4.0 supports another option: more control over the algorithm that generates the client ID of a server control. The System.Web.UI.Control class now features a brand new property named ClientIDMode. The property can assume a few predefined values as detailed in **Figure 2**.

Admittedly, the ID-generation algorithm, especially when master pages are involved, is not easy to understand. It is guaranteed to generate unique IDs, but ends up with strings that are really hard to predict. A more predictable option has been introduced primarily to be used with data-bound controls so that developers can easily guess the ID of, say, a Label control used to render a given property on the nth data item. In this case, you want the ID to reflect the hierarchy of the control (simplifying the naming container structure to the parent controls) but also a particular key value, such as the primary key. Consider the following code:

```
<asp:GridView ID="GridView1" runat="server"
     ClientIDMode="Predictable"
     RowClientIdSuffix="CustomerID">
  ...
</asp:GridView>
```

In this case, each row of the grid is identified by one or more columns in the data source with a trailing index. Here's an example:

```
Panel1_GridView1_ALFKI_1
```

The GridView is the sole control to support multiple columns. If you have multiple columns to concatenate, then you use the comma to separate names. The ListView will accept only one column, whereas the Repeater control will limit to trail a 0-based index and accept no column name.

Finally, note that the ClientIDMode property affects only the ID attribute of the resulting HTML element. By design, the name attribute remains unchanged.

View Controls Refinements

Most data-bound controls offer the ability to select a given displayed element—mostly a row. In previous versions of ASP.NET, the selection was stored as the index of the selected item within the page. This means that for pageable controls (such as the GridView control), the same selection made on, say, page one is retained on page two unless you programmatically reset the selection during the page/changing event.

In ASP.NET 4.0, the current selection on a data-bound control is tracked via the value on the data key field you indicate through the DataKeyNames property. To enable this feature, you use the new PersistSelection Boolean property and set it to true. The default value is false for compatibility reasons.

In addition, the FormView and ListView controls offer a little better control over their generated HTML markup. In particular, the FormView now accounts for a brand new RenderTable Boolean property. If you set it to false (the default is true), then no extra HTML table tags will be emitted and the overall markup will be easier to style via CSS. The ListView no longer needs a layout template in ASP.NET 4.0:

Figure 2 **The ClientIDMode Enumeration**

Value	Description
Legacy	Default value for the ClientIDMode property, indicates that the ID should be generated as in earlier versions of ASP.NET.
Static	ASP.NET doesn't make any attempt to scope the client ID. The ID is assigned "as-is," no matter the naming container's structure. Note that when this option is chosen, developers must ensure that no duplicated IDs exist, or know how to recognize and handle them.
Predictable	The ID is obtained by simply concatenating the ID of parent controls and ignoring master pages' parent elements. This option is designed to be used in conjunction with the RowClientIdSuffix property in the context of data-bound template controls.
Inherit	The control will use the same algorithm as its parent. Settings defined for the page and/or in the configuration file do matter.

```
<asp:ListView ID="ListView1" runat="server">
    <ItemTemplate>
        <% Eval("CompanyName")%>
        <hr />
    </ItemTemplate>
</asp:ListView>
```

The preceding code snippet is sufficient to repeat the content of the CompanyName column for each element in the data source.

HTML Enhancements

In the beginning, ASP.NET didn't offer much programmatic control over all possible tags of a Web page. For a long time, the title of a page missed an ad hoc property on the Page class. The same could be said for other popular features, like CSS files.

In ASP.NET 4.0, the Page class exposes two new string properties to let you set some common tags in the <head> section of a page. The two new properties are Keywords and Description. The content of these server properties will replace any content you may have indicated for the metatags as HTML literals.

The Keywords and Description properties can also be set directly as attributes of the @Page directive, as in the following example:

```
<%@ Page Language="C#"
    AutoEventWireup="true"
    CodeFile="Default.aspx.cs"
    Inherits="_Default"
    Keywords="ASP.NET, AJAX, 4.0"
    Description="ASP.NET 4.0 Web Forms" %>
```

In ASP.NET, when you invoke Response.Redirect, you return the browser an HTTP 302 code, which means that the requested content is now available from another (specified) location. Based on that, the browser makes a second request to the specified address, and that's it. A search engine that visits your page, however, takes the HTTP 302 code literally. This is the actual meaning of the HTTP 302 status code in that the requested page has been temporarily moved to a new address. As a result, search engines don't update their internal tables and when the user clicks to see your page, the engine

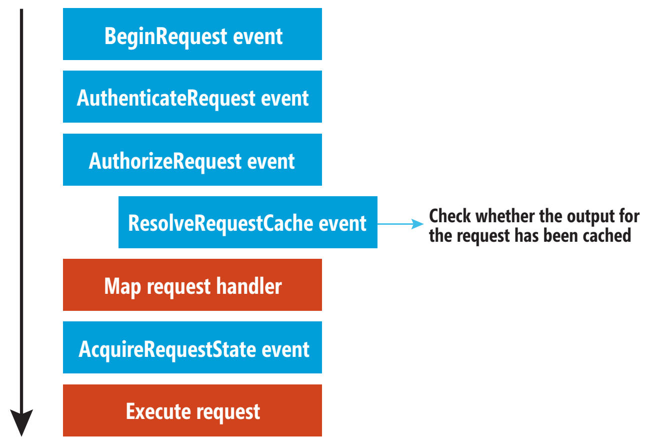

BeginRequest event
AuthenticateRequest event
AuthorizeRequest event
ResolveRequestCache event
Check whether the output for the request has been cached
Map request handler
AcquireRequestState event
Execute request

Figure 3 **Application Events and Output Caching**

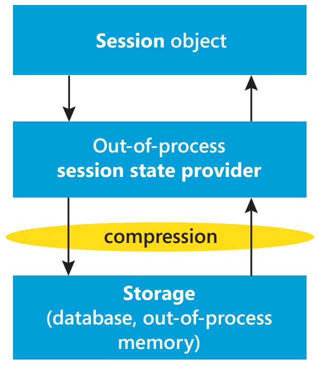

Session object
Out-of-process session state provider
compression
Storage (database, out-of-process memory)

Figure 4 **Compression in Out-of-Process Session State**

returns the original address. Next, the browser gets an HTTP 302 code and makes a second request to finally display the desired page.

To smooth the whole process, in ASP.NET 4.0 you can leverage a brand new redirect method, called RedirectPermanent. You use the method in the same way you use the classic Response.Redirect, except that this time, the caller receives an HTTP 301 status code. Code 301 actually means that the requested content has been permanently moved. For the browser, it makes no big difference, but it is a key difference for search engines.

Search engines know how to process an HTTP 301 code and use that information to update the page URL reference. Next time they display search results that involve the page, the linked URL is the new one. In this way, users can get to the page quickly and save themselves a second round-trip.

More Control over Output Caching

In ASP.NET 2.0, several key parts of the ASP.NET runtime were refactored and made much more flexible and configurable. This was achieved through the introduction of the provider model. The functionality of some ASP.NET core services, including session state, membership and role management, were abstracted to a sort of service contract so that different implementations of a given service could be used interchangeably and administrators could indicate the one they wanted from the configuration file.

For example, under the hood of the old faithful Session object, there's an instance of the HttpSessionState class whose content is retrieved by the selected session state provider. The default session state provider gets data from the in-process memory (specifically, from a slot within the Cache dictionary), but additional providers exist to store data in databases and external host processes.

In ASP.NET 4.0, the provider model covers another extremely important feature of ASP.NET that for some reason was left behind in previous versions: the output caching.

There are many situations where it is acceptable for a page response to be a little stale if this brings significant performance advantages. Think of an e-commerce application and its set of pages for a products catalog, for example. These pages are relatively expensive to create because they could require one or more database calls and likely some form of data join. Product pages tend to remain the same for weeks and are rarely updated more than once per day. Why should you regenerate the same page a hundred times per second? Since ASP.NET 1.0, output caching allows you to cache page responses so that following requests can be satisfied returning the cached output instead of executing the page. **Figure 3** shows the sequence of application events, includ-

Figure 5 **Split Script Files in the Microsoft AJAX Library for ASP.NET 4.0**

File	Goal
MicrosoftAjaxCore.js	Core functionalities of the library
MicrosoftComponentModel.js	Base classes for the client-side object model
MicrosoftAjaxSerialization.js	JSON serialization
MicrosoftAjaxGlobalization.js	Data and functions for globalization
MicrosoftAjaxHistory.js	History points and the client API for history management
MicrosoftAjaxNetwork.js	Raw AJAX and HTTP functionalities
MicrosoftAjaxWebServices.js	Wrapper client API for invoking services
MicrosoftAjaxApplicationServices.js	Wrapper client API for predefined ASP.NET services such as authentication
MicrosoftAjaxTemplates.js	Client-side rendering API for ASP.NET AJAX 4.0
MicrosoftAjaxAdoNet.js	Wrapper client API for ADO.NET services in ASP.NET AJAX 4.0

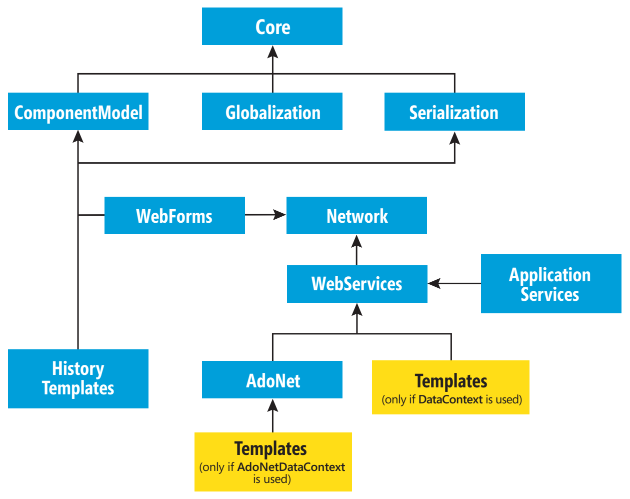


Figure 6 **Dependencies Between Split Scripts**

ing the step where the system attempts to resolve the request looking into the output cache.

Up until now, any page output (which can be grouped by form and query string parameters, requesting URL, or custom strings) is stored in memory in a private area of the ASP.NET Cache. In the long run, the amount of the output cache puts additional pressure on the Web server machine by consuming memory and generating frequent updates on the Cache object. In ASP.NET 4.0, the output caching subsystem fully supports the provider model, thus giving developers the opportunity of storing page responses outside the ASP.NET worker process.

A custom output cache provider is a class that derives from OutputCacheProvider. The name of the class must be registered in the configuration file, as shown below:

```
<caching>
  <outputCache defaultProvider="AspNetInternalProvider">
    <providers>
      <add name="DiskCacheProvider"
          type="Samples.DiskCacheProvider, MyProvider"/>
    </providers>
  </outputCache>
</caching>
```

As usual, you can have multiple providers registered and select the default one via the defaultProvider attribute on the outputCache node. The default behavior is offered through the AspNetInternalProvider object that turns out to be the default provider, if you don't change anything in the configuration file. The output cache provider doesn't have to be the same for all pages. You can choose a different provider on a per-request basis or even for a particular user control, page, or combination of parameters for a page. You can specify the provider name in the @OutputCache directive wherever this directive (page and/or user controls) is accepted:

```
<% @OutputCache Duration="3600"
                VaryByParam="None"
                providerName="DiskCache" %>
```

To change the provider on a per-request basis, instead, you need to override a new method in global.asax, as shown below:

```
public override string GetOutputCacheProviderName(HttpContext context)
{
   // Decide which provider to use looking at the request
   string providerName = ...;

   return providerName;
}
```

Starting with version 2.0, session state can be stored outside the memory of the worker process. This means that any data stored in the Session object must be serialized to and from an out-of-process environment. If you look back at **Figure 3**, session state is loaded into the Session object around the AcquireRequestState application event. The in-memory content is then serialized back to storage at the end of the request processing.

ASP.NET 4.0 enables developers to request some compression to be applied to the data stream being sent in and out of the session provider (see **Figure 4**). Compression is obtained by using the GZipStream class instead of a plain Stream class to do any serialization:

```
<sessionState mode="SqlServer" compressionEnabled="true" ... />
```

To enable compression, you simply add the compressionEnabled attribute to the sessionState section of the configuration file. By default, compression is not enabled.

Any JavaScript library is made up of myriad specific JavaScript files with a more or less sophisticated graph of interconnections. ASP.NET AJAX, instead, always tried to abstract JavaScript details away from developers and offered a rather monolithic client JavaScript library, via the ScriptManager control. This will change in ASP.NET 4.0. The Microsoft AJAX client library has been refactored and split into the individual files listed in **Figure 5**. **Figure 6** shows the dependencies between individual script files in the overall library. A new property has been added to the ScriptManager control that lets you specify how the building blocks of the library should be handled. The property is MicrosoftAjaxMode, and it accepts values as shown in **Figure 7**.

Figure 7 **The MicrosoftAjaxMode Enumeration**

Value	Description
Enabled	Legacy option, indicates that just one file will be linked from the page as in previous versions of ASP.NET.
Disabled	No script file is automatically referenced from the ScriptManager control.
Explicit	Script files to be referenced are explicitly listed in the Scripts section of the ScriptManager control.

Better Platform

ASP.NET 4.0 Web Forms contains a number of small-scale changes that, taken together, make it a better development platform. Web Forms is a mature framework that needs refinement, not redesign. If you don't feel entirely comfortable with Web Forms and are looking for something completely different, then ASP.NET MVC is worth a careful look. ■

Dino Esposito *is an architect at IDesign and co-author of "Microsoft .NET: Architecting Applications for the Enterprise" (Microsoft Press, 2008). Based in Italy, Esposito is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos.*

TED NEWARD

Concurrency with Channels, Domains and Messages

In the last article, we looked at the programming language Cobra, an object-oriented language loosely derived from Python that embraces both static- and dynamic-typing, some "scripting" concepts similar to those found in Python and Ruby, and baked-in unit-testing features, among other things. Upon deeper investigation, we saw that it had value and we were happy to have learned a new general-purpose programming language.

As interesting and powerful as general-purpose languages are, however, sometimes what's needed is not a hammer, saw or screwdriver, but a 3/16-inch nut driver; in other words, as much as we as developers value the tools that provide wide-ranging capability and facility, sometimes the right tool for the job is a very, very specific one. In this piece, we'll focus not on a language that re-creates all of the Turing-complete constructs that we're used to, but a language that focuses on one particular area, seeking to do it well, in this case, concurrency.

What developers find the easiest to do should be the right thing, and what developers find hard to do or impossible to do should be the wrong thing.

Careful and thorough readers of this magazine will note that this concurrency meme is not an unfamiliar one to these pages. The past few years have seen numerous articles on concurrency come through here, and *MSDN Magazine* is hardly unique in that respect—the blogosphere, Twitterverse and developer Web portals are all rife with discussions of concurrency, including a few naysayers who think this whole concurrency thing is another attempt to create a problem out of thin air. Some industry experts have even gone so far as to say (according to rumor, at a panel discussion at an OOPSLA conference a few years ago) that "concurrency is the dragon we must slay in the coming decade."

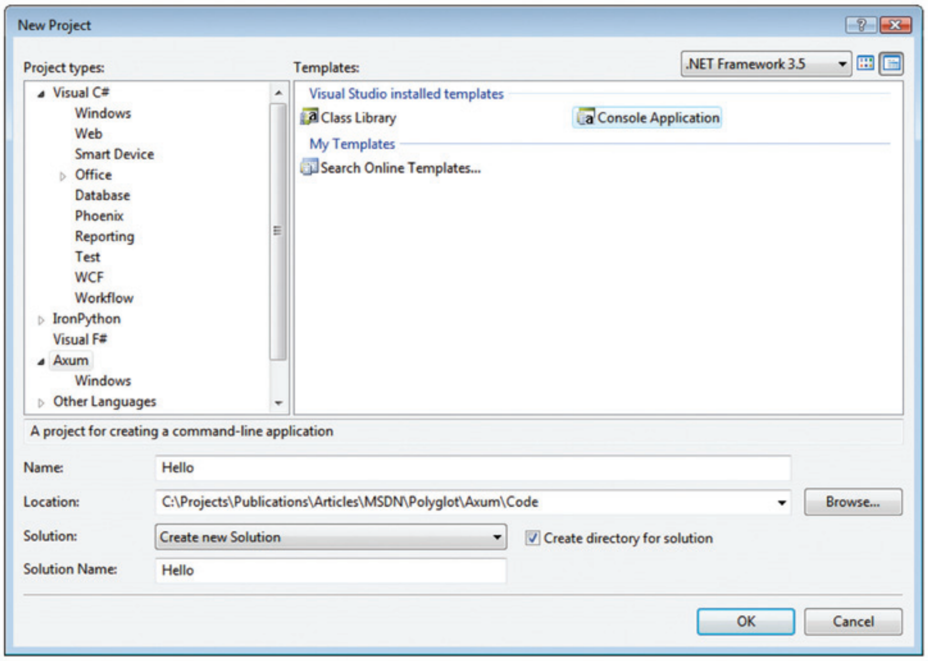
Some developers will be wondering what the problem really is—after all, .NET has had multi-threading capability for years, via the asynchronous delegate invocation idiom (BeginInvoke/EndInvoke), and provides thread-safety mechanisms via the monitor construct (the lock keyword from C#, for example) and other explicit concurrency object constructs (Mutex, Event, and so on, from System.Threading). So all this new fuss seems like making a mountain out of a molehill. Those developers would be absolutely right, assuming they write code that is (1) 100 percent thread-safe, and (2) taking advantage of every opportunity for task or data parallelism in their code by spinning up threads or borrowing threads from thread pools to maximize the use of all the cores provided by the underlying hardware. (If you're safe on both counts, please move on to the next article in the magazine. Better yet, write one and tell us your secrets.)

Numerous proposals to solve, or at least mitigate, this problem have been floated through the Microsoft universe, including the Task Parallel Library (TPL), Parallel FX extensions (PFX), F# asynchronous workflows, the Concurrency and Control Runtime (CCR), and so on. In each of these cases, the solution is either to extend a host language (usually C#) in ways to provide additional parallelism or to provide a library that can be called from a .NET general-purpose language. The drawback to most of these approaches, however, is that because they depend on the semantics of the host language, doing the right thing, from a concurrency perspective, is an elective choice, something developers have to explicitly buy into and code appropriately. This unfortunately means that the opportunity for doing the wrong thing, whatever that may be, thus creates a concurrency hole that code can ultimately fall into and create a bug waiting to be discovered, and this is a bad thing. Developer tools shouldn't, in the ideal, allow developers to do the wrong thing—this is why the .NET platform moved to a garbage-collected approach, for example. Instead, developer tools should cause developers to, as Rico Mariani (Microsoft Visual Studio architect and former CLR Performance architect) puts it, "fall into the pit of success"—what developers find the easiest to do should be the right thing, and what developers find hard to do or find impossible to do should be the wrong thing.

Toward that end, Microsoft has released as an incubation project a new language aimed squarely at the concurrency domain, called "Axum." In the spirit of enabling developers to "fall into the

Disclaimer: Axum is an incubation project. As such, Microsoft has made no commitment to ship Axum as a supported product. The details of Axum's language and operation are entirely subject to change to any degree and at any time.

Send your questions and comments for Ted to polyglot@microsoft.com.



New Project

Project types:
- Visual C#
  - Windows
  - Web
  - Smart Device
  - Office
  - Database
  - Phoenix
  - Reporting
  - Test
  - WCF
  - Workflow
- IronPython
- Visual F#
- Axum
  - Windows
- Other Languages

Templates: .NET Framework 3.5

Visual Studio installed templates
Class Library
Console Application
My Templates
Search Online Templates...

A project for creating a command-line application

Name: Hello
Location: C:\Projects\Publications\Articles\MSDN\Polyglot\Axum\Code
Browse...
Solution: Create new Solution
[✓] Create directory for solution
Solution Name: Hello

OK
Cancel

Figure 1 **Axum Console Application Project**

pit of success," Axum is not a general-purpose language like C# or Visual Basic, but one aimed squarely at the problem of concurrency, designed from the outset to be part of a suite of languages that collectively cooperate to solve a business problem. In essence, Axum is a great example of a domain-specific language, a language designed specifically to solve just one vertical aspect of a problem.

As of this writing, Axum is labeled as version 0.2, and the details of Axum's language and operation are entirely subject to change, as stated by the disclaimer at the front of the article. But between "as I write this" and "as you read this," the core concepts should remain fairly stable. In fact, these concepts are not unique to Axum and are found in a number of other languages and libraries (including several of the aforementioned projects, such as F# and the CCR), so even if Axum itself doesn't make it into widespread use, the ideas here are invaluable for thinking about concurrency. What's more, the general idea—a language specific to the problem domain—is quickly growing into a "big thing" and is worth examining in its own right, albeit in a later piece.

Beginnings

The Axum bits for download currently live on the Microsoft Labs Web site at msdn.microsoft.com/en-us/devlabs/dd795202.aspx; be sure to pick up at least both the .msi (either for Visual Studio 2008 or for Beta 1 of Visual Studio 2010) and the Programmer's Guide. Once you install these, fire up Visual Studio, and create a new Axum Console Application project, as shown in **Figure 1**.

Replace it with the code shown in **Figure 2**. This is the Axum "Hello World," which serves the same purpose as every other Hello World application does: to verify that the installation worked and to get a basic sense of the syntax. Assuming everything compiles and runs, we're good to go.

Figure 2 **Axum "Hello World"**

```
using System;
using Microsoft.Axum;
using System.Concurrency.Messaging;

namespace Hello
{
    public domain Program
    {
        agent MainAgent : channel Microsoft.Axum.Application
        {
            public MainAgent()
            {
                String [] args = receive(PrimaryChannel::CommandLine);

                // TODO: Add work of the agent here.
                         Console.WriteLine("Hello, Axum!");

                PrimaryChannel::ExitCode <-- 0;
            }
        }
    }
}
```

As you can see, the syntax of Axum is highly reminiscent of C#, putting Axum loosely into the category of languages known as the C family of languages: it uses curly braces to denote scope, semicolons to terminate statements, and so on. With that said, however, there are obviously a few new keywords to get to know (domain, agent, channel, receive, just for starters), plus a few new operators. Formally, Axum is not really a derivative of C#, but a selective subset with added new features. Axum includes all C# 3.0 statement types (such as lambdas and inferred local variables) and expressions (such as algebraic calculations and yield return/yield break), methods, field declarations, delegates, and enumeration types, but it leaves out classes, interfaces, or struct/value type definitions, operator declarations, properties, const fields/locals, and static fields/methods.

Hear that whooshing sound? That's the wind rushing past your ears, and if you feel a slight fluttering in your stomach, it's OK. Jumping off cliffs is a good thing.

Concepts and Syntax

The Axum syntax directly reflects the core concepts of Axum, just as it does for C#. Where C# has classes, however, Axum has agents. In Axum, an agent is an entity of code, but unlike a traditional object, agents never directly interact—instead, they pass messages to one another in an asynchronous fashion through channels, or to be more accurate, across the ports defined on the channel. Sending a message on a port is an asynchronous operation—the send im-

Figure 3 **Process Application**

```
using System;
using Microsoft.Axum;
using System.Concurrency.Messaging;

namespace Process
{
    public domain Program
    {
        agent ProcessAgent : channel Microsoft.Axum.Application
        {
            public ProcessAgent()
            {
                String [] args = receive(PrimaryChannel::CommandLine);

                for (var i = 0; i < args.Length; i++)
                            ProcessArgument(args[i]);

                PrimaryChannel::ExitCode <-- 0;
            }

            function void ProcessArgument(String s)
            {
                Console.WriteLine("Got argument {0}", s);
            }
        }
    }
}
```

mediately returns—and receiving a message on a port blocks until a message is available. This way, threads never interact on the same data elements at the same time, and a major source of deadlock and inconsistency is effectively lifted away.

To see this in action, consider the Hello code. The agent MainAgent implements a special channel, called Application, which is a channel the Axum runtime understands directly—it will process any incoming command line arguments and pass them over the Application's PrimaryChannel channel, on the port CommandLines. When constructed by the runtime, the MainAgent uses the receive operation (an Axum primitive) to block that port until those arguments are available, at which point execution in the MainAgent constructor continues. The runtime, having done its job, then blocks on the ExitCode port of the PrimaryChannel, which is the signal that the application is finished. Meanwhile, the MainAgent rolls through the array, printing each one, and once finished, sends the integer literal 0 to the ExitCode port. This is received by the runtime, which then terminates the process.

Notice how Axum, from the very beginning, is taking this concurrent execution idea seriously—not only are developers sheltered from the creation and manipulation of threads or concurrency/locking objects, but the Axum language assumes a concurrent approach even for something as simple as Hello. For something as simple as Hello, arguably, this is a waste of time—however, most of us don't write applications as simple as Hello very often, and those who do, can always fall back to the traditional object-oriented single-threaded-by-default application language (C# or Visual Basic or whatever else strikes your fancy).

Saying Hello Is for Wimps

A slightly more interesting example is the Process application, as **Figure 3** shows. In this version, we see a new Axum concept, the function. Functions are different from traditional methods in that the Axum compiler guarantees (by way of compiler error) that the method never modifies the agent's internal state; in other words, functions may not have side effects. Preventing side effects adds to Axum's usefulness as a concurrency-friendly language—without side effects, it becomes extremely difficult to accidentally write code where a thread modifies shared state (and thus introduces inconsistency or incorrect results).

But we're still not done here. Axum not only wants to make it easier to write thread-safe code, it also wants to make it easier to write thread-*friendly* code. With two- and four-core machines becoming more mainstream, and 8- and 16-core machines visible on the horizon, it's important to look for opportunities to perform operations in parallel. Axum makes this easier, again by lifting the conversation away from threads and providing some higher-level constructs to work with. **Figure 4** shows a modified version of the ProcessAgent.

Two new things are happening here: one, we've created an interaction point, which can serve as a source or a recipient of messages (in this case, it will be both); and two, we've used the forwarding operator to create a pipeline of messages from the interaction point to the ProcessArgument function. By doing this, any message sent to the "arguments" interaction point will be forwarded to ProcessArgument for , well , processing. From there, all that's left is to iterate through the command-line arguments and send each one as a message (via the send operator, the <-- operation inside the loop) to the arguments interaction point, and the work begins.

(Axum formally defines this as a data flow network, and there's

Figure 4 **Modified Version of the ProcessAgent**

```
using System;
using Microsoft.Axum;
using System.Concurrency.Messaging;

namespace Process
{
    public domain Program
    {
        agent ProcessAgent : channel Microsoft.Axum.Application
        {
            public ProcessAgent()
            {
                String [] args = receive(PrimaryChannel::CommandLine);

                // Second variation
                //
                var arguments = new OrderedInteractionPoint<String>();
                arguments ==> ProcessArgument;

                for (var i = 0; i < args.Length; i++)
                    arguments <-- args[i];

                PrimaryChannel::ExitCode <-- 0;
            }

            function void ProcessArgument(String s)
            {
                Console.WriteLine("Got argument {0}", s);
            }
        }
    }
}
```

much it can do in addition to creating a simple 1-to-1 pipeline, but that's beyond the scope of this introduction. The Programmer's Guide has more details for those who are interested.)

By now putting the data through the pipeline, Axum can decide how and when to spin up threads to do the processing. In this simple example, it may not be useful or efficient to do so, but for more sophisticated scenarios, it quite often will. Because the arguments interaction point is an ordered interaction point, the messages sent down the pipeline—and, more important, the results returned—will preserve their place in the ordering, even if each message is processed on a separate thread.

What's more, this concept of pipelining messages from one interaction point to another is a powerful concept that Axum borrows from functional languages like F#. **Figure 5** shows how easy it is to add some additional processing into the pipeline.

Notice that, again, the function UpperCaseIt is not modifying internal ProcessAgent state but is operating only on the object handed in. Additionally, the pipeline is modified to include UpperCaseIt before the ProcessArgument function, and the intuitive thing happens—the results of UpperCaseIt are passed into the ProcessArgument function.

As a mental exercise, consider how many lines of C# code would be required to do this, bearing in mind that all of this is also being done across multiple threads, not just a single thread of execution. Now imagine debugging the code to make sure it executes correctly. (Actually, imagining it isn't necessary—use a tool like ILDasm or Reflector to examine the generated IL. It's definitely not trivial.)

By the way, I have a small confession to make—as written, my sample doesn't execute correctly. When the preceding code is run, the application returns without displaying anything. This isn't a bug in the Axum bits; this is as intended behavior, and highlights how programming in a concurrent mindset requires a mental shift.

When the arguments interaction point is receiving the command-line arguments via the send operator (<--), those sends are done asynchronously. The ProcessAgent is not blocking when it sends those messages, and therefore if the pipeline is sufficiently complex, the arguments are all sent, the loop terminates, and the ExitCode is sent, terminating the application, all before anything can reach the console.

To fix this, ProcessAgent needs to block until the pipeline has completed operation; it needs to hold the thread alive with something like a Console.ReadLine(). (This turns out to be tricky in practice—see the Axum team blog for the details.) Or we need to change the way ProcessAgent works, and it's this latter course I'm going to take, primarily in order to demonstrate a few more of Axum's features.

Instead of doing the work inside itself, ProcessAgent will defer the work to a new agent. But now, just to make things a bit more interesting, this new agent will also want to know whether the argument should be uppercase or lowercase. To do this, the new agent will need to define a more complex message, one that takes not only the string to operate on, but also a Boolean value (true for uppercase). To do that, Axum requires that we define a schema.

Figure 5 **Pipelining Messages**

```
using System;
using Microsoft.Axum;
using System.Concurrency.Messaging;

namespace Process
{
    public domain Program
    {
        agent ProcessAgent : channel Microsoft.Axum.Application
        {
            public ProcessAgent()
            {
                String [] args = receive(PrimaryChannel::CommandLine);

                // Third variation
                //
                var arguments = new OrderedInteractionPoint<String>();
                arguments ==> UpperCaseIt ==> ProcessArgument;

                for (var i = 0; i < args.Length; i++)
                    arguments <-- args[i];

                PrimaryChannel::ExitCode <-- 0;
            }

            function String UpperCaseIt(String it)
            {
                return it.ToUpper();
            }

            function void ProcessArgument(String s)
            {
                Console.WriteLine("Got argument {0}", s);
            }
        }
    }
}
```

Process my Argument

Implicitly until this point, one of the goals of Axum has been to isolate different executing entities (agents) away from one another, and it has done so by making copies of the messages and handing those to the agent, rather than passing objects directly. No shared state—even through parameters—means no accidental chances of thread conflict.

While that works fine for types defined in the .NET Base Class Library, it can easily be defeated if .NET programmers don't do the right thing in user-defined types. To combat this, Axum requires that a new kind of message be defined as a schema type—essentially an object without the methods and with required and/or optional fields, using the schema keyword:

```
schema Argument
{
    required String arg;
    optional bool upper;
}
```

This defines a new data type, Argument, which consists of a required field, arg, containing the string that is to be made uppercase or lowercase, and an optional field, upper, indicating whether it should be made upper or lower. Now, we can define a new channel with a request/response port that takes Argument messages in and gives String messages back:

```
channel Operator
{
   input Argument Arg : String; // Argument in, String out
}
```

Having defined this channel, it becomes relatively trivial to write the agent that implements the channel, as shown in **Figure 6**. Notice that this agent technically never exits its constructor—it simply spins in a loop, calling receive on the Arg port, blocking until an Argument instance is sent to it, and then sending the result of the ToUpper or ToLower back on the same port (depending on the value of upper, of course).

From the callers' (users') perspective, using the OperatingAgent is no different than the ProcessAgent itself: we create an instance of it using the built-in method CreateInNewDomain, and start posting Argument instances to the Arg port. When we do, however, an object is returned that will eventually yield the response back from the agent, which is the only way to harvest the results (via another receive() operation), as **Figure 7** shows.

When run, it does as expected—based on the current millisecond at the time of the Argument sent, the command-line string will either be uppercase or lowercase. And, still, all without any direct thread interaction on the part of the developer.

So Little, yet So Far

Axum clearly represents a different way of thinking about programming, and despite its stark differences from C#, implements a significant number of features found in most general-purpose programming languages. Its main purpose, however, centers around concurrency and thread-friendly code, and as such, works best with problems that require (or benefit from) concurrency.

Fortunately, the team building Axum didn't try to reinvent the C# language. Because Axum compiles to .NET assemblies just as other .NET languages do, it's relatively trivial to build the concurrent-heavy parts of the application in Axum (as a Class Library project, instead of a Console Application), and then call into it from C# or Visual Basic. The Axum distribution ships with a sample that does this (the Dining Philosophers sample, illustrating the classic concurrency problem in a WinForms application), and quality time spent with Reflector over the Axum-compiled libraries will reveal a great deal about that interoperability footprint. Using Axum to build library libraries callable from another language (C# or Visual Basic) can go a long way toward making heavy concurrency use vastly more accessible to the other .NET technologies, such as Web or desktop applications.

Figure 6 **Agent Implementing Channel**

```
agent OperatingAgent : channel Operator
{
     public OperatingAgent()
     {
         while (true)
         {
             var result = receive(PrimaryChannel::Arg);
             if (result.RequestValue.upper)
                 result <-- result.RequestValue.arg.ToUpper();
             else
                 result <-- result.RequestValue.arg.ToLower();
         }
     }
 }
```

Figure 7 **Using Method CreateInNewDomain**

```
agent ProcessAgent : channel Microsoft.Axum.Application
{
  public ProcessAgent()
  {
    String [] args = receive(PrimaryChannel::CommandLine);

    var opAgent = OperatingAgent.CreateInNewDomain();

    var correlators = new IInteractionSource<String>[args.Length];

    for (int i=0; i<args.Length; i++)
     correlators[i] = opAgent::Arg <-- new Argument {
      arg = args[i],
      upper = ((System.DateTime.Now.Millisecond % 2) == 0) };

    for (int i=0; i<correlators.Length; i++)
     Console.WriteLine("Got {0} back for {1}",
      receive(correlators[i]), args[i]);

    PrimaryChannel::ExitCode <-- 0;
   }
 }
```

In fact, Axum uses an experimental C# compiler that provides some interesting and different features, a superset of C# 3.0 (3.0 + asynchronous methods), none of which is implied for C# 4.0, by the way. What this does permit, however, is the ability to mix-and-match C# and Axum code in the same project, something unique to Axum so far. See the Axum Programmer's Guide for details.Axum has additional features not described here, covered in some detail in the Axum Programmer's Guide, including a close relationship with WCF to make it easier to write high-scale services, but this introduction should be sufficient to get started building applications, libraries or services in Axum. Remember that this is a research project, and users can offer feedback via the Axum Forums on the DevLabs Web site.

What if Axum fails to turn into a shipping product, or readers want to use it before then? For starters, Axum's success or failure depends significantly in part on user reaction, and so will live or die by quality feedback—send the Axum team your thoughts, and agitate for its release in a public fashion! But even if Axum fails to graduate, remember, the concepts behind Axum are not unique. In the academic phraseology, Axum embodies the Actors model of concurrency, and several other Actors models are available for the .NET developer who wants something for production today. The open-source project "Retlang" embodies several of these concepts, as does the F# language (look at F#'s MailboxProcessor type) or the Microsoft Concurrency and Control Runtime (CCR), both of which are near-shipping status, if not there already.

In the end, remember, the goal is not to create projects that use every CLR-based language in existence, but to find languages that can solve particular problems and use them when the situation demands.

And remember, *Quot linguas calles, tot homines vales.* ■

TED NEWARD *is the principal with Neward and Associates, through which he speaks, writes and coaches on building reliable enterprise systems. He's written numerous books, taught and spoken at conferences all over the world, is an INETA speaker, and has received the MVP award in three different areas of expertise. Reach him at ted@tedneward.com or through his blog at blogs.tedneward.com.*

Shake and Skip to Interact with Your Music

Chris Mitchell

The G-Sensor, or accelerometer, is fast becoming a staple in the mobile development arsenal. Windows Mobile devices have joined the array of gadgets offering capabilities that let developers do everything from determining how often someone drops his phone to allowing users to control applications and games by tilting, shaking, hitting, or throwing their phones about—the last action perhaps needs a healthy disclaimer attached. This article looks at how to access and use accelerometer data with an HTC Diamond to control music playback, so the phone skips tracks in Windows Media Player Mobile when a user shakes it.

Because the use of accelerometers is a new area of development for Windows Mobile, and to keep with the general purpose of the article, I've written the application Shake 'n Skip (shown in **Figure 1**) to make it easy to determine how to start, get data from, and close the sensor, as well as change the sample application's function with a couple of lines of code. The application discussed in this article can easily be recoded to open an Internet browser, lock the device, or do anything else you want to try, all with minimal coding effort. In addition, the application has a logger that enables raw output from the sensor to be viewed so that other detection techniques beyond the simple shake-detection algorithm described here can be developed. Such algorithms might encompass the detection of other phone gestures to control your applications and games.

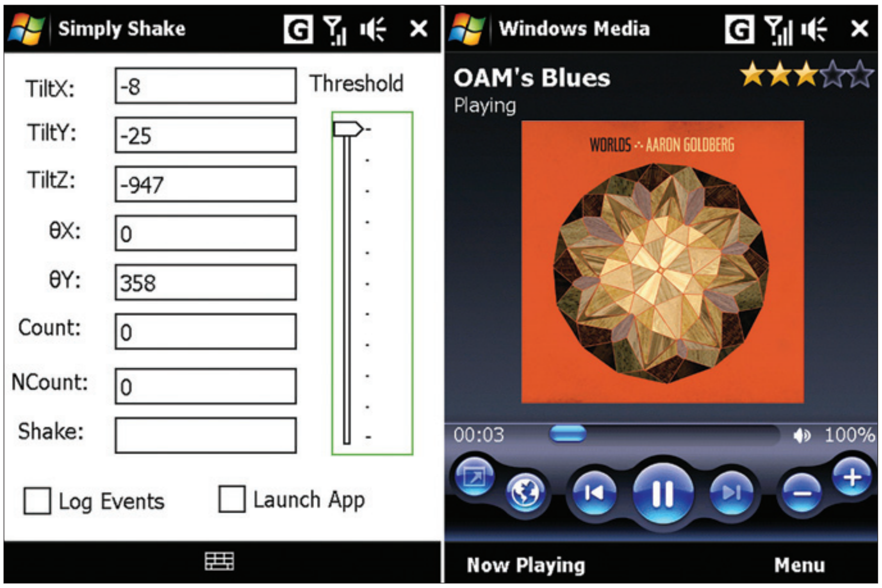


Figure 1 **Screenshots of the Shake 'n Skip application (left) and controlled Windows Media Player Mobile (right).**

You can read more about developing with the HTC Diamond accelerometer at blog.enterprisemobile.com/2008/07/using-htc-diamonds-sensor-sdk-from-managed-code. Note that the code in this article works with the HTC Diamond and other HTC devices, but will not work with other phones.

This article uses the following technologies:
Windows Mobile, Windows Media Player Mobile

This article discusses:
- Windows Mobile development
- Accelerometers for mobile devices

Code download available at:
code.msdn.microsoft.com/mag200909WiMoAccel

Application Discussion

The Shake 'n Skip application needs to perform several tasks. First, it needs to periodically get the most up-to-date information from the sensor. Second, it needs to encompass a metric that can detect whether the phone is being shaken. Third, if the application determines that the phone is being shaken, it needs to take the relevant action, which in this example is to cause a Windows Mobile media player to skip to the next track. In the sample application, I'll simulate key presses to launch and control Windows Media Player. This method enables you to control much of the phone's functionality, return to the home screen, lock the device, or launch any application.

Many of the applications that use accelerometers are giving a new lease on life to older applications and games.

Many of the applications that use accelerometers are giving a new lease on life to older applications and games. For example, Labyrinth, a game in which you move a ball around on a maze-like board, trying to get through the maze without rolling the ball into a hole, has been around for a long time. A Labyrinth toy was patented as long ago as 1891 by S. D. Nix. Does anyone think he worried about having enough spare MIPS to run his application, or that he investigated using vibration control to simulate a ball hitting the side of the maze at some speed? Many of the applications use the accelerometer in different ways to create an engaging new interface for the phone and can also combine the accelerometer with other outputs to create some very compelling user experiences.

Accelerometers

A quick definition to get us started: an accelerometer is a sensor that measures acceleration, the rate of change in velocity across time. Velocity is described using a vector, which, unlike a scalar, has a magnitude and direction—something I'm sure you remember from Physics 101.

There are many different types of accelerometers, from a water tube with an air bubble that shows the direction of acceleration to the integrated circuit types we work with in mobile phones. The HTC Diamond uses a MEMS (Micro-Electro Mechanical System)-based accelerometer, a type that forms small structures with dimensions in the micrometer scale (one millionth of a meter). Most other phones on the market do as well. In the phone, a three-axis accelerometer provides orientation, indicating whether the phone is facing up or down, whether the phone is standing up with the screen toward you or is upside down with the screen away, and all combinations in-between. The orientation of the phone at any point can be described with the following three values:

1. TiltX or Roll; 0 is flat. Measures rotation around the center of the phone running lengthwise from the mouth piece to the earphone.
2. TiltY or Pitch; 0 is flat. Measures rotation around the center of the phone running widthwise across the screen.
3. TiltZ (not yaw); 0 is straight up, the minus sign (-) is flat, and the plus sign (+) is upside down. These measures can be seen more clearly in **Figure 2**, which shows raw output together with the axis. Keep in mind that these three values together form a vector that shows the direction of gravity.

Now we need to detect when the phone is being shaken and, in that case, to issue a command to skip a track in Windows Media Player.

System Components

The Shake 'n Skip application is not hosting the Windows Mobile media player, nor is the application hosted in some type of plug-in, despite a plug-in being the obvious choice for redistribution. The application is written as a simple mobile Win form, so you can reuse the application for many different purposes and test your own ideas for how you might use the accelerometer. The application allows you to output the sensor's data to a log file that you can then test offline to see how well your algorithm ideas work. **Figure 3** shows the logical flow of the application.

The application's sensor component is made up of a single function that takes a reference to a structure and returns the values in that

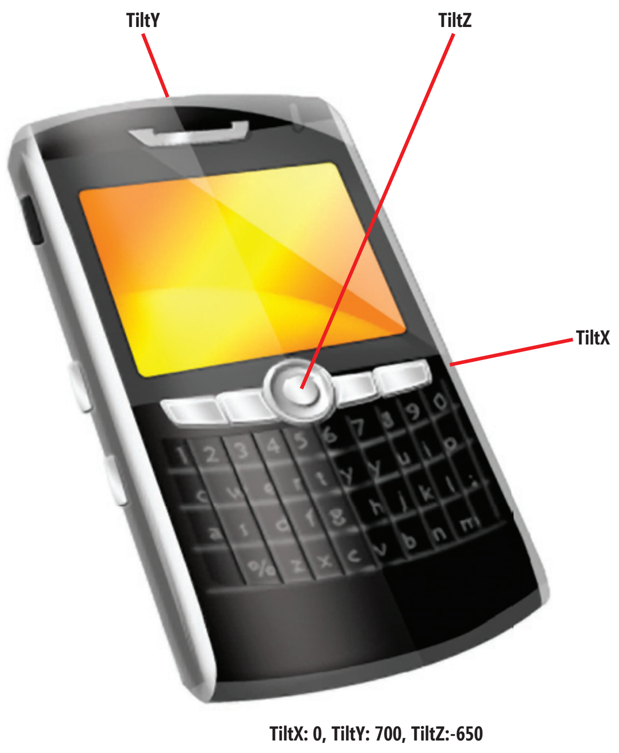


Figure 2 **A Graph Showing Output from the Accelerometer for a Phone Position**

structure. This sensor component is called on a periodic timer. In the application, the timer is set to call this function every 100 milliseconds. Be careful if you change this value because you could end up with aliasing effects if the value is set too low. The shake algorithm detection is also made up of a single function that takes the most up-to-date data returned from the sensor component. The shake algorithm compares the Euclidean distance (described later) between the last sensor data and the most recent sensor data. It then determines, using a threshold, whether a shake has been detected over a set number of comparisons. The number of successful comparisons needed to detect a shake is determined by two inputs: detect window and reset window. After a shake has been detected, the command component conditionally calls two functions, depending on the mode of operation selected on the interface: KeyCommand for Application or Logger. KeyCommand for Application takes an application named Wmplayer.exe and a phone key to simulate being pressed. The logger logs the data from the accelerometer to Gsensorlog.txt, a file placed in the root of your phone's storage. For simplicity, I added the class for the application to the same namespace created by Visual Studio. The namespace holds the default class inherited from the form class that holds the UI.

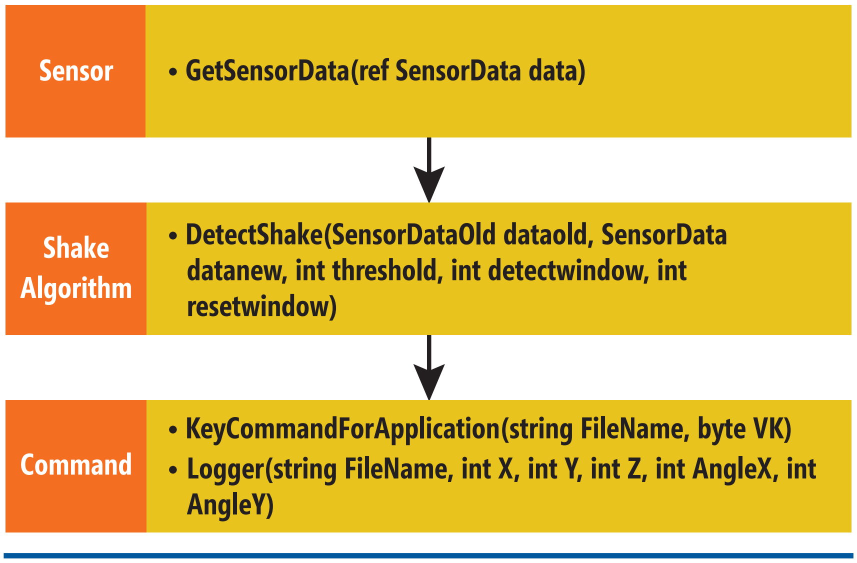

Sensor
• GetSensorData(ref SensorData data)
Shake Algorithm
• DetectShake(SensorDataOld dataold, SensorData datanew, int threshold, int detectwindow, int resetwindow)
Command
• KeyCommandForApplication(string FileName, byte VK)
• Logger(string FileName, int X, int Y, int Z, int AngleX, int AngleY)

Figure 3 **System Flow for Monitoring and Alerting of Location-Aware Tasks**

System Components—G-Sensor

Accessing the accelerometer's data requires calling from a managed C# environment to the unmanaged DLL provided on HTC phones that queries the sensor and provides the data. This sensor call requires three parts to be set up.The first part is a struct that is marshaled and stores the information needed from the main routine that calls the unmanaged DLL. The variables AngleY and AngleX are not used in the application but return the number of degrees to their respective planes.

```
//Data structure passed to sensor query api
public struct SensorData
{
 public short TiltX;
 public short TiltY;
 public short TiltZ;
 public short Unknown1;
 public int AngleY;
 public int AngleX;
 public int Unknown2;
};
```

Several Pinvoke calls are needed as well, as you can see in the following code. It is worth noting that if you want to access an external DLL function such as CreateEvent but the function is listed as using User32 as the DLL, you can replace User32 with Coredll on Windows Mobile in a large number of situations and the call will still work.

```
[DllImport("HTCSensorSDK")]
extern static IntPtr HTCSensorGetDataOutput(IntPtr handle,
 out SensorData sensorData);

[DllImport("HTCSensorSDK")]
extern static IntPtr HTCSensorOpen(int sensor);

[DllImport("HTCSensorSDK")]
extern static void HTCSensorClose(IntPtr handle);

[DllImport("coredll", SetLastError = true)]
extern static IntPtr CreateEvent(IntPtr eventAttributes, bool
manualReset, bool intialState, string name);

[DllImport("coredll", SetLastError = true)]
extern static bool EventModify(IntPtr handle, uint func);

[DllImport("coredll")]
extern static bool CloseHandle(IntPtr handle);
```

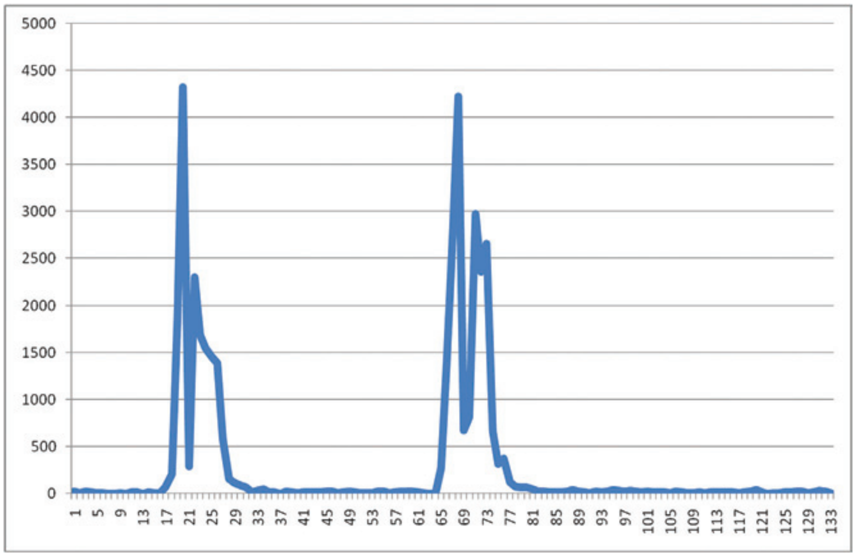


Figure 4. **A Graph Showing Euclidean Distance Change over Two Shakes**

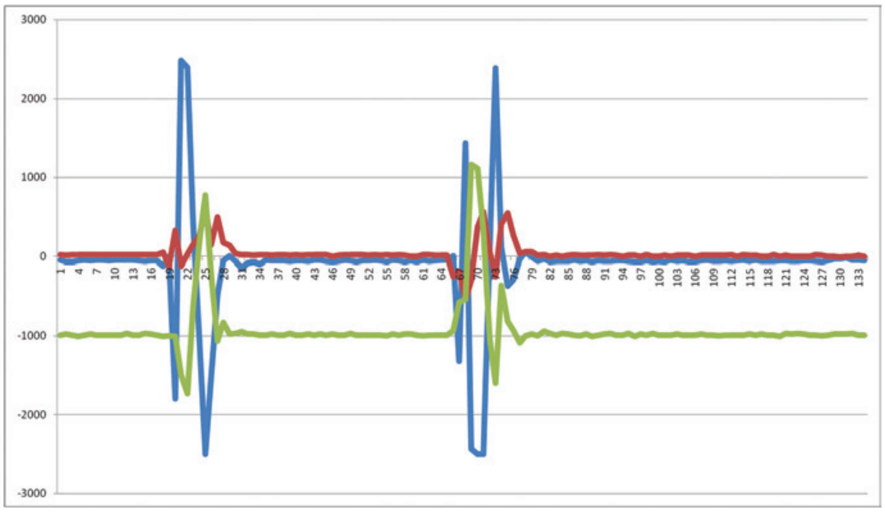


Figure 5. **A Graph Showing Changes to TiltX (Blue), TiltY (Red) and TiltZ (Green) Over Two Shakes**

The final part of getting the data is to call the functions shown in the following code and pass in the SensorData struct to handle the data. The CreateEvent call synchronization object tells the OS that HTC_GSENSOR_SERVICESTART must occur before the thread can resume running. When that event occurs, the thread can again be scheduled for CPU time. After it is scheduled, the thread continues running. The application thread is now synchronized with the sensor event.

```
public void GetSensorData(ref SensorData data)
{
 //Initialise and start sensor
 IntPtr Handle = HTCSensorOpen(1);
 IntPtr hEvent = CreateEvent
 (IntPtr.Zero, true, false, "HTC_GSENSOR_SERVICESTART");
 EventModify(hEvent, 3);
 CloseHandle(hEvent);
 HTCSensorGetDataOutput(Handle, out data);

 return;
}
```

Shaking a phone creates a rapid change in the direction of gravity relative to the phone's axis.

System Components—Shake Algorithm

Now that the sensor information can be obtained, we need to find an algorithm that allows us to detect a shake using this data.

The thinking behind the development of the algorithm goes like this: shaking a phone creates rapid change in the direction of gravity relative to the phone's axis. This means we should see rapid change in the TiltX, TiltY and TiltZ values, and if we set a threshold value to determine when rapid-enough change is occurring, we should be able to filter out slow-moving change—for example, answering or picking up the phone rather than shaking.

However, we still need a measure to detect the rapid change. The most obvious way is to look at distance measured between two sets of data. If the distance is greater than a certain value, we can say the change is happening rapidly.

Given *$P=(p_x, p_y, p_z)$ and $Q=(q_x, q_y, q_z)$*, the distance is computed as $\sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}$.

Euclidean distance is a simple distance metric I've used to measure rapid change. The calculation takes the past three axis accelerometer outputs (q_x, q_y, q_z) and compares the result with the current output (p_x,p_y,p_z). However, dropping the phone sharply or hitting the phone could also generate a rapid change, so we need to also insert a time feature into the algorithm to ensure that only a protracted shake will trigger the shake-detection component.

Figure 6 **Main Shake-Detection Function**

```
public bool DetectShake(SensorDataOld dataold, SensorData datanew,
  int threshold, int detectwindow, int resetwindow)
{
  if (FirstTimeEntryFlag != 0)
  {
    //Convert values to use inbuilt Math library
    double Xold = Convert.ToDouble(dataold.TiltX);
    double Yold = Convert.ToDouble(dataold.TiltY);
    double Zold = Convert.ToDouble(dataold.TiltZ);
    double X = Convert.ToDouble(datanew.TiltX);
    double Y = Convert.ToDouble(datanew.TiltY);
    double Z = Convert.ToDouble(datanew.TiltZ);

    //Set new values to old
    dataold.TiltX = datanew.TiltX;
    dataold.TiltY = datanew.TiltY;
    dataold.TiltZ = datanew.TiltZ;

    //Calculate Euclidean distance between old and data points
    double EuclideanDistance = Math.Sqrt(Math.Pow(X - Xold, 2)
      + Math.Pow(Y - Xold, 2)
      + Math.Pow(Y - Yold, 2));

    //Set shake to true if distance between data points is
    //greater than the defined threshold
    if (EuclideanDistance > threshold)
    {
      DetectWindowCount++;
      if (DetectWindowCount > detectwindow)
      {DetectWindowCount = 0; ResetWindowCount = 0; return true;}
    }
    else
    {
      ResetWindowCount++;
      if (ResetWindowCount > resetwindow)
        {DetectWindowCount = 0; ResetWindowCount = 0; return false;}
    }
  }

  //No longer the first run.
  FirstTimeEntryFlag = 1;
  return false;
}
```

Keys and Key Codes for Windows Mobile

There are many fairly sophisticated actions that can be performed on Windows Mobile devices with minimal programming by simulating key presses. Want to show the today screen? Want to answer a call or lock your phone? The same command can be used for all three. Windows Mobile maintains a device–independent keyboard model that enables it to support a variety of keyboards. At the lowest level, each key on the keyboard generates a scan code when the key is pressed and released. The scan code is a hardware–dependent number that identifies the key. The keyboard driver translates or maps each scan code to a virtual key code. The virtual key code is a hardware–independent hexadecimal number that identifies the key.

```
const int KEYEVENTF_KEYUP = 0x02;
const int KEYEVENTF_KEYDOWN = 0x00;
keybd_event(VK, 0, KEYEVENTF_KEYDOWN, 0);
keybd_event(VK, 0, KEYEVENTF_KEYUP, 0);
```

Answer a call:

```
byte VK = 0x72;  // Simulate answer phone call being pushed.
```

Lock device:

```
byte VK = 0x85;  // Simulate phone being locked.
```

Go to homepage:

```
byte VK = 0x5B;  // Simulate home key being pressed.
```

For more information, see msdn.microsoft.com/library/bb431750.aspx

As you can see from **Figure 4,** a large and (most important) sustained distance value differentiates the phone being shaken from its normal resting position. If you look at **Figure 5,** you see the corresponding sensor values. You should be able to see that the phone was shaken with the screen facing upward. The change in TiltX is caused by my wrist twisting to shake the phone, whereas the TiltY value changes very little because the phone was held mostly flat while resting and when being shaken. The code in **Figure 6** shows the calculation of the Euclidean distance and the strategy for determining when the phone has been shaken.

One other feature of the algorithm worth mentioning is the reset count. I found that it was useful not only to require multiple consecutive values but to also allow two high values followed by one small followed by another large to trigger the shake detection—three consecutive high values separated by one low value. To deal with scenarios similar to these, I used two counters instead of one. The first counts up to the required number of high Euclidean values detected above the threshold. The second keeps a reset count zeroed at the point of one large change detection, and can reset the entire count if a given number of large changes are not detected consecutively after the last large change. And now that we have a metric that can be used to determine whether a phone is being shaken, we need to know what to do with this information and what it can control.

Watch the Shake 'n Skip Video

To see how the Shake 'n Skip application works, you can watch a video that is available with the online version of this article at msdn.microsoft.com/magazine/ee413721.

System Components—Skip Track

As I mentioned earlier, one of the reasons to construct the application as a Win Form application is to be sure the application can be reused as much as possible. To continue this strand of simplicity, I used a specific technique to control Windows Mobile media player that involved simulating keypad controls that can be used to achieve a number of activities. (See the sidebar, Keys and Keycodes for Windows Mobile.) This technique is joined with the use of the System Diagnostic namespace to make sure that the application is started and at the forefront of the view on the Windows Mobile device. Here's the code that starts an application and simulates a key press on the device.

The use of accelerometers in mobile application development is likely to increase.

```
public void KeyCommandForApplication(string FileName, byte VK)
{
 System.Diagnostics.Process p = new System.Diagnostics.Process();
 p.StartInfo.FileName = FileName;
 p.Start();
 const int KEYEVENTF_KEYUP = 0x02;
 const int KEYEVENTF_KEYDOWN = 0x00;
 keybd_event(VK, 0, KEYEVENTF_KEYDOWN, 0);
 keybd_event(VK, 0, KEYEVENTF_KEYUP, 0);
}
```

Wrapping Up

I am sure that the algorithm used here is a little simplistic and can probably be improved, but under limited testing the application functions well. The use of accelerometers in mobile application development is likely to increase. When this is combined with mobile GPS (see msdn.microsoft.com/en-us/magazine/2009.01.wm6gps.aspx), light sensors and other inputs, new and exciting interactions with applications become possible.

Many thanks to Scott Seligman and Koushik Dutta for their great pages and code on working with HTC's accelerometer. Also many thanks to Craig of bexmedia.net for making the video look good for me. ■

CHRIS MITCHELL *finished his Ph.D. in machine learning and music/sound signal process and was then a Kauffman/NCGE fellow. He is currently involved in a setting up a startup company in Cambridge, U.K. You can contact him at chris.mitchell@anglia.ac.uk.*

Code Coverage for Concurrency

Chris Dern and Roy Patrick Tan

We are at another crossroads in the industry, as more and more transistor-laden processors demand multi-threaded code to realize their full potential. While machines from desktops to netbooks sport no less than dual-core processors under the hood, throngs of hungry transistors sit idle—crying for multi-threaded applications to devour. To address the oncoming wave of concurrent applications, companies like Intel and Microsoft are racing to market with profilers, frameworks, debuggers and libraries. As multi-threaded applications proliferate, so too will discussions of deadlocks, live locks and data races become increasingly common across the software industry. Software development professionals need to adopt new tools, techniques and metrics that can deal with multi-threaded software.

Code Coverage and Concurrency

Traditional code coverage metrics, such as statement, block, and branch, fail to address test adequacy concerns introduced by concurrency. Let's take statement coverage as an example. Although an imperfect metric, statement coverage is popular, and it is helpful to a degree in finding out how thoroughly you have tested your code. As a refresher, statement coverage measures how many statements of your application's code were executed. It is a simple metric that is used by many unit testing tools, and many software development teams include high statement coverage as a code-quality bar. Unfortunately, statement coverage does not give you any visibility as to how much of the concurrency in your code is being tested.

As an example, **Figure 1** shows a simple implementation of a thread-safe queue type written in C# (throughout this article, we use C# and .NET for our examples, but the idea of synchronization coverage is applicable to native code as well).

Our example has only two methods: Enqueue and Dequeue, and wraps a .NET Queue<T> by surrounding every enqueue and dequeue operation in a lock. How might we test this queue implementation? The following code shows a simple unit test for our thread-safe queue:

```
void SimpleQueueTest() {
    ThreadSafeQueue<int> q = new ThreadSafeQueue<int>();
    q.Enqueue(10);
    Assert.AreEqual(10, q.Dequeue());
}
```

Note that this simple test gives us 100 percent statement coverage, since this one test exercises every statement in our queue implementation.

But, wait, we have a problem—the queue is supposed to be thread-safe, but thus far we've tested it using only a single thread! We aren't testing the critical behavior which was the reason why we implemented this ThreadSafeQueue type in the first place. Statement coverage tells us that we have 100 percent coverage when we use one thread on a type meant to be accessed simultaneously. Clearly, statement coverage is inadequate to tell us how much of the *concurrency* aspects of the code we have tested. What are we missing?

Disclaimer: This article is partly based on a prototype synchronization coverage tool for .NET, called Sync Cover. All details are subject to change.

This article uses the following technologies:

Synchronization Coverage

This article discusses:

- Code Coverage
- Synchronization Coverage
- Round Tripping with IL
- Capturing Contention

Code download available at :

code.msdn.microsoft.com/magazine/mag200909SyncCover

Figure 1 **A Thread-Safe Queue implementation in C#**

```
public class ThreadSafeQueue<T> {

    object m_lock = new object();
    Queue<T> m_queue = new Queue<T>();

    public void Enqueue(T value) {
        lock (m_lock) {
            m_queue.Enqueue(value);
        }
    }

    public T Dequeue() {
        lock (m_lock) {
            return m_queue.Dequeue();
        }
    }
}
```

What we are missing is a branch hidden inside the lock statement. Imagine that we replace the lock with a custom locking primitive, such as the simplified implementation of a spin-lock method, as follows:

```
void Acquire() {
   while (!TryAcquire ()) {
     // idle for a bit
     Spin();
   }
}
```

The Enter method calls TryAcquire, and if it fails to acquire the lock (such that TryAcquire returns false), it spins a little and tries again. If we combined this custom locking code with our Enqueue method, how would that affect statement coverage? The following code shows how we can rewrite Enqueue with custom locking:

```
public void Enqueue(T value) {
    while (!TryAcquire ()) {
        //idle for a bit
        Spin();
    }

    m_queue.Enqueue(value);
    Release();
}
```

Now, if we run our test, we suddenly see missing statement coverage. Any single threaded test will miss the spin statement, because TryAcquire will only return false if there was another thread that already holds the lock. The only way the method will spin is if some other thread has entered the critical section. That is, we can only cover the spin statement if there was contention. This implicit branch inside the lock statement is the source of our coverage hole.

Synchronization Coverage

Because we don't expect anybody to replace lock statements with their own mutual exclusion primitives (nor do we advocate doing so), we need to find a way to expose this hidden branch, so that we can measure the amount of contention that occurred during our tests. Researchers at IBM came up with a code coverage model called Synchronization Coverage that can do just this.

You can read the paper referenced in the More Information section below, but the core idea is simple. First, we take a synchronization primitive—for example the .NET Monitor type (which is used by the lock keyword in C# and the SyncLock keyword in Visual Basic). Then, at every point in your code where a lock is being acquired, record whether it either blocked (in the case of, say, Monitor.Enter), or otherwise failed to acquire the lock (for example with Monitor.TryEnter). Either outcome means that contention occurred. Thus, to say that we have coverage over a lock, it's not sufficient to have executed a lock statement; we must also execute the lock statement while some other thread is holding the lock.

While this technique applies to both native and managed applications, the rest of this article will discuss our managed solution—a prototype synchronization coverage tool for .NET called Sync Cover.

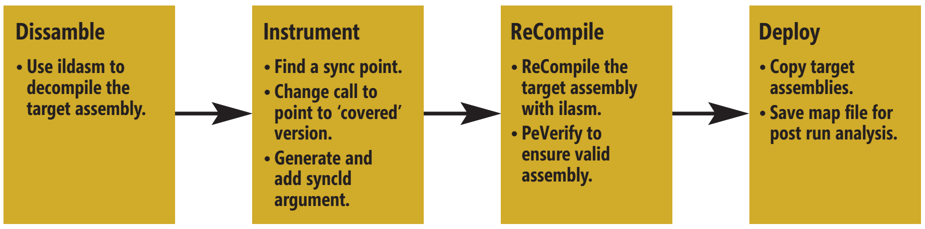
Let's take a moment to clarify the concepts used by our coverage tool. A sync point is a specific lexical call site, which invokes a synchronization method. Taking a quick look back at our sample thread-safe queue (**Figure 1**), with our C# compiler hat on, we see two such places hidden in the lock statements in Enqueue and Dequeue. Upon any single execution we are interested in two types of coverage: Statement Coverage, where the method did not experience any contention; and Contention Coverage, where the method was forced to block or wait for another thread to release it.

For the rest of this article we will focus specifically on System.Threading.Monitor, the work horse of the .NET synchronization stable. However, it's useful to note this approach works equally well with other primitives, like System.Threading.Interlocked.CompareExchange.

Round Tripping with IL

Our goal for this synchronization coverage tool is to transparently instrument existing .NET assemblies such that we can intercept every Monitor.Enter call in that assembly and determine whether that sync point encountered contention. Like a majority of the code coverage solutions in use today, our tool applies techniques that rewrite IL (the .NET Intermediate Language) to produce instrumented versions of the target assemblies. Although we do spend a majority of our time in C#, by dealing directly with the IL, in theory this tool should work for all .NET languages that compile to IL.

While there are a few possible technologies to enable code rewriting, including the newly released Common Compiler Infrastructure from Microsoft Research (MSR), for simplicity we chose to fall back to the trusty IL compiler and decompiler combo ILASM and ILDASM. Working directly with the IL in plain text files proved



Dissamble
• Use ildasm to decompile the target assembly.
Instrument
• Find a sync point.
• Change call to point to 'covered' version.
• Generate and add syncId argument.
ReCompile
• ReCompile the target assembly with ilasm.
• PeVerify to ensure valid assembly.
Deploy
• Copy target assemblies.
• Save map file for post run analysis.

Figure 2 **Tool Chain**

to be a huge boon during the discovery, design and development phases of the project, enabling us to explore design choices in a "pay-for-play" approach with our time investment. We proved the viability and practicality of our solution at first via a completely manual process, comprised of batch files and notepad. While this may not be the best choice for a production-quality tool, we recommend this approach for any similar prototype rewriting application.

With this overall design decision complete, let's take a quick tour of our tool chain, as shown in **Figure 2**.

Using ILDASM, we decompile the target assembly into a text file with the IL instructions. We then load the source code into memory behind an object-based façade. We then apply the required modifications and code injections to the code, emitting the modified source back to disk. Then it is just a simple matter of recompiling the covered assembly with ILASM, and checking it with PeVerify to catch any silly invalid transformations we may have applied. Once we have our coverage assemblies, we perform our test pass as normal, and collect the runtime coverage file for later analysis.

Our tool set is comprised of two main parts: Sync Cover, which automates this instrumentation process and provides an IL round trip platform upon which we build our instrumentation; and Cover Viewer, a Windows Presentation Foundation (WPF) application that provides a familiar and intuitive way to explore the coverage files.

Capturing Contention

By now, you should be wondering how we can observe and record when a sync point experiences contention. Looking back at the "hidden-branch" mental model of how Monitor.Enter could be implemented provides us a clue, if we apply it almost literally in our transformation. We see in the preceding simplified implementation of a spin-lock code that the Acquire method captures this exact information in the return value. All we need to do is see it.

As much as we like to rewrite other people's IL, inserting arbitrary branches promises to open a Pandora's box of problems we would rather avoid. Recalling the second problem of recording and linking the sync point data suggests we turn to our programmers tool box and reach for some indirection.

We solve both problems by introducing a wrapper class—a synchronization coverage-aware façade over Monitor whose methods take a syncId argument. This syncId is a generated unique integer that uniquely identifies a sync point—that is, we give every sync point a unique ID, and when we hit a sync point, we pass that ID to our wrapper class. Our coverage implementation begins by calling Monitor.TryEnter, recording the result and then, if needed, delegating to the original blocking Monitor.Enter. The sync point states are managed by a simple in-memory database class we call the CoverageTracker. Putting all the pieces together, our coverage version of Monitor.Enter looks like this, as shown in **Figure 3**.

Figure 3 **Coverage Version of Monitor.Enter**

```
namespace System.Threading.SyncCover {

  public static class CoverMonitor {

    public static void Enter(object obj, int syncId) {
      if (Monitor.TryEnter(obj)) {
        coverageTracker.MarkPoint(syncId, false); // No Contention.
      } else {
        Monitor.Enter(obj);
        coverageTracker.MarkPoint(syncId, true); // Contention
      }
    }
  }
}
```

The upcoming .NET 4 Monitor type includes changes that make it resilient against asynchronous exceptions.

Note that the code in **Figure 3** is intended to support the .NET 3.5 version of Monitor.Enter. The upcoming .NET 4 Monitor type includes changes that make it resilient against asynchronous exceptions—see blogs.msdn.com/ericlippert/archive/2009/03/06/locks-and-exceptions-do-not-mix.aspx. Adding support for the .NET 4 overloads is just a matter of overloading the wrappers in a similar fashion.

Once we have our method in hand, the instrumentation process becomes fairly straight forward. Let's look at the IL produced by the .NET 3.5 C# compiler for the lock statement, as follows:

```
IL_0001:  ldfld object class ThreadSafeQueue'1<!T>::m_lock
IL_0002:  call  void [mscorlib]System.Threading.Monitor::Enter(object)
```

We just need to look for calls to Monitor.Enter and replace them with calls to our CoverMonitor.Enter, while not forgetting to inject an additional syncId argument before the method invocation. The following code illustrates this transformation:

```
IL_0001: ldfld object class ThreadSafeQueue'1<!T>::m_lock
  I_ADD01: ldc.i4 1  // sync id
  IL_0002: call void System.Threading.Coverage.Monitor::Enter(object,
int32)
```

As a final part to this process, we should talk a bit about reporting, and what type of information is useful. So far, we know about the sync points that are identified by a syncId, injected directly into the target source code. It would be much more useful if we could get the source files and line numbers of each sync point. We found that ILDASM with the /LINENUM option provides us the information we need by extracting source file locations and line numbers from the program database (PDB).

When we encounter a new sync point during the instrumentation process, we generate the next syncId, and capture this contextual information in a map. This mapping, seen in the following code, is then emitted as a file at the end of the instrumentation:

```
T|ThreadSafeQueue.exe
SP|
0|D:\ThreadSafeQueue\ThreadSafeQueue.cs|9|ThreadSafeQueue`1<T>|Enqueue
1|D:\ThreadSafeQueue\ThreadSafeQueue.cs|15|ThreadSafeQueue`1<T>|Dequeue
```

Figure 4 **Example Coverage Run**

```
~|B3|~
A|ThreadSafeQueue.exe
C|ThreadSafeQueue.exe
D|20090610'091754
T|demo
M|DWEEZIL
O|Microsoft Windows NT 6.1.7100.0
P|4
SP|
0|1|1
1|1|0
~|E|~
```

Show Me the Data!

It's show time. Once we have our covered binaries, it's a simple matter of executing the program as many times as you like. This produces a runtime file containing the coverage metrics collected during the execution, as shown in **Figure 4**. Taking inspiration from other coverage tools in the market, we provide a simple WPF-based viewer to visualize the coverage results.

But let's step back for a moment. We know that our single-threaded test case will give us zero percent synchronization coverage. How can we fix the test so that we can cause contention? **Figure 5** shows a slightly improved test that should cause contention in the Enqueue method. After running the test, we can see the results, as shown in **Figure 6**.

Here we see the three possible coverage cases for a given sync point. In this example, we see that Enqueue experienced at least one count of contention and so appears in green. Dequeue, on the other hand was executed but did not contend, as shown in yellow. We've also added a new property Count, which was never called, and shows as red.

Figure 5 **A Test That Causes Contention on Enqueue**

```
void ThreadedQueueTest()
{
    ThreadSafeQueue<int> q = new ThreadSafeQueue<int>();
    Thread t1 = new Thread(
        () =>
        {
            for (int i = 0; i < 10000; i++)
            {
                q.Enqueue(i);
            }
        });

     Thread t2 = new Thread(
        () =>
        {
            for (int i = 0; i < 10000; i++)
            {
                q.Enqueue(i);
            }
        });

    t1.Start();
    t2.Start();
    t1.Join();
    t2.Join();

    Assert.AreEqual(0, q.Dequeue());
}
```

Using Synchronization Coverage

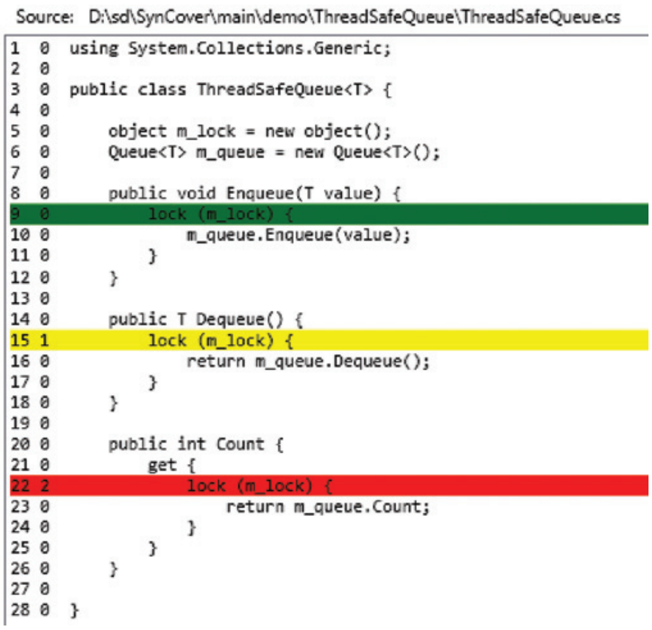
So when should we use synchronization coverage? Like any code coverage metric, synchronization coverage attempts to measure how well you have tested your code. Any sort of locking in your application means that your code was meant to be accessed simultaneously, and you seriously should be curious whether your test suite actually exercises those locks. Your team should aim for 100 percent synchronization coverage, especially if your application expects to run in parallel a lot.

Your team should aim for 100 percent synchronization coverage, especially if your application expects to run in parallel a lot.

Trying to practice what we preach, we have used this synchronization coverage tool in the course of testing the Parallel Extensions to the .NET Framework. It has helped us find testing holes and bugs during this development cycle, and we expect to continue using the metric going forward. Two scenarios where synchronization coverage has helped us are particularly interesting:

Non-concurrent multi-threaded tests—Synchronization coverage found that some tests that we thought were running concurrently were actually not. **Figure 7** is a simple multi-threaded

Figure 6 **Sync Cover Viewer**



Source: D:\sd\SynCover\main\demo\ThreadSafeQueue\ThreadSafeQueue.cs

```
1  0  using System.Collections.Generic;
2  0
3  0  public class ThreadSafeQueue<T> {
4  0
5  0      object m_lock = new object();
6  0      Queue<T> m_queue = new Queue<T>();
7  0
8  0      public void Enqueue(T value) {
9  0          lock (m_lock) {
10 0              m_queue.Enqueue(value);
11 0          }
12 0      }
13 0
14 0      public T Dequeue() {
15 1          lock (m_lock) {
16 0              return m_queue.Dequeue();
17 0          }
18 0      }
19 0
20 0      public int Count {
21 0          get {
22 2              lock (m_lock) {
23 0                  return m_queue.Count;
24 0              }
25 0          }
26 0      }
27 0
28 0  }
```

Figure 7 **A Concurrent Test That Might Not Be Concurrent After All**

```
void ThreadedQueueTest() {

    ThreadSafeQueue<int> q = new ThreadSafeQueue<int>();
    Thread t1 = new Thread(
        () => {
            for (int i = 0; i < 10; i++) {
                q.Enqueue(10);
            }
        });

    Thread t2 = new Thread(
        () => {
            for (int i = 0; i < 10; i++) {
                q.Enqueue(12);
            }
        });

    t1.Start();
    t2.Start();
    t1.Join();
    t2.Join();
}
```

test similar to **Figure 5**, but this time each thread enqueues only 10 items to the queue.

Just because we started two threads, however, does not mean that they are actually running in parallel. What we found was that these kinds of tests with very short execution times per thread more often than not will have one thread execute completely after the other thread. We want a test that consistently experiences contention. One solution is to have each thread enqueue more items to make it more likely to cause contention. A better solution is to use a tool like CHESS, which forces every interleaving between the two threads to occur.

Unnecessary locks—It is quite possible that some synchronization points are not being covered because they cannot be covered. In the following code example, if the lock on b is always acquired when the lock on a is held, then it is impossible to cover the lock on b:

```
lock(a) {
    lock(b) {
        //... stuff ...
    }
}
```

Surprisingly, synchronization coverage has actually helped us find unnecessary locks like these. It turns out that sometimes a resource being protected by a lock may no longer need to be. For example, a thread that competes over the resource may not be needed anymore and removed from the code, but the lock on the remaining thread was not removed. While the extra lock is harmless in the behavioral sense, it could still hurt performance. (Note, though, that just because a lock is never contended for doesn't mean it's not necessary; such information just provides a starting point of an investigation to determine whether it's actually needed.)

Limitations and Future Work

Like any code coverage metric, synchronization coverage is not perfect—it has its limitations. A chief limitation that synchronization coverage shares with other coverage metrics is that it cannot measure what is not there. Synchronization coverage cannot tell you that you need to lock over some resource, only that the resources you have locked have been contended over. Thus, even 100 percent synchronization coverage does not mean that your testing effort is done, only that you have achieved some level of thoroughness in your testing.

Another issue is that instrumenting your code for synchronization coverage could alter the scheduling of the threads in your tests such that you might get coverage in your instrumented test run, but not in your uninstrumented execution (although in our experience, we've found that this is usually not a problem). Again, a tool like CHESS can help. Our prototype tool allows us to measure contention over Monitor and Interlocked operations. In the future, we plan on adding features that will allow us to measure other synchronization primitives such as Semaphore and ManualResetEvent. We believe that a code coverage metric, like synchronization coverage, could be as useful and widespread for concurrent applications as statement coverage is for single-threaded applications.

Measure a Must

You can't improve what you can't measure. Thus, as we develop more and more multi-threaded software applications, we must measure how thoroughly we have tested the concurrency aspects of our software. Synchronization coverage is a simple, practical way to do this. We hope that as you navigate this whole new world of multi-core computing, you can take the ideas in this article to improve your quality process. ■

More Information

- Parallel Computing at Microsoft:
  msdn.microsoft.com/concurrency
- The synchronization coverage paper:
  Bron, A., Farchi, E., Magid, Y., Nir, Y., and Ur, S. 2005. Applications of synchronization coverage. In Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Chicago, Ill., USA, June 15-17, 2005). PPoPP '05. ACM, New York, N.Y., 206-212.
- The CHESS tool:
  research.microsoft.com/en-us/projects/chess/default.aspx
  Musuvathi, M. and Qadeer, S. 2007. Iterative context bounding for systematic testing of multi-threaded programs. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego, Calif., USA, June 10-13, 2007). PLDI '07. ACM, New York, N.Y., 446-455.
- Common Compiler Infrastructure (CCI):
  ccimetadata.codeplex.com

ROY TAN *is a software development engineer in test with the Parallel Computing Platform group at Microsoft. He received his Ph.D. in Computer Science at Virginia Tech in 2007.*

CHRIS DERN *is a concurrency software development engineer in test for the Parallel Computing Platform team at Microsoft—where the only thing better than writing concurrency software, is testing it.*

Debugging Task-Based Parallel Applications in Visual Studio 2010

Daniel Moth and Stephen Toub

Ask any CPU or GPU hardware manufacturer, and they will tell you that the future is manycore. Processor core speeds are not increasing at the exponential rates of the past four decades; instead, new machines are being built with more cores. As a result, the "free" performance improvements that application developers relied on year after year are gone. To regain the free lunch offered by better and better hardware—and to enhance your applications with new performance-sensitive features—you need to take advantage of multiple cores via parallelism.

In Visual C++ 10 and the Microsoft .NET Framework 4, both available with Visual Studio 2010, Microsoft is introducing new libraries and runtimes to significantly ease the process of expressing parallelism in your code base, together with new tool support for performance analysis and debugging of parallel applications. In this article, you will learn about debugging support in Visual Studio 2010, much of which is focused on task-based programming models.

This article is based on a prerelease version. All information herein is subject to change.

Technologies discussed:

Visual C++, Visual Studio 2010, Microsoft .NET Framework 4, PLINQ

This article discusses:

- Parallelism and tasked-based programming
- Visual Studio 2010 debugging tools

Code download available at:

code.msdn.microsoft.com/mag200909ParallelDbg

The Need for Task-Based Programming

The reason to inject parallelism into your application is to take advantage of multiple cores. A single, sequential piece of work will run on only one core at a time. For an application to use multiple cores, multiple pieces of work are needed to enable multiple threads to process that work in parallel. Thus, given a single piece of work, achieving parallel acceleration through multicore execution requires partitioning that single piece of work into multiple units that can run concurrently.

The simplest schemes involve static partitioning: split the work into a fixed number of fixed-size units. Of course, you don't want to have to write your code for each configuration of hardware it will be executed on, and predetermining a fixed number of units ahead of time inhibits the scalability of your application as it runs on bigger and better machines. You can instead choose the number of units dynamically at run time based on details of the machine. For example, you can partition work into one unit per core. This way, if all the units are equal in size in terms of the processing time they require, and if you use one thread per unit, you should be able to saturate the machine.

This approach, however, still leaves a lot to be desired. It's rare that a real-world workload can be split in such a way that each unit is guaranteed to take the same amount of time, especially when you take into account external factors such as other workloads that might be running on the machine concurrently and consuming some of the machine's resources. In such cases, one-unit-per-core partitioning will likely end up distributing the work unevenly: some threads will complete their units before others, resulting in load imbalance, and some cores will sit idle while others finish. To address this, you want to over-partition the work, dividing the workload into the smallest feasible units so that all of the machine's resources can partake in the processing of the workload until it is complete.

If executing a unit of work incurred zero overhead, the solution just proposed would be ideal, but very few real-world operations incur zero overhead. Historically, threads have been the mechanism for executing such a unit of work: you create a thread for each unit of work, let it execute, and then tear down the thread. Unfortunately, threads are relatively heavy weight, and the overhead incurred from using them in this manner can prohibit the type of over-partitioning we've described. What you need is a lighter-weight mechanism for executing these partitioned units to minimize overhead—a mechanism that will let you over-partition with less guilt. With this approach, rather than creating one thread per unit, you could utilize a scheduler that schedules individual units to be executed on threads that it manages, keeping the number of units as small as possible while still ensuring maximum throughput.

What we've just described is a thread pool, which amortizes the cost of thread management across all the work items scheduled to it, thus minimizing the overhead associated with an individual work item. In Windows, such a thread pool is accessible through the QueueUserWorkItem function exported from Kernel32.dll. (Windows Vista introduced new thread pooling functionality as well.) In .NET Framework 4, such a pool is accessible through the System.Threading.ThreadPool class.

While the previously mentioned APIs enable decomposition with relatively minimal overhead, they're largely targeted at "fire and forget" work. For example, the .NET Framework 4 ThreadPool class doesn't provide any consistent mechanism for exception handling, for cancellation of work, for waiting on work to complete, for receiving notifications when work completes, and so forth. These gaps are filled by new APIs in .NET Framework 4 and Visual C++ 10 designed for "task-based" programming in both managed and native code. Tasks represent units of work that can be executed efficiently by an underlying scheduler while still exposing rich functionality for working with and controlling aspects of their execution. In Visual C++ 10, these APIs are centered on the Concurrency::task_group and Concurrency::task_handle types. In .NET Framework 4, they are centered on the new System.Threading.Tasks.Task class.

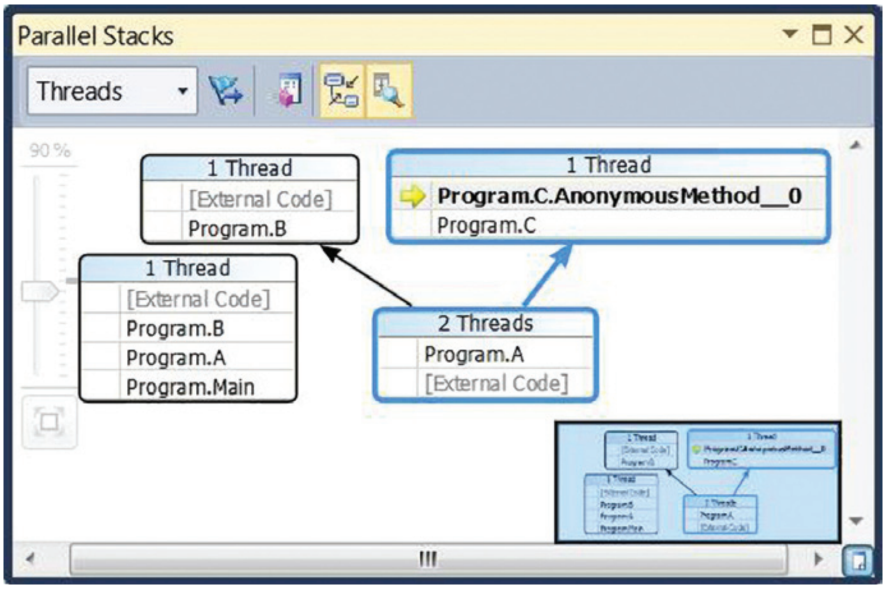
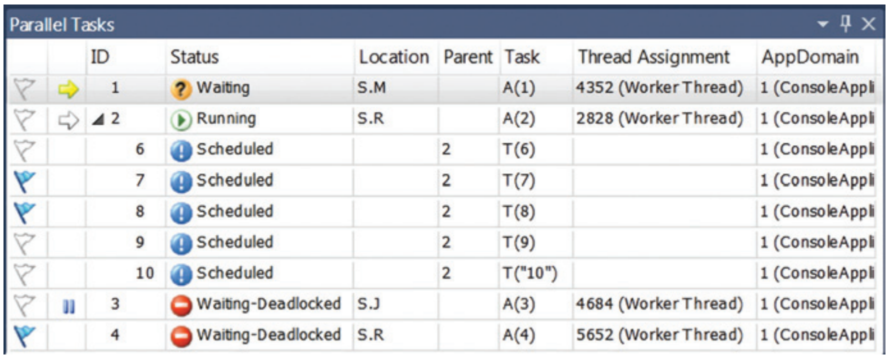


Figure 1 **Parallel Stacks Window**



Parallel Tasks

		ID	Status	Location	Parent	Task	Thread Assignment	AppDomain
	⇨	1	Waiting	S.M		A(1)	4352 (Worker Thread)	1 (ConsoleAppli
	⇨	◢ 2	Running	S.R		A(2)	2828 (Worker Thread)	1 (ConsoleAppli
		6	Scheduled		2	T(6)		1 (ConsoleAppli
		7	Scheduled		2	T(7)		1 (ConsoleAppli
		8	Scheduled		2	T(8)		1 (ConsoleAppli
		9	Scheduled		2	T(9)		1 (ConsoleAppli
		10	Scheduled		2	T("10")		1 (ConsoleAppli
	‖	3	Waiting-Deadlocked	S.J		A(3)	4684 (Worker Thread)	1 (ConsoleAppli
		4	Waiting-Deadlocked	S.R		A(4)	5652 (Worker Thread)	1 (ConsoleAppli

Figure 2 **Parallel Tasks Window**

Debugging in Visual Studio Today

The history of software development has demonstrated time and time again that programming models benefit greatly from exemplary debugging support, and Visual Studio 2010 delivers in this regard by providing two new debugging tool windows to assist with task-based parallel programming. But before we look at these new features, let's review the debugging experience in Visual Studio today to set the stage.

(For the rest of this article, we'll use the .NET task-based types for explanatory purposes, but the debugging support described applies equally to native code as well.)

The entry point to debugging a process in Visual Studio is, of course, attaching the debugger. This occurs by default when you press F5 (the equivalent of choosing the Debug > Start Debugging command) in a project that is open in Visual Studio. You can also manually attach the debugger to a process by choosing the Debug > Attach to Process menu command. Once the debugger is attached, the next step is to break into the debugger. This can occur in multiple ways, including hitting a user-defined break-

Figure 3 **Finding Primes**

```
static void Main(string[] args)
{
    var primes =
        from n in Enumerable.Range(1,10000000)
                  .AsParallel()
                  .AsOrdered()
                  .WithMergeOptions(ParallelMergeOptions.NotBuffered)
        where IsPrime(n)
        select n;
    foreach (var prime in primes) Console.Write(prime + ", ");
}

public static bool IsPrime(int numberToTest) // WARNING: Buggy!
{
    // 2 is a weird prime: it's even. Test for it explicitly.
    if (numberToTest == 2) return true;

    // Anything that's less than 2 or that's even is not prime
    if (numberToTest < 2 || (numberToTest & 1) == 0) return false;

    // Test all odd numbers less than the sqrt of the target number.
    // If the target is divisible by any of them, it's not prime.
    // We don't test evens, because if the target is divisible
    // by an even, the target is also even, which we already checked for.
    int upperBound = (int)Math.Sqrt(numberToTest);
    for (int i = 3; i < upperBound; i += 2)
    {
        if ((numberToTest % i) == 0) return false;
    }

    // It's prime!
    return true;
}
```

point, manually breaking (via the Debug > Break All command), the process requesting it (for example, in managed code, via a call to the System.Diagnostics.Debugger.Break method), or even when an exception is thrown.

After your process breaks into the debugger, all the threads in your application are halted: no code executes at that point until you continue execution (excluding threads that the debugger itself uses). This halt in execution allows you to inspect the state of your application at that moment. When inspecting application state, you often have a mental picture of what the state should be, and you can use the various debugger windows to spot a difference between expectation and reality.

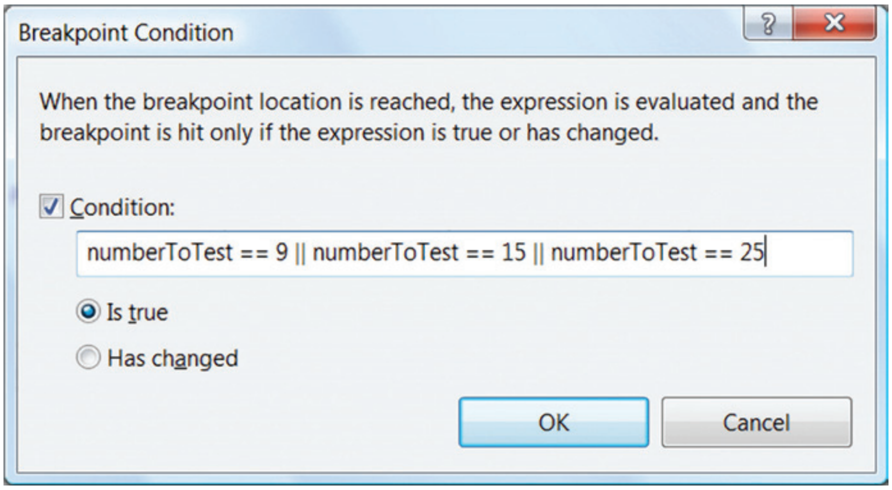
The main debugging windows that developers use in Visual Studio are the Threads window, the Call Stack window, and the variable windows (Locals, Autos, Watch). The Threads window displays a list of all the threads in your process, including information such as the thread ID and thread priority and an indication (a yellow arrow) of the current thread, which by default is the thread that was executing when the debugger broke into the process. Probably the most important information about a thread is where it was executing when the debugger halted its execution, shown by the callstack frame in the Location column. Hovering your cursor over that column reveals the equally important call stack—the series or method calls that the thread was in the process of executing before reaching the current location.

The Call Stack window, which displays the call stack of the current thread, provides much richer information about the call stack, including interaction opportunities.

To display the call stack of another thread in the Call Stack window, you have to make the other thread current by double-clicking it in the Threads window. The method it is currently executing in (which is at the top of the call stack) is indicated by a yellow arrow and is known as the "topmost frame," the "leaf frame," or the "active stack frame." This is the method from which the thread will continue execution when you leave the debugger and continue running the application. By default, the active stack frame is also the current stack frame—in other words, the method that drives the variable inspection, which we'll describe next.

The variable windows are used to inspect the values of variables in your application. The variables of local methods are usually browsed in the Locals and Autos windows; global state (variables not declared in a method) can be examined by adding them to the Watch window. Starting with Visual Studio 2005, more and more developers examine state by hovering their mouse pointers over a variable of interest and reviewing the resulting pop-up DataTip (which can be thought of as a shortcut to the Quick Watch windows). It is important to note that values for variables can be displayed only if the variables are in the scope of the current stack frame (which, as we established earlier, is by default the active stack frame of the current thread).

To examine variables that were in scope earlier in the call stack of the thread, you need to change the current stack frame by double-clicking the stack frame you want to examine in the Call Stack window. At this point, the new current stack frame is indicated by a green curved tail arrow (while the active stack frame retains the yellow arrow). If you also want to examine variables on another thread, you need to change the current thread in the Threads win-


Breakpoint Condition

When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.

☑ Condition:

numberToTest == 9 || numberToTest == 15 || numberToTest == 25

[✓ Is true] Has changed

Figure 4 **Setting Conditional Breakpoints**

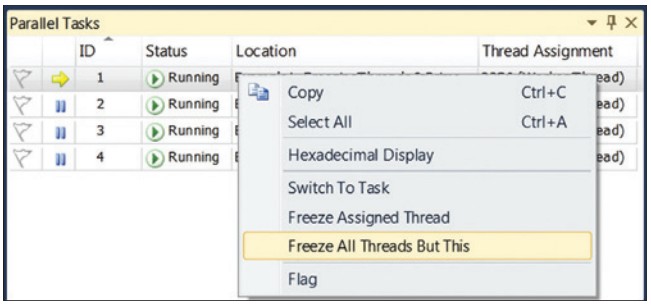


Figure 5 **Choosing the Freeze All Threads But This Command**

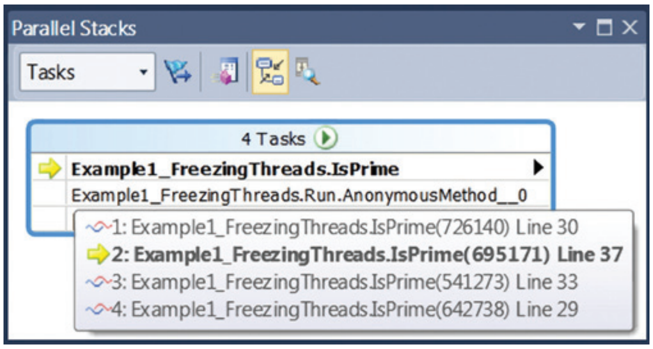

Parallel Stacks
Tasks
4 Tasks
Example1_FreezingThreads.IsPrime
Example1_FreezingThreads.Run.AnonymousMethod__0
1: Example1_FreezingThreads.IsPrime(726140) Line 30
2: Example1_FreezingThreads.IsPrime(695171) Line 37
3: Example1_FreezingThreads.IsPrime(541273) Line 33
4: Example1_FreezingThreads.IsPrime(642738) Line 29

Figure 6 **Coalescing of Stack Frames**

dow and then switch the current frame on its call stack in the Call Stack window.

In summary, when you break into your process in the debugger, you can very easily inspect the variables in scope at the executing method of one of the threads. However, to create a complete picture of where all your threads are executing, you need to individually examine the calls stack of each thread by double-clicking each thread to make it current, looking at the Call Stack window, and then creating the holistic picture mentally. Furthermore, to examine variables on various stack frames of various threads, two levels of indirection are needed again: switch threads and then switch frames.

Parallel Stacks

When applications use more threads (which will become commonplace as people use machines with more processing resources), you need to be able to see in a single view where those threads are executing at any given moment. That is what the Parallel Stacks tool window in Visual Studio 2010 delivers.

To preserve screen real estate, but also to indicate methods of particular interest to parallelism scenarios, the window coalesces into the same node the call stack segments that are common among threads at their root. For example, in **Figure 1**, you can see the call stacks of three threads in a single view. The figure shows one thread that went from Main to A to B and two other threads that started from the same external code and then went to A. One of these threads continued to B and then to some external code, and the other thread continued to C and then to some AnonymousMethod. AnonymousMethod is also the active stack frame, and it belongs to the current thread. Many other features are supported

Figure 7 **Task-Based Code with Dependencies**

```
static void Main(string[] args) // WARNING: Buggy!
{
    var task1a = Task.Factory.StartNew(Step1a);
    var task1b = Task.Factory.StartNew(Step1b);
    var task1c = Task.Factory.StartNew(Step1c);

    Task.WaitAll(task1a, task1b, task1c);

    var task2a = Task.Factory.StartNew(Step2a);
    var task2b = Task.Factory.StartNew(Step2b);
    var task2c = Task.Factory.StartNew(Step2c);

    Task.WaitAll(task1a, task1b, task1c);

    var task3a = Task.Factory.StartNew(Step3a);
    var task3b = Task.Factory.StartNew(Step3b);
    var task3c = Task.Factory.StartNew(Step3c);

    Task.WaitAll(task3a, task3b, task3c);
}
```

in this window, such as zoom, a bird's-eye view, filtering of threads via flagging, and most of the same functionality already available in the Call Stack window.

If your application creates tasks rather than threads, you can switch to a task-centric view. In this view, call stacks of threads not executing tasks are omitted. Additionally, call stacks for threads are trimmed to represent the real call stacks of tasks—that is, a single-thread call stack could include two or three tasks that you want to split out and view separately. A special feature of the Parallel Stacks window allows you to pivot the diagram on a single method and clearly observe the callers and callees of that method context.

Parallel Tasks

In addition to looking at the real call stacks of tasks in the Parallel Stacks window, another new debugger window exposes additional information about tasks, including the task ID, the thread assigned to the task, the current Location, and the entry point (the delegate) passed to the task at creation. This window, called the Parallel Tasks window, exposes features similar to the Threads window, such as indicating the current task (the top-most task running on the current thread), the ability to switch the current task, flagging of tasks, and freezing and thawing threads.

Figure 9 **Deadlocking Code**

```
static void Main(string[] args)
{
    int transfersCompleted = 0;
    Watchdog.BreakIfRepeats(() => transfersCompleted, 500);

    BankAccount a = new BankAccount { Balance = 1000 };
    BankAccount b = new BankAccount { Balance = 1000 };
    while (true)
    {
        Parallel.Invoke(
            () => Transfer(a, b, 100),
            () => Transfer(b, a, 100));
        transfersCompleted += 2;
    }
}

class BankAccount { public int Balance; }

static void Transfer(BankAccount one, BankAccount two, int amount)
{
    lock (one) // WARNING: Buggy!
    {
        lock (two)
        {
            one.Balance -= amount;
            two.Balance += amount;
        }
    }
}
```

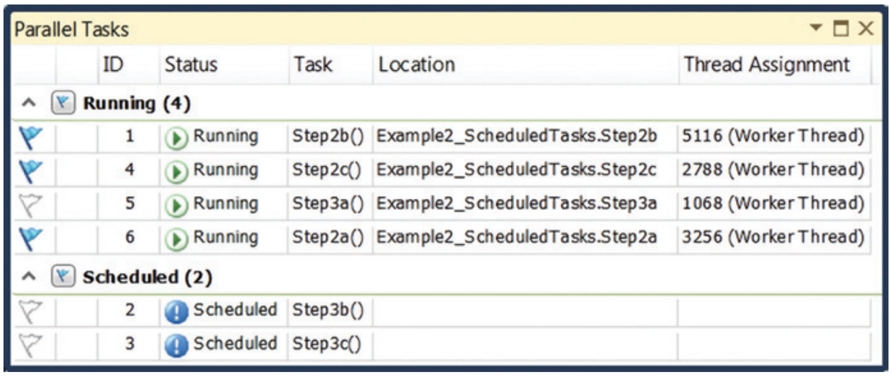
Perhaps the biggest value to developers is the Status column. The information provided in the Status column allows you to distinguish between running tasks and tasks that are waiting (on another task or on a synchronization primitive) or tasks that are deadlocked (a specialization of waiting tasks for which the tool detects a circular wait chain). The Parallel Tasks window also displays scheduled tasks, which are tasks that have not run yet but are sitting in some queue waiting to be executed by a thread. An example can be seen in **Figure 2**. For more information on both the Parallel Stacks and Parallel Tasks windows, see the blog posts at danielmoth.com/Blog/labels/ParallelComputing.html and the MSDN documentation at msdn.microsoft.com/dd554943(VS.100).aspx.

Find the Bug

One of the best ways to understand new tooling functionality is to see it in action. To do that, we've created a few buggy code snippets, and we'll use the new tool windows to find the underlying errors in the code.

Single-Stepping First, consider the code shown in Figure 3. The goal of this code is to output the prime numbers between 1 and 10,000,000 and to do so in parallel. (The parallelization support is provided by Parallel LINQ; see blogs.msdn.com/pfxteam and msdn.microsoft.com/dd460688(VS.100).aspx for more information.) The implementation of IsPrime is buggy, as you can see by running the code and viewing the first few numbers output:

```
2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 23, 25, …
```



Parallel Tasks

	ID	Status	Task	Location	Thread Assignment
Running (4)					
⚑	1	Running	Step2b()	Example2_ScheduledTasks.Step2b	5116 (Worker Thread)
⚑	4	Running	Step2c()	Example2_ScheduledTasks.Step2c	2788 (Worker Thread)
⚐	5	Running	Step3a()	Example2_ScheduledTasks.Step3a	1068 (Worker Thread)
⚑	6	Running	Step2a()	Example2_ScheduledTasks.Step2a	3256 (Worker Thread)
Scheduled (2)					
⚐	2	Scheduled	Step3b()		
⚐	3	Scheduled	Step3c()		

Figure 8 **Using Parallel Tasks to Find Dependency Problems**

Most of these numbers are primes, but 9, 15, and 25 are not. If this were a single-threaded application, you could easily step through the code to find the reason for the inaccurate results. However, when you perform single-stepping (choose Debug > Step Into, for example) in a multithreaded program, any thread in the program is eligible for stepping. This means that as you step you might be jumping between threads, making it more difficult to understand the control flow and the diagnostic information about the current location in the program. To assist with this, you can take advantage of several capabilities of the debugger. The first is to set conditional breakpoints.

As shown in **Figure 4**, you can set a breakpoint (in this case on the first line of the IsPrime method) and indicate to the debugger to break in only when a certain condition is met—in this case, when one of the inaccurate "primes" is being evaluated.

We could have set the debugger to break in when one of these values was hit (rather than when any of them was hit), but we can't make assumptions about the order in which PLINQ evaluates the values under the covers. Instead, we told the debugger to look for any of these values so as to minimize wait time before it breaks.

After the debugger breaks in, we want to tell it to single-step just the current thread. To do that, we can take advantage of the debugger's ability to freeze and thaw threads and specify that frozen threads won't run until we thaw them. The new Parallel Tasks window makes it easy to find the thread that should be allowed to continue (look for the yellow arrow icon) and to freeze all other threads (via the ContextMenu), as shown in **Figure 5**.

With the irrelevant thread(s) frozen, we can now single-step through our buggy IsPrime. By debugging numberToTest==25, we can easily see what's gone wrong: the loop should include the upperBound value in its test, whereas this value is currently being excluded because the loop uses the less-than operator rather than less-than-or-equals. Here, the square root of 25 is 5, and 25 is evenly divisible by 5, but 5 won't be tested, so 25 is erroneously categorized as a prime.

The Parallel Stacks window also provides a nice, consolidated view of what's happening in our program when we break. **Figure 6** shows the current state of the application after we run it again, and this time explicitly break in using the debugger's Break All capability.

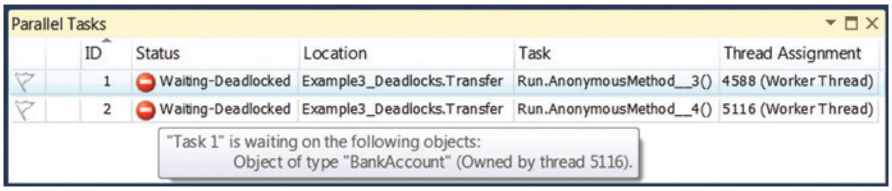
PLINQ is executing IsPrime in multiple tasks, and the value of numberToTest for all those tasks is visible in the pop-up, here showing that Task 1 is processing numberToTest==8431901, while Task 2 is processing numberToTest==8431607.

Dependency Problems

The code in **Figure 7** shows an instance of a pattern common in parallel applications. This code forks off multiple operations (step1a, step1b, step1c, which are all methods of the form "void StepXx()") that might run in parallel and then joins on them. Subsequently, the application forks again with code that requires the previous operations to already be complete because of a dependency on the operations' side effects (such as writing data to some shared arrays).

Unfortunately, this code includes a bug, and the developer who wrote it is seeing some inaccurate results being computed by the third set of tasks. The implication is that even though the developer is waiting for all of the prior tasks to complete, something is amiss, and not all of the previous computations have actually completed their results. To debug the code, the developer sets a breakpoint on the last WaitAll call and uses the Parallel Tasks window to see the current state of the program, which is shown in **Figure 8**.

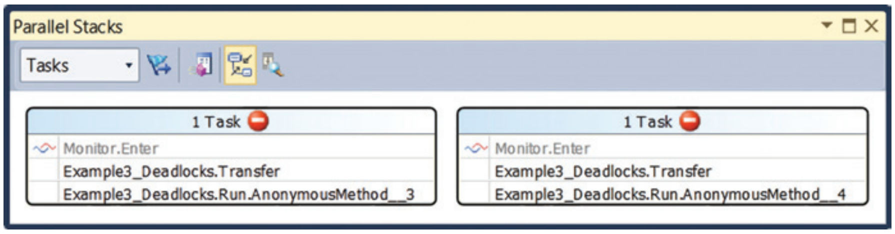
Sure enough, the Parallel Tasks window shows that the Task for Step2c is still running even though the tasks for Step 3 have been scheduled. A review of the second Task.WaitAll call demonstrates



	ID	Status	Location	Task	Thread Assignment
	1	Waiting-Deadlocked	Example3_Deadlocks.Transfer	Run.AnonymousMethod__3()	4588 (Worker Thread)
	2	Waiting-Deadlocked	Example3_Deadlocks.Transfer	Run.AnonymousMethod__4()	5116 (Worker Thread)

"Task 1" is waiting on the following objects:
Object of type "BankAccount" (Owned by thread 5116).

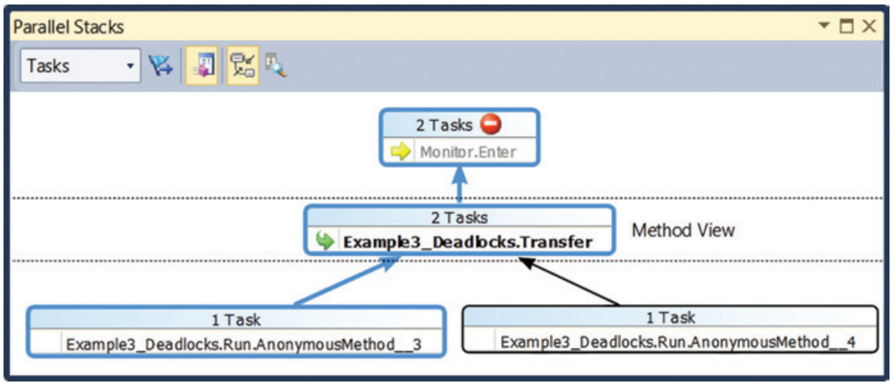
Figure 10 **Information on Deadlocks in Parallel Tasks**



1 Task
Monitor.Enter
Example3_Deadlocks.Transfer
Example3_Deadlocks.Run.AnonymousMethod__3

1 Task
Monitor.Enter
Example3_Deadlocks.Transfer
Example3_Deadlocks.Run.AnonymousMethod__4

Figure 11 **Parallel Stacks Showing Deadlocks**



2 Tasks
Monitor.Enter

2 Tasks
Example3_Deadlocks.Transfer

1 Task
Example3_Deadlocks.Run.AnonymousMethod__3

1 Task
Example3_Deadlocks.Run.AnonymousMethod__4

Figure 12 **Method View in Parallel Stacks**

why: because of typing errors, task1a, task1b, and task1c are being waited on instead of their task2 counterparts.

Deadlocks

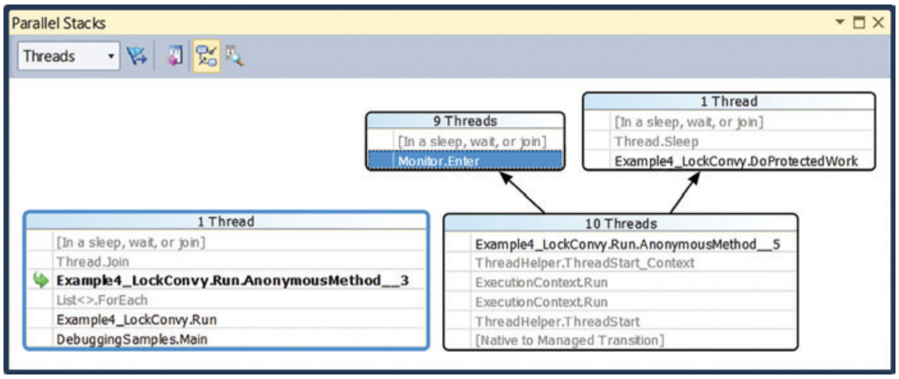
Figure 9 provides the prototypical example of a deadlock scenario, which results from not paying attention to lock ordering. The main code is continually transferring money between bank accounts. The Transfer method is meant to be thread-safe so that it can be called concurrently from multiple threads. As such, it internally locks on the BankAccount objects handed to it simply by locking on the first and then locking on the second. Unfortunately, this behavior can lead to deadlocks, as running this code will demonstrate. Eventually, the debugger breaks in when it finds that no transfers are proceeding. (Breaking in is performed using code that issues a Debugger.Break if it notices that no new transfers have been completed after a certain amount of time. This code is included in the download that accompanies this article.)

When you're working in the debugger, you immediately see a graphical representation demonstrating that there is a deadlock, as shown in **Figure 10**. The figure also demonstrates that hovering the pointer over the Waiting-Deadlocked status provides further details about exactly what's being waited on and which thread is holding the protected resource. Looking at the Thread Assignment column, you can see that Task 2 is waiting on a resource held by Task 1, and if you were to hover over Task 1, you would see the inverse.

This information can also be deduced from the Parallel Stacks tool window. **Figure 11** shows the Task View in Parallel Stacks, which highlights that there are two tasks, each of which is blocked in a call to Monitor.Enter (due to the lock statements from **Figure 9**). And **Figure 12** demonstrates the Method View available in the Parallel Stacks window (via the corresponding toolbar button). By focusing our view on the Transfer method, we can easily see that there are two tasks currently in Transfer, both of which have gone on to a call to Monitor.Enter. Hovering the pointer over that box provides further information on the deadlocked status of both tasks.

Figure 13 **Creating a Lock Convoy**

```
static void Main(string[] args) // WARNING: Buggy!
{
    object obj = new object();
    Enumerable.Range(1, 10).Select(i =>
    {
        var t = new Thread(() =>
        {
            while (true)
            {
                DoWork();
                lock (obj) DoProtectedWork();
            }
        }) { Name = "Demo " + i };
        t.Start();
        return t;
    }).ToList().ForEach(t => t.Join());
}
```



9 Threads
[In a sleep, wait, or join]
Monitor.Enter

1 Thread
[In a sleep, wait, or join]
Thread.Sleep
Example4_LockConvy.DoProtectedWork

1 Thread
[In a sleep, wait, or join]
Thread.Join
Example4_LockConvy.Run.AnonymousMethod__3
List<>.ForEach
Example4_LockConvy.Run
DebuggingSamples.Main

10 Threads
Example4_LockConvy.Run.AnonymousMethod__5
ThreadHelper.ThreadStart_Context
ExecutionContext.Run
ExecutionContext.Run
ThreadHelper.ThreadStart
[Native to Managed Transition]

Figure 14 **Lock Convoys with Parallel Stacks**

Lock Convoys

Lock convoys can occur when multiple threads repeatedly compete for the same protected region. (Wikipedia provides a good summary of lock convoys at en.wikipedia.org/wiki/Lock_convoy.) The code in **Figure 13** provides a quintessential example of a lock convoy: multiple threads repeatedly doing some amount of work outside a protected region but then taking a lock to do some additional work inside that protected region. Depending on the ratio between the work inside and outside the region, performance problems might result. These sorts of problems are visible after program execution by using a tool like the concurrency profiler that's available in Visual Studio 2010, but they can also be caught during execution by using a debugging tool like the Parallel Stacks window.

Figure 14 shows an execution of the code in **Figure 13.** The code was broken into a few seconds into its execution. You can see at the top of the image that nine threads are currently blocked waiting on a monitor—all the threads waiting for one thread to exit DoProtectedWork so that one of them can continue into the protected region.

Wrapping Up

In this article, you've seen examples of how Visual Studio 2010 debugger tool windows can simplify the act of finding bugs in task-based code. The task-based APIs for managed and native code are richer than what we could show in the short examples in this article, and we encourage you to explore them further in .NET Framework 4 and Visual C++ 10. On the tools front, in addition to the two new debugger windows discussed, a new parallel performance analyzer is integrated into the existing profiler in Visual Studio.

To get your hands on all the bits above, download the beta of Visual Studio 2010 from msdn.microsoft.com/dd582936.aspx. ■

DANIEL MOTH *works in the Parallel Computing Platform Team at Microsoft. He can be reached through his blog at danielmoth.com/Blog.*

STEPHEN TOUB *works in the Parallel Computing Platform Team at Microsoft. He is also a contributing editor to* MSDN Magazine.

Core Instrumentation Events in Windows 7

Dr. Insung Park and Alex Bendetov

Today's computer software constantly breaks new grounds. Consumer software applications offer a sophisticated set of features that enable rich new experiences. Powerful server applications are setting new records in throughput, speed and scale. These improvements have been made possible by rapid progress in hardware technologies and continuous adoption of software advancements in optimization, virtualization, and distributed and parallel computing. However, as a result, software applications have become larger and more complicated. At the same time, users' expectations about software quality are higher than ever. Fundamental characteristics such as performance, reliability and manageability have proved essential in the long-term success of software products, and they are often celebrated as primary features.

Increasing software complexity and higher user expectations on quality thus present a difficult challenge in software development. When an unexpected problem occurs, predicting internal states of all relevant components is nearly impossible. Retracing the history of execution flows is cumbersome and tricky, but often necessary in finding out the root cause of software problems. When users report problems after deployment, they expect the root cause of the problem to be quickly identified and addressed. The overwhelming number of hardware and software combinations, different workload characteristics, and usage patterns of end users make such tasks even tougher. The ability to use a mechanism that enables you to understand system execution in a transparent manner, with minimal overhead, is invaluable.

Disclaimer: This article is based on a prerelease version of Windows 7. Details are subject to change.

This article uses the following technologies:

Windows 7

This article discusses:

- Event Instrumentation
- Event Tracing for Windows (ETW)
- Core OS Events
- Memory Events

Event Instrumentation

Instrumentation is one such effective solution in measuring and improving software quality. Software performance counters have provided a convenient way to monitor application execution status and resource usage at an aggregate level. Event instrumentation has also been popular over the years. Events raised by a software component at different stages of execution can significantly reduce the time it takes to diagnose various problems. In addition to scanning for certain events or patterns of events, one can apply data mining and correlation techniques to further analyze the events to produce meaningful statistics and reports on program execution and problematic behavior. The ability to collect events on production systems in real time helps avoid the need to have an unwieldy debugger setup on customer machines.

Introduced in the Windows 2000 operating system, Event Tracing for Windows (ETW) is a general-purpose event-tracing platform on Windows operating systems. Using an efficient buffering and logging mechanism implemented in the kernel, ETW provides a mechanism to persist events raised by both user-mode applications and kernel-mode device drivers. Additionally, ETW gives users the ability to enable and disable logging dynamically, making it easy to perform detailed tracing in production environments without requiring reboots or application restarts.

The operating system itself has been heavily instrumented with ETW events. The ability to analyze and simulate core OS activities based on ETW events in development, as well as on production-mode systems, has been valuable to developers in solving many quality problems. With each subsequent Windows release, the number of ETW events raised by the operating system has increased; Windows 7 is the most instrumented operating system to date. In addition, Windows 7 contains tools that can utilize these operating system ETW events to analyze system performance and reliability, as well as uncover quality problems in software applications.

Many application problems surface as anomalies in OS resource usage, such as unexpected patterns or spikes in the consumption of CPU, memory, network bandwidth, IOs and so on. Because OS events for most system activities can be traced to the originating process and thread, one can make considerable progress in narrowing down possible root causes of many application problems, even without ETW instrumentation in applications. Of course, ETW instrumentation in the application would allow further diagnosis to be significantly more efficient.

In the first article of our two-part series, we present a high-level overview of the ETW technology and core OS instrumentation. Then, we discuss tool support to obtain and consume OS events. Next, we provide more details on the events from various subcomponents in the core OS. We also explain how the different system events can be combined to produce a comprehensive picture of system behavior, which we demonstrate by using a set of Windows PowerShell scripts.

Event Tracing for Windows

As mentioned earlier, ETW is a logging platform that efficiently records the events sent by software applications or kernel-mode components. Using ETW provider APIs, any application, DLL or driver can become an event provider (a component that raises events) for ETW. A provider first registers with ETW and sends events from various points in the code by inserting ETW logging API calls. Any recordable activity of importance can be an event, and it is represented by a piece of data written by ETW at the time of logging. These logging API calls are ignored when the provider is not enabled. An ETW controller application starts an ETW session and enables certain providers to it. When an enabled event provider makes a logging API call, the event is then directed to the session designated by the controller. Events sent to a session may be stored in a log file, consumed programmatically in real time, or kept in memory until the controller requests a flush of that data to a file. A previous article, "Improve Debugging And Performance Tuning with ETW" (msdn.microsoft.com/en-us/magazine/dvdarchive/cc163437.aspx), has more details about the ETW technology and how to add ETW instrumentation into an application. Over the years, ETW has come to support many different logging modes and features, which are documented on MSDN.

An ETW event consists of a fixed header followed by context-specific data. The header identifies the event and the component raising the event, while the context-specific data ("event payload" hereafter) refers to any additional data that the component raising the event wants to record. When an event raised by a provider is written to an ETW session, ETW adds additional metadata to the header, including thread and process IDs, the current CPU on which the logging thread is running, CPU time usage of the thread, and timestamp. **Figure 1** shows an XML representation of an event (a Process event of type Start) as decoded by the tracerpt tool (to be discussed later) in the XML dump file. The <System> section is common to all events and represents the common header that ETW records for each event. This contains timestamp, process and thread ID, provider GUID, CPU time usage, CPU ID, and so on. The <EventData> section displays the logged payload of this event. As shown in **Figure 1**, a Process Start event from the Windows kernel contains process key (a unique key assigned to each process for identification), process ID, parent process ID, session ID, exit status (valid only for a Process End event), user SID, executable file name of the process, and the command that started the process.

Many application problems surface as anomalies in OS resource usage.

ETW controllers are the applications that use the ETW control API set to start ETW sessions and enable one or more providers to those sessions. They need to give each session a unique name, and on Windows 7 there can be up to 64 sessions running concurrently.

Figure 1 **Process Start Event in XML Dump**

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
    <System>
        <Provider Guid="{9e814aad-3204-11d2-9a82-006008a86939}" />
        <EventID>0</EventID>
        <Version>3</Version>
        <Level>0</Level>
        <Task>0</Task>
        <Opcode>1</Opcode>
        <Keywords>0x0</Keywords>
        <TimeCreated SystemTime="2009-07-14T16:27:43.441456400Z" />
        <Correlation ActivityID="
          {00000000-0000-0000-0000-000000000000}" />
        <Execution ProcessID="2584" ThreadID="4324"
          ProcessorID="1" KernelTime="90" UserTime="15" />
        <Channel />
        <Computer />
    </System>
    <EventData>
        <Data Name="UniqueProcessKey">0xFFFFFA8005BBC950</Data>
        <Data Name="ProcessId">0x1430</Data>
        <Data Name="ParentId">0xA18</Data>
        <Data Name="SessionId">       1</Data>
        <Data Name="ExitStatus">259</Data>
        <Data Name="DirectoryTableBase">0x4E1D6000</Data>
        <Data Name="UserSID">guest</Data>
        <Data Name="ImageFileName">notepad.exe</Data>
        <Data Name="CommandLine">notepad</Data>
    </EventData>
    <RenderingInfo Culture="en-US">
        <Opcode>Start</Opcode>
        <Provider>MSNT_SystemTrace</Provider>
        <EventName xmlns=
          "http://schemas.microsoft.com/win/2004/08/events/trace">
          Process</EventName>
    </RenderingInfo>
    <ExtendedTracingInfo xmlns=
      "http://schemas.microsoft.com/win/2004/08/events/trace">
        <EventGuid>{3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c}</EventGuid>
    </ExtendedTracingInfo>
</Event>
```

The events from the kernel and core system components, however, are logged in a manner different from user-mode applications or kernel-mode device drivers. The core system logs events to a special session with a reserved name, "NT Kernel Logger." Only the "NT Kernel Logger" session ("kernel session" hereafter) receives core system events, and it does not accept events from any other regular event provider. Also, core OS events are enabled by specifying appropriate flags when a session is started. Each flag represents event instrumentation in a different core component that can be enabled selectively. This helps reduce the instrumentation overhead in the kernel and reinforces the authenticity of system events. In addition, a new feature in Windows 7 allows users to capture call stacks at the time of logging. If symbols are available, one can trace a chain of function calls that trigger the kernel events to be logged. The benefit from call-stack analysis will be discussed in subsequent sections, when we discuss individual events in more detail.

We have found it effective to construct a state machine and simulate OS activities when we look at core system events.

Collecting Events Using Tools on Windows

There are a few ETW control tools available on Windows that allow users to collect events. For instance, the Performance Monitor exposes ETW control in the form of a data collection set. The EventLog service is also capable of starting and stopping ETW sessions and viewing events. For command-line and script interfaces, logman.exe offers options to perform ETW control operations. For event consumption, the command-line tool tracerpt.exe can consume ETW log files and produce dumps in several formats, including CSV and XML. An example of an XML representation of a Process Start event is shown in **Figure 1**. In this article, we use logman.exe and tracerpt.exe in the samples we present. The "logman query providers" command in **Figure 2** lists the different flags that can be used by a controller when enabling the kernel session.

The following command starts the kernel session and enables process, thread, disk, network, image, and registry events. The collected events will be stored in a file called systemevents.etl in the current directory. Controlling the kernel session and collecting core OS events require administrator privileges:

```
> logman start "NT Kernel Logger" -p "Windows Kernel Trace" (process,thr
ead,img,disk,net,registry) -o systemevents.etl -ets
```

To stop the collection, users need to issue the "logman stop –ets" command:

```
> logman stop "NT Kernel Logger" -ets
```

The tracerpt tool can process the events in the log file into a readable format. By default, tracerpt accepts one or more log files and generates an event output file and a summary file. The default format of the output file is XML:

```
> tracerpt systemevents.etl
```

The logman and tracerpt help text, Windows Help and Support, and online documentation have more details on the switches and features of these tools.

There are advanced performance analysis tools that consume core OS events for various analysis and tuning scenarios. Windows Performance Toolkit (WPT) is one such notable tool, and is available from the Windows SDK (msdn.microsoft.com/en-us/performance/cc825801.aspx). WPT is useful to a broad audience, including system builders, hardware manufacturers, driver developers and general application developers. Its trace analysis tools, Xperf and XperfView, apply sophisticated techniques (including those introduced here) to aggregate and analyze the core OS events to offer meaningful perspectives into the OS and application behavior. Its flexible GUI provides many rich and customizable presentation options that can help users focus on different aspects of system activities. **Figure 3** shows a screenshot of the XperfView tool in action.

Core OS Events

The Windows core OS has many components instrumented with ETW. As shown in **Figure 2**, events are available for activities spanning various subsystems including processes, threads, disk and file IOs, memory, network, registry and so on. In this section, we provide details for each group of events. We also discuss a number of core operating system concepts. More information on these concepts can be found in OS-centric reference materials such as "Windows Internals, 5th Edition," by Russinovitch, Solomon and Ionescu (Microsoft Press, 2009). Core OS events are subject to change in future versions of Windows, as the platform and its instrumentation evolve to meet new requirements.

There are several ways to make use of event data. One can scan for a certain event, such as an error event, or a pattern of events that represent execution flow. Other popular methodologies include statistical analysis (counting and summarizing events), delta analysis (analyzing deltas between pairs of events, such as Start and End), activity analysis (tracking an activity/request through event correlation), and aggregation and pattern analysis based on call-stacks. Over the years, we have found it effective to construct a state machine and simulate OS activities when we look at core system events. When we explain the system events in this section, we will also describe how we use them in building the state machine.

Process, Thread, Image, Process Counter Events

A thread is a basic unit of execution on the Windows OS, and a process acts as a container for threads. Each process (and thread) is assigned an ID that is unique while it is running. IDs for processes and threads share the same number space. That is, while a thread with ID A is active, A is never given to other processes or threads. The only exception is the per-CPU idle threads and the idle process, whose IDs are all identical and have been historically 0. Process and Thread events are the basic building blocks in establishing the state machine, which is helpful in understanding more advanced system activities. There are two types of events for Process and Thread events: Start and End. Process Start and End are logged when a process starts and terminates, respectively. The same applies to Thread Start and End events. The payload for Process events has more details about the process, such as process name and parent process ID, as shown in **Figure 1**. Likewise, the Thread event payload contains thread-specific information, such as stack base and limit, and start address. It should be noted that the IDs of the starting processes or threads are part of the Process and Thread event payload, although the event header already has those items in it. Process Start events are logged in the context of a parent process that creates the current process. In that case, the process ID in the <System> section (event header) is the parent process. The process ID in the event payload is the one being created. The same applies to thread ID in Thread events.

For processes and threads that started before the event collection, ETW logs state rundown events. For the purpose of analysis, they are used to denote the running processes and threads at the time the event collection begins. ETW also logs rundown events for the remaining processes and threads still running when the collection ends. Process and Thread rundown events enumerate and log events in the same format as Process and Thread Start events for all processes and threads, including system and idle processes. Process and Thread rundowns events use different types, DCStart and DCEnd, to distinguish themselves from real process and thread creation and termination. A separate Defunct event is written for a process that has terminated, but with outstanding references to it.

Every process and thread active during event collection can be tracked using Process and Thread events. When you build a state machine, the processing routine should keep a list of active processes and threads, perhaps as some type of objects in the program. When a thread or process terminates, the corresponding object may be placed in a different ("complete") list. The process objects may also contain more details (such as process name) on the processes or threads that they refer to, picked up from the Process and Thread event payload. It is simpler and a lot more informative if one can refer to a process by name during analysis, rather than by ID, as IDs can be recycled once a process or thread terminates (and all references to it are released). When events from other core components indicate OS activities during the state machine construction and simulation, the threads and processes that initiated them are located in the state machine, and their objects are updated to attribute those activities, using primarily the process and thread IDs in the event headers. At the end, process and thread objects are aggregated and summarized into a report with various metrics.

Process and Thread rundown events enumerate and log events in the same format as Process and Thread Start events for all processes and threads, including system and idle processes.

Image events correspond to image (also known as module) files getting loaded and unloaded into process address space. There are four types of Image events: Load, Unload, DCStart and DCEnd. These events do not directly correlate to LoadLibrary calls, however. If a DLL is already loaded in the process, subsequent LoadLibrary calls for the same DLL simply increment the count of module references but will not map the module again. Like the DCStart and DCEnd types of Process and Thread events, Image DCStart and DCEnd are used to enumerate loaded modules of already running processes. Image events allow for the tracking of loaded modules and the mapping of addresses within a process. They are also im-

Figure 2 **NT Kernel Logger Enable Flags**

```
> logman query providers "Windows Kernel Trace"

Provider                                 GUID
-------------------------------------------------------------------------------
Windows Kernel Trace                     {9E814AAD-3204-11D2-9A82-
006008A86939}

Value               Keyword              Description
-------------------------------------------------------------------------------
0x0000000000000001  process              Process creations/deletions
0x0000000000000002  thread               Thread creations/deletions
0x0000000000000004  img                  Image load
0x0000000000000008  proccntr             Process counters
0x0000000000000010  cswitch              Context switches
0x0000000000000020  dpc                  Deferred procedure calls
0x0000000000000040  isr                  Interrupts
0x0000000000000080  syscall              System calls
0x0000000000000100  disk                 Disk IO
0x0000000000000200  file                 File details
0x0000000000000400  diskinit             Disk IO entry
0x0000000000000800  dispatcher           Dispatcher operations
0x0000000000001000  pf                   Page faults
0x0000000000002000  hf                   Hard page faults
0x0000000000004000  virtalloc            Virtual memory allocations
0x0000000000010000  net                  Network TCP/IP
0x0000000000020000  registry             Registry details
0x0000000000100000  alpc                 ALPC
0x0000000000200000  splitio              Split IO
0x0000000000800000  driver               Driver delays
0x0000000001000000  profile              Sample based profiling
0x0000000002000000  fileiocompletion     File IO completion
0x0000000004000000  fileio               File IO

The command completed successfully.
```

portant in mapping DPC, ISR, and System Call events, as will be discussed in a later section and in Part 2 of this series. The Image event payload contains information such as the module address base and size, and the name of the binary file loaded (or unloaded). Image events are also required for decoding call-stacks. In a typical state machine construction, Image events trigger the updating of a list of loaded modules into an aforementioned process object.

Process Counter events, when enabled, are logged at process termination and record in its payload a few properties regarding the process execution statistics over the lifetime of the process. They consist of peak memory size, peak working set size, peak paged and nonpaged pool usage, and peak page file usage. These events indicate how a process behaved with respect to memory usage. Like Process events, separate rundown Process Counter events are logged for all active processes at the end of event collection.

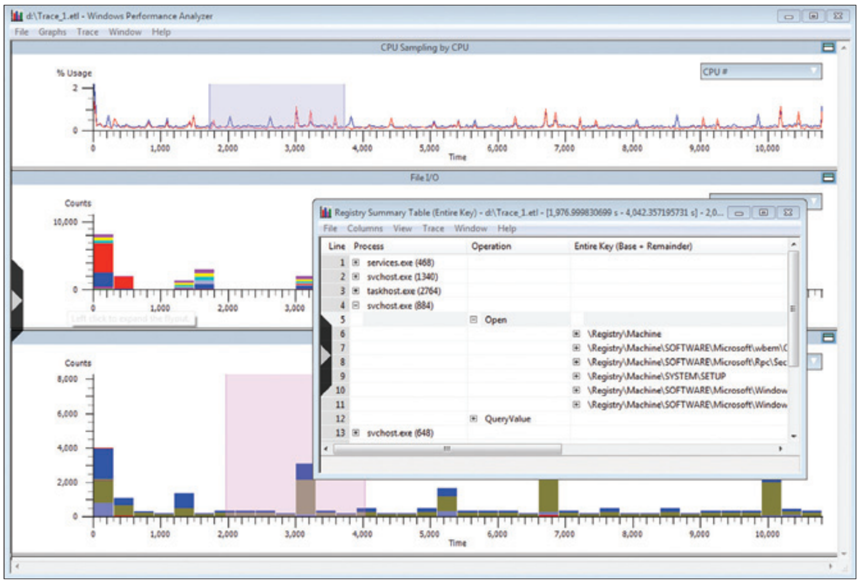
Context Switch, DPC and ISR Events

Context Switch events are logged every time thread switches occur on a CPU and can be used to construct a very accurate history as to which threads have been running and for how long. They occur very frequently and produce a large amount of data. In each switch, two threads are involved. The old thread will give up its share of execution time and hand the execution to the new thread. Thus, Context Switch events contains old and new thread IDs, old and new thread priorities, wait reason and wait time. Context switches can happen for various reasons, including blocking on kernel synchronization objects (events, timers, semaphores and so on.), preemption by a higher priority thread, quantum expiration, and changes in thread affinity. A certain amount of context switches are always expected. However, excessive context switching can be an indication of inefficient use of synchronization primitives and can lead to poor scaling in performance. Enabling call-stacks on Context Switch events allows in-depth analysis on reasons for threads getting switched out.

Deferred Procedure Call (DPC) events are logged when DPCs are executed. DPC is a kernel-mode function executed at elevated interrupt-level execution mode, and it preempts regular thread execution. The DPC event payload includes DPC entry time and routine address. Interrupt Service Routine (ISR) is a similar mechanism, and it runs at a higher execution level than DPC. ISR events have ISR entry time, routine address, ISR vector and ISR return value. DPC and ISR mechanisms are important elements in a Windows driver, as they are typically used for handling hardware interrupts. Drivers and kernel-mode components have a right to use DPC and ISR, but it is strongly recommended that they spend as little time as possible in these elevated modes. DPC and ISR events are used to monitor and verify the behavior of various drivers and kernel-mode components. By comparing the routine addresses against the range information in Image events, one can locate the kernel component responsible for those DPC and ISR events.

In state machine construction, combining Context Switch, DPC and ISR events enables a very accurate accounting of CPU utilization. By setting aside storage for each processor that records its current active thread based on Context Switch, DPC and ISR events, one can monitor—given any timestamp—what each CPU was doing at that time and whose code it was executing. In the state machine simulation method, when a context switch takes place, a CPU object is updated with the new thread ID, and so is the object for the old thread with CPU usage up to the switch. Likewise, DPC and ISR events are attributed to the corresponding kernel-mode components, if needed.

In certain cases, such as with IO, Memory or System Call events, ETW does not record the process or thread IDs in the event header, primarily to reduce the overhead of very frequent events. For such events, the values of thread and process IDs in the header show up as 0xFFFFFFFF (= 4294967295). If Context Switch, DPC and ISR events are tracked as described above, those events can be traced to the thread or kernel-mode component by examining the CPU object for the currently running thread or DPC/ISR.


d:\Trace_1.etl - Windows Performance Analyzer
File Graphs Trace Window Help
CPU Sampling by CPU
% Usage
CPU #
Time
File I/O
Counts
Left click to expand the flyout
Time

Registry Summary Table (Entire Key) - d:\Trace_1.etl - [1,976.999830699 s - 4,042.357195731 s] - 2,0...

File Columns View Trace Window Help

Line	Process	Operation	Entire Key (Base + Remainder)
1	services.exe (468)		
2	svchost.exe (1340)		
3	taskhost.exe (2764)		
4	svchost.exe (884)		
5		Open	
6			\Registry\Machine
7			\Registry\Machine\SOFTWARE\Microsoft\wbem\C
8			\Registry\Machine\SOFTWARE\Microsoft\Rpc\Sec
9			\Registry\Machine\SYSTEM\SETUP
10			\Registry\Machine\SOFTWARE\Microsoft\Window
11			\Registry\Machine\SOFTWARE\Microsoft\Window
12		QueryValue	
13	svchost.exe (648)		

Figure 3 **XperfView in Use**

Memory Events

Memory events denote memory manager (MM) operations. Windows offers Page Fault events, Hard Page Fault events and Virtual Memory events. Memory events tend to be very frequent, especially on a busy system.

A page fault occurs when a sought-out page table entry is invalid. If the requested page needs to be brought in from disk, it is called a hard page fault (a very expensive operation), and all other types are considered soft page faults (a less expensive operation). A Page Fault event payload contains the virtual memory address for which a page fault happened and the instruction pointer that caused it. A hard page fault requires disk access to occur, which could be the first access to contents in a file or accesses to memory blocks that were paged out. Enabling Page Fault events causes a hard page fault to be logged as a page fault with a type Hard Page Fault. However, a hard fault typically has a considerably larger impact on performance, so a separate event is available just for a hard fault that can be enabled independently. A Hard Fault event payload has more data, such as file key, offset and thread ID, compared with a Page Fault event.

Virtual Memory events consist of Virtual Memory Allocation and Virtual Memory Free types and correspond to MM calls to allocate and free virtual memory ranges. Their payload contains the process ID, base address, requested size and flags used in the API call. Virtual Memory events were newly added for Windows 7 and are useful for tracking down leaked calls to the VirtualAlloc function and excessive virtual memory usage by applications.

Memory events tend to be very frequent, especially on a busy system.

Memory event headers do not contain the IDs of the threads and processes that caused the particular activities. The Page Fault event payload, however, has the thread ID of the thread causing the fault. This allows the correlation of Page Fault events to threads and processes through the state machine. The Virtual Memory event payload contains the ID of the process for which virtual memory operations were performed. To track it to the thread making the API call, you need the context switch accounting, described in the previous section.

Next Time

Windows 7 features hundreds of event providers from various components. In the first part of this two-part article series, we have presented some of the core OS ETW events available on Windows 7 and the analysis techniques that we have used for many years. Individual events indicate certain activities in the core OS, but if combined through context-sensitive analysis methods, they can be used to produce meaningful reports that provide insights into patterns and anomalies in resource usage. In Part 2, we plan to cover other core OS ETW events as well as present simple scripts to demonstrate a few basic accounting techniques on some of the OS events introduced in these two parts. We hope that many people will take advantage of the content presented here and that it will lead to the promotion of sound engineering practice, greater software quality and better user experiences. ■

DR. INSUNG PARK *is a development manager for the Windows Instrumentation and Diagnosis Platform Team. He has published a dozen papers on performance analysis, request tracking, instrumentation technology, and programming methodology and support. His e-mail address is insungp@microsoft.com.*

ALEX BENDETOV *is a development lead for the Windows Instrumentation and Diagnosis Platform Team. He works on both Event Tracing for Windows and the Performance Counter technologies. He can be reached at alexbe@microsoft.com.*

A Follow-on Conversation About Threat Modeling

The May 2009 issue of *MSDN Magazine* includes an article titled "A Conversation About Threat Modeling" that discloses a conversation between Paige, a young security neophyte, and Michael, a somewhat jaded security guy. This month I'll take up the conversation where it left off.

Scene I

A small office kitchen, next to the coffeepot.

Paige: Last time we met, you took a good look at my threat model, but you said you'd cover some cryptographic and secure design issues at a later date. Well, welcome to that later date.

Michael: Can I please get a coffee first?

Not waiting for a response, Michael pours himself a huge coffee.

Paige: Er, sure.

Michael: Remind me again what your app is.

Paige: It's a product that allows users to store data on our servers. There's a small piece of client code that pushes the bits to a server set aside for that user. This code can upload files from the user to our back end via the Web server, and the files are stored in the file system along with file metadata stored in SQL Server for rapid lookup. We might store billions of files eventually. The two major environments are domain-joined computers and Internet-joined computers.

Michael: Oh, that's right, I remember now. So much code, so little time. OK, let's go back to your threat model to see which part we're concerned about. Do you have the DFD—the data flow diagram?

The two walk over to Paige's desk. She logs on with her smart card and loads the SDL Threat Modeling Tool.

Paige: Here it is.

Michael looks over the diagram.

Michael: This is the Level-1 diagram, right? It is one level more detailed than the context diagram?

Paige: Yup. We also have a Level-2 diagram, but I don't think we need to go that deep just yet.

Michael: You're right, this is perfect. If more precision is needed as we go through this, we can look at the Level-2 diagram.

Paige: By the way, we don't call them DFDs anymore.

Michael: Er, OK! What are they called, then?

Paige: Application diagrams.

Michael: Whatever floats your boat, I s'pose. We'll be using crayons next. OK, back to the diagram. So the user makes a request of the client application to upload or download files to or from the server, and the server persists that data in the file system at the back end, along with some metadata about the files that is held in SQL Server?

Paige: That's one use; in fact, it's probably the main scenario. Of course, the admin needs to set up, configure and monitor the application; that's what the admin tool does.

Michael: Let's focus on that core scenario, then.

Scene II

Michael is staring intently at the application diagram.

Michael: Let's start by looking at each element in the core scenario, and we'll spell out each of the STRIDE threats.

Paige: STRIDE? Remind me again.

Michael: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege.

Michael starts writing quickly on a piece of paper.

Michael: Take a look at this list:

Item	Type	Subject to
1.0 User	External Entity	SR
9.0 Admin	External Entity	SR
2.0 Client app	Process	STRIDE
3.0 Server app	Process	STRIDE
4.0 SQL Server	Process	STRIDE
8.0 Admin tool	Process	STRIDE
5.0 Files in server file system	Data store	TID
6.0 Files in client file system	Data store	TID
7.0 Metadata	Data store	TID

Paige: Aren't all of these in the Threat Modeling Tool?

Michael: Yes, but I want to show you how the tool arrives at the list. By the way, you don't need to use the threat modeling tool to be SDL-compliant, so long as the threat model is complete and accurate. Basically, each element is subject to a specific set of threats. I think you can work it out from the list.

Paige: Yeah, I get it, but aren't you missing something? All those data flows between the various application elements?

Michael: Yup, but I did that on purpose, because I really don't want to focus on those yet—we discussed them in detail last time.

Send your questions and comments to briefs@microsoft.com.

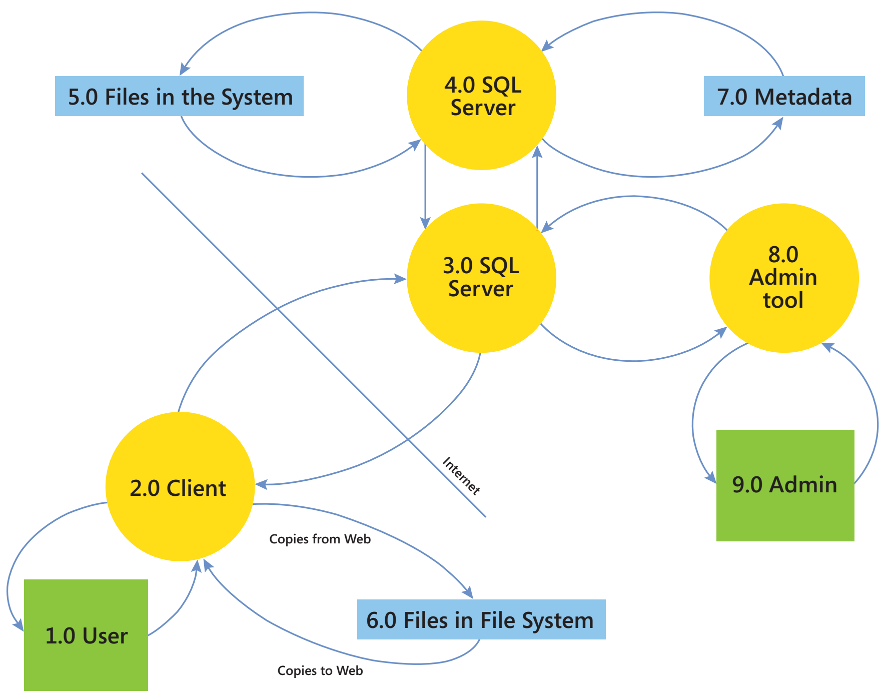


Figure 1: **Threat Model for the Application**

Paige: We did?

Michael: Yes. Look at the Threat Model.

Paige looks over the SDL Threat Modeling Tool

Paige: Oh, I see, that's where we discussed using SSL/TLS to fix the tampering and information disclosure threats to the data flow between the client and server processes, right?

Michael: Good! So here's a question for you. The data that moves from the user to the server and then onto the server's file system—is that data sensitive?

Paige: You asked that last time. Yes, it might be.

Michael: Uh-oh.

Paige: What? The data is encrypted using SSL/TLS, so we're fine, right?

Michael: Not at all. SSL/TLS mitigates the information disclosure threat to the data as it flows between the two processes—the 2.0 Client process and the 3.0 Server process. But after the data leaves that secured tunnel, the data is in the clear, and you're writing data in the clear to the file system.

Paige: Yeah, but what's the risk?

Michael: You tell me!

Paige: I don't understand.

Michael sighs.

Michael: Let's say a client of your application is an employee of a publicly traded company. Let's be more specific: the employee is the CFO of a publicly traded company and he uses your application to store a spreadsheet that shows fiscal data for the current quarter, data that is not public and will not be public until the end of the quarter, when the company announces its earnings. Let's say a hacker breaks into your system, gets that data, and uses it to sell or buy stock in the company. That might be insider trading.

Paige: Uh-oh.

Michael: Uh-oh, indeed. This is serious. The CFO does not have appropriate controls on this sensitive data, which might be a violation of SOX regulations.

Paige: You said "might" a lot.

Michael: You bet I did. Do I look like a lawyer to you? So back to my original question. Is this situation something you care about?

Paige: Well, not really. I think our terms state that you shouldn't use our service for ultra-sensitive data. But I'll play along. Let's assume I say, "Yes, we care about this scenario." Now what?

Michael: My first bit of advice would be to consult your attorneys to make sure you're not putting the company at risk with this scenario. But let's assume they say you can go for it, but you need to be sure you protect the data at the back end.

Paige: What you're trying to say is that the data held in data store 5.0, the file system at the server, is subject to information disclosure and we need to mitigate that threat. Am I right?

Michael: You're spot on. So how do you fix it?

Paige: ACLs.

Michael: Why access control lists?

Paige: We can limit access to only valid users of the data.

Michael: So how does the server application read and write the data?

Paige: Oh, let's assume the process runs as a unique identity. We'll call it FooID. We could apply an ACL to the files that allows FooID as well as to the valid users' access to the files.

Michael: It won't work; it's not secure.

Paige: Why not?

Michael: If I'm an attacker, I can compromise the server process by running as FooID and then run my malicious code on that server. *Voila*, my code is running as FooID, and I own the data!

Paige looks dejected.

Paige: Humph.

Michael: You have to use encryption.

Paige: Of course! The server will just read and write encrypted blobs, and if an attacker compromises the server, he still can't get at the data unless he can break the encryption.

Paige perks up.

Michael: Now the fun really starts. We touched on some of the crypto issues last time, especially as they relate to keys.

Paige: What do you mean?

Michael: OK, how are you going to encrypt the data?

Paige: The user types in a password in the client-side application, and the client-side application encrypts the data with the password and sends the encrypted blob across the wire to the server. The server writes metadata to SQL Server and then writes the encrypted blob to the server file system.

Michael: What's in the metadata?

Paige: Owner's identity, the file size, the filename, the time it was written to the file system and last read time. That kind of stuff. I know this information is held in the file system, too, but it's way quicker to do a lookup in something designed to store this kind of data: a SQL Server database.

Michael: Good, I'm glad it doesn't store data held within the file!

Paige: Why?

Michael: For a couple of reasons. First, it would mean your server application has access to data in the clear, and to get that data requires your server process to decrypt the data.

Paige: So?

In a loud but not-quite-shouting voice, Michael responds.

Michael: Because it means your server application needs to know a decryption key, which means you get into all sorts of truly horrible key management games. If at all possible, you want to stay out of that business! There are ways you can do this cleanly by having multiple keys, but I don't want to explain that right now. If ever! If you really want to understand this, read up on how Microsoft encrypts files using the Encrypting File System, or EFS.

Paige: Could we use EFS?

Michael: Possibly. It depends on your clients. What platforms do you support at the client?

Paige: We'll ship at the end of the year on Windows and then a couple of months later on Linux.

Michael: No Solaris?

Paige: What's Solaris?

Michael sniggers and ignores Paige's reply.

Michael: You can't use EFS because it requires Windows accounts. So you have to encrypt the data using different technology. It'd be great if you could use EFS, or even Data Protection API, known as DPAPI, because both use the same underlying crypto technology and can seamlessly encrypt and decrypt data by using keys derived from the user's password. Oh well. Let's see what else we can do.

Paige: Can we use an encryption library?

Michael: Of course we can. In fact, that's a much better idea than something I heard the other day.

Paige: What?

Michael: Someone asked me if it would be OK to create his own crypto algorithm.

Paige: You said no, right?

Michael: Of course I said no. What would you expect me to say? I also made it pretty clear that it's a complete violation of SDL policy, and he should not even contemplate the possibility of using any homegrown crypto.

Paige: So what should we do?

Michael: Because your client code is C#, you could use the .NET System.Security.Cryptography namespace. It's available in Mono, which means you could call it from Linux. I haven't tried it, but you could do an experiment. You'd also need to chat with the lawyers to make sure there are no licensing issues.

Paige: What licensing issues?

Michael: It's third-party code. Who knows what the license says.

Paige: OK, so we encrypt the data with the user's password, send the blob …

Michael: No. Non. Nyet. Nada. Nope. You *do not* use the user's password as an encryption key; you derive the encryption key from the password. Passwords are way too easy to guess.

Paige: How does one "derive" a key?

Michael smiles.

Michael: With a key-derivation function.

Paige: Mr. Smarty. Could you be a little more precise?

Michael: Sure. You pass the key into a function such as Rfc2898DeriveBytes in .NET, along with a salt. A salt is just a unique number that makes it harder to perform certain attacks. It's an iteration count, usually in the tens if not hundreds of thousands. The function takes the password and munges it thousands of times with the salt. This is often referred to as "stretching the password." At the end of the operation, which usually takes less than a second, you get some bytes, and you can use those bytes as keys. Key derivation not only makes it hard to guess the key, it also makes it hard to mount high-speed password-guessing attacks because the attacker has to go through the iteration count, too. So if an attacker could normally test 1,000,000 passwords per second, with an iteration count of 100,000, he's reduced to 10 per second! Cool, huh?

Paige: Very cool. So we use that key to encrypt the data, using, say, Advanced Encryption Standard?

Michael: Yes. Of course, all this does is encrypt the data. It doesn't provide any form of integrity check, but that's pretty easy. You can derive another key and use that to create a message authentication code and store that along with the metadata. Don't use the same key for encryption and integrity checking. Derive a second key, and use that.

Adam, another security guy, walks by muttering.

Adam: You security wizards always want to go depth-first into crypto and the like. But the attackers go for the weak link.

Michael: Adam's right, security people tend to dig deep quickly. And I'm guilty as charged, but I want to get this out of the way.

Paige: Er, OK. Anything else?

Michael: Well, there's also the sticky problem of users forgetting their passwords. I would give users the opportunity to back up their password to a USB stick or something, and make them aware that we don't have the password and that if they forget it, there's no way we can bring the data back from the dead!

Paige: Are we done?

Michael: For the moment, yes. It's a big and important section of the threat model, and I hope this gives you an idea of some of the trade-offs you need to make when building secure applications.

Paige: I do now. Thanks. ■

Michael Howard *is a senior security program manager at Microsoft who focuses on secure process improvement and best practices. He is the coauthor of five security books, including "Writing Secure Code for Windows Vista," "The Security Development Lifecycle," "Writing Secure Code" and "19 Deadly Sins of Software Security."*

SCOTT ALLEN

Search Engine Optimization with ASP.NET 4.0, Visual Studio 2010 and IIS7

Anyone with a public Web site knows that search engines play a key role in bringing visitors to the site. It's important to be seen by the search engines and rank highly in their query results. Higher rankings can bring you more visitors, which can lead to more paying customers and higher advertisement revenue. Search engine optimization (SEO) is the practice of fine-tuning a site to achieve higher rankings in search results. In this article, we'll take a look at SEO practices you can apply when using the latest Microsoft Web technologies.

SEO Basics

There are many factors in play when a search engine formulates the relative rank of your site, and some of the more important factors are not under your direct control. For example, we know search engines like to see incoming links to your site. An incoming link is a hyperlink on an outside domain that points into your domain. When a search engine sees many incoming links to a site, it assumes the site has interesting or important content and ranks the site accordingly. The SEO community describes this phenomenon using technical terms like "link juice" and "link love." The more "link juice" a site possesses, the higher the site will appear in search results.

If your site is interesting, then the rest of the world will naturally start adding links to your site. Because Visual Studio doesn't come with a "Make My Site More Interesting" button, you'll ultimately have to work hard at providing link-worthy content for the Web.

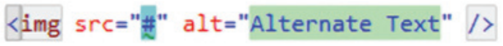
Once you have great content in place, you'll want to make sure the search engines can find and process your content. We don't know the exact algorithms used by search engines like Bing.com and Google. However, most search engines have published design and content guidelines you can follow to help boost your ranking. The Internet community has also compiled an extensive amount of knowledge acquired through experimentation, trial and error. Here's the key: you want to think like a search engine. Search engines don't execute scripts or recognize the shapes in the images on your site. Instead, they methodically follow links to parse, index and rank the content they find in HTML. When thinking like a search engine, you'll focus on your HTML.

This column is based on a prerelease version of Visual Studio 2010. Details are subject to change.

Send your questions and comments to xtrmasp@microsoft.com.

Code download available at code.msdn.microsoft.com/mag200909ExtremeASPN

Figure 1 **Only Four Keystrokes Generated This Markup**



```
<img src="#" alt="Alternate Text" />
```

Quick and Valid HTML

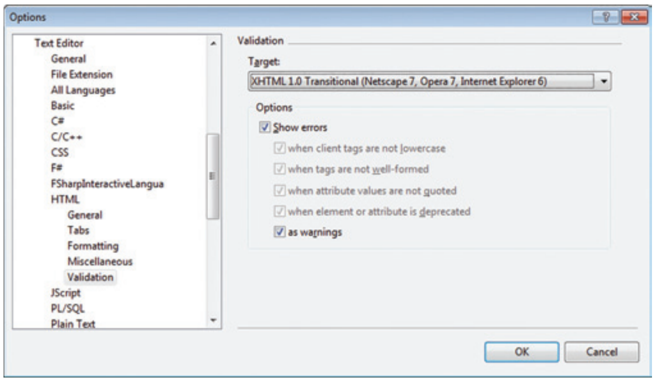
Visual Studio has a long history in WYSIWYG development for both the desktop and the Web. The Web Forms designer allows you to drag and drop server controls on the design surface, and set values for controls in the Properties window. You can quickly create a Web page without ever seeing HTML. If you're focused on HTML, however, you'll want to work in the Source view window. The good news is you can work in the Source view without sacrificing speed or accuracy in Visual Studio 2010.

You can work in the Source view without sacrificing speed or accuracy in Visual Studio 2010.

Visual Studio 2010 is going to ship with a number of HTML IntelliSense code snippets for creating common HTML tags and server-side controls using a minimal number of keystrokes. For example, when you are in the source view of an .aspx file, you can type img and then hit the Tab key to generate the markup shown in **Figure 1**. Only four keystrokes give you more than 20 of the characters you needed to type!

Notice how the editor highlights the src and alt values in **Figure 1**. When using code snippets, you can tab between highlighted areas and begin typing to overwrite the values inside. This feature is another productivity bonus that saves you the effort of navigating to the proper insertion point and manually deleting the existing value.

Both ASP.NET Web Forms and ASP.NET MVC projects will have HTML snippets available in Visual Studio 2010 to create everything from ActionLinks to XHTML DOCTYPE declarations. The snippets are extensible, customizable and based on the same



Validation

Target: XHTML 1.0 Transitional (Netscape 7, Opera 7, Internet Explorer 6)

Options

- [✓] Show errors
  - [✓] when client tags are not lowercase
  - [✓] when tags are not well-formed
  - [✓] when attribute values are not quoted
  - [✓] when element or attribute is deprecated
  - [✓] as warnings

Figure 2 **Validation Settings**

snippet engine that has been available since Visual Studio 2005. See Lorenzo Minore's MSDN article for more details on snippets (msdn.microsoft.com/en-us/magazine/cc188694.aspx).

Validation

Creating valid HTML is crucial if you want search engines to index your site. Web browsers are forgiving and will try to render a page with malformed HTML as best they can, but if a search engine sees invalid HTML, it may skip important content or reject the entire page.

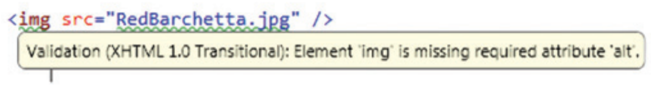
Since there are different versions of the HTML specifications available, every page you deliver from your application should include a DOCTYPE element. The DOCTYPE element specifies the version of HTML your page is using. Web browsers, search engines and others tools will examine the DOCTYPE so that they know how to interpret your markup. Visual Studio will place a DOCTYPE in the proper locations when you create new Web form pages and master pages. The default DOCTPYE, as shown in the following code snippet, specifies that the page will comply with the XHTML 1.0 specification:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Although you don't need to know all the subtle differences between the HTML specifications, you do need to know if your page conforms to a specific DOCTYPE. Visual Studio has included HTML validation features since the 2005 release, and validation is on by default. You can control the validation settings and validation target type by navigating to the Tools | Options | Text Editor | HTML | Validation settings, as shown in **Figure 2**).

The "as warnings" setting means HTML validation problems will not stop your build, but will show as warnings in the Error window of Visual Studio. In the source view for Web forms, the text editor will draw your attention to HTML validation errors using squiggly lines. You can mouse over the element to see the exact error message, as we see in **Figure 3**.

Figure 3 **Validation Error Message**



```
<img src="RedBarchetta.jpg" />
```

Validation (XHTML 1.0 Transitional): Element 'img' is missing required attribute 'alt'.

Descriptive HTML

The img tag in **Figure 3** is a good example of how you need to think like a search engine. As I said earlier, a search engine doesn't see or interpret the shapes and words in an image, but we can give the search engine some additional information about the graphical content by using the alt attribute. If the image is a company logo, your alt text might be "company logo," but it would be better to include the name of the company in the logo's alt text. A search engine will use the alt text as another clue in understanding the theme and essence of the page.

Search engines are always looking for these types of clues, and to the search engine some clues are more important than others. For example, we typically use header tags, like h1 tags, to make certain pieces of content standout in a page. Search engines will generally give more weight to a keyword inside of an h1 tag than if the same keyword were inside a normal paragraph. You'll want to make sure your h1 content is descriptive and uses keywords related to the theme of your page. A best practice for SEO work is to always include at least one H1 tag in every page.

If you look back at the headers I've chosen for this article, you'll see they revolve around phrases like "Valid HTML," "SEO Basics," and so on. These are all descriptive phrases that will give both the reader and the search engine a good idea of what the article is about.

Descriptive Titles and Metadata

Another important area for descriptive keywords is inside the head tag. The head section from one of the pages in the associated code download is shown here:

```
<head runat="server">
 <title>Programming meta tags in ASP.NET 4.0</title>
 <meta name="keywords" content="ASP.NET, ASP.NET 4.0, SEO, meta" />
 <meta name="description" content=
  "How to use Page.MetaKeywords and Page.MetaDescription in ASP.NET" />
</head>
```

The words inside the page title tag are heavily weighted, so you'll want to choose a good title. The head tag can also enclose meta tags. You'll want to use two meta tags for SEO work—one to set the page's associated keywords and one to set the page's description. Visitors will generally not see this meta information, but some search engines do display the meta description of a page in search results. The meta keywords are another place to advertise the real meaning of your page by feeding the search engine important words to associate with the page.

If you are building dynamic content, or changing the title and meta data on a frequent basis, then you don't want to hard code this content in an .aspx file. Fortunately, Web Forms in ASP.NET 4.0 makes it easy to manipulate the title, keywords and description of a page from code-behind:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
    {
      Page.Title = "Programming meta tags in ASP.NET 4.0";
      Page.MetaKeywords = "ASP.NET 4.0, meta, SEO, keywords";
      Page.MetaDescription =
        "How to use Page.Keywords and Page.Description in ASP.NET";
    }
}
```

SEO No-Nos

Some sites attempt to game the Internet search engines by filling their pages with irrelevant keywords, too many keywords, or by duplicating keywords repeatedly. This practice (known as "keyword stuffing") is an attempt to gain a high search engine rank for specific search terms without providing useful content for real visitors. The unfortunate visitors who land on such a site are invariably disappointed because they don't find any content of substance, but the visit still counts as a hit when the site totals its advertising revenue.

Search engines try to detect deceptive behavior like keyword stuffing to protect the quality of their search results. You don't want a search engine to accidently categorize your site as misleading because you've used too many keywords in too many places. Search engine penalties can range from lowering the relative importance of a page in search results, to dropping the content of a site from the search index entirely.

Another practice to avoid is serving different content to a search engine crawler than you would serve to a regular visitor. Some sites will do this by sniffing the user agent header or IP address of an incoming request. Although you might be able to think of some useful features for this type of behavior, too many sites have used this technique to hide well-known malware and phishing content from search engines. If a search engine detects this behavior (known as cloaking), you'll be penalized. Stay honest, provide good content, and don't try to game or manipulate the search engine results.

You can see how to use the page's Title, MetaKeywords and MetaDescription properties during the Page_Load event. The Title property has been in ASP.NET since version 2.0, but MetaKeywords and MetaDescription are new in ASP.NET 4.0. Although we are still using hard-coded strings, you could load these property values from any data source. You could then allow someone in charge of marketing the Web site to tweak the meta data for the best search engine results, and they would not have to edit the source code for the page.

While effective keywords and descriptions might give you a bit of an edge in search engine results, content is still king. We'll return to see more HTML tips later in the article, but in the next couple of sections we'll see how URLs can play an important role in how your content is located and ranked.

Canonical URLs

Duplicate content generally presents a problem for search engines. For example, let's say the search engine sees a recipe for your famous tacos at two different URLs. Which URL should the search engine prefer and provide as a link in search results? Duplicate content is even more of an optimization problem when it comes to incoming links. If the "link love" for your taco recipe is distributed across two different URLs, then your famous taco recipe might not have the search engine ranking it deserves.

Unfortunately, you might be duplicating content without realizing it. If search engines can read your site from a URL with a www prefix and without a www prefix, they'll see the same content under two different URLs. You want both URLs to work, but you want just one URL to be the standard or canonical URL.

As an example, consider the Microsoft.com Web site. Both www.microsoft.com and microsoft.com will take you to the same content. But, watch closely if you go to the home page using microsoft.com. The Microsoft site will redirect your browser to www.microsoft.com. Microsoft uses redirection to enforce www.microsoft.com as its canonical URL.

Luckily, redirecting visitors to your canonical URL is easy with ASP.NET. All you need to do is provide some logic during the application pipeline's BeginRequest event. You can do this by implementing a custom HTTP module, or by using an Application_BeginRequest method in global.asax. **Figure 4** is what the logic would look like for this feature.

The code in **Figure 4** is using another new feature in ASP.NET 4.0—the RedirectPermanent method of the HttpResponse object. The traditional Redirect method in ASP.NET sends an HTTP status code of 302 back to the client. A 302 tells the client that the resource temporarily moved to a new URL, and the client should go to the new URL, just this once, to find the resource. The RedirectPermanent method sends a 301 code to the client. The 301 tells the client that the resource moved permanently, and it should look for the resource at the new URL for all future requests. Notice the call to RedirectPermanent also uses a new feature in C# 4.0—the named parameter syntax. Although this syntax isn't required for the method call, the named parameter syntax does make the intent of the parameter explicit.

With a redirect in place, both Web browsers and search engines should be using only your canonical URL. Your "link love" will consolidate and search engine rankings should improve.

Descriptive URLs

In the January 2009 issue of *MSDN Magazine* I wrote about how to use the routing features of .NET 3.5 SP1 with ASP.NET Web Forms (msdn.microsoft.com/en-us/magazine/2009.01.extremeaspnet.aspx). As I said

Figure 4 **RedirectPermanent Method of the HttpResponse Object**

```
void Application_BeginRequest(object sender, EventArgs e)
{
    HttpApplication app = sender as HttpApplication;

    if (app != null)
    {
        string domain = "www.odetocode.com";
        string host = app.Request.Url.Host.ToLower();
        string path = app.Request.Url.PathAndQuery;

        if (!String.Equals(host, domain))
        {
            Uri newURL = new Uri(app.Request.Url.Scheme +
                                 "://" + domain + path);

            app.Context.Response.RedirectPermanent(
                newURL.ToString(), endResponse: true);
        }
    }
}
```

URL Rewrite by Carlos Aguilar Mares

URL Rewrite for IIS 7.0 is a tool Microsoft makes available for download from iis.net/extensions/URLRewrite. This tool can perform all of the URL canonicalization work for you without requiring any code. The tool will do host header normalization, lowercasing and more (as described in this blog post: ruslany.net/2009/04/10-url-rewriting-tips-and-tricks/). The tool can also help you "fix" broken links by rewriting or redirecting using a map, so you don't need to even change your application/HTML. See: blogs.msdn.com/carlosag/archive/2008/09/02/IIS7UrlRewriteSEO.aspx

URL Rewrite can also do the "descriptive" URLs for any version of ASP.NET and its performance is far superior to any other existing option, including ASP.NET routing, because the tool works with kernel-mode caching.

then, the clean and descriptive URLs you can achieve with routing are important to both users and search engines. Both will find more meaning in a URL like /recipes/tacos than they will in /recipe.aspx?category=40&topic=32. In the former, the search engine will consider "recipes" and "tacos" as important keywords for the resource. The problem with the latter URL is that many search engine crawlers don't work well when a URL requires a query string with multiple parameters, and the numbers in the query string are meaningless outside the application's backend database.

The ASP.NET team has added some additional classes to the 4.0 release that make routing with Web Forms easy. In the code download for this article, I've re-implemented January's demo Web site with the new classes in ASP.NET 4.0. Routing begins by describing the routes your application will process during the Application_Start event. The following code is a RegisterRoutes method that the site invokes during the Application_Start event in global.asax:

```
void RegisterRoutes()
{
    RouteTable.Routes.Add(
        "Recipe",
        new Route("recipe/{name}",
            new PageRouteHandler("~/RoutedForms/RecipeDisplay.aspx",
                                 checkPhysicalUrlAccess:false)));
}
```

Figure 5 **Get Name Parameter From RouteData to Display Information About a Recipe**

```
private void DisplayRecipe()
{
 var recipeName = RouteData.Values["name"] as string;
 if (recipeName != null)
 {
  var recipe = new RecipeRepository().GetRecipe(recipeName);
  if (recipe != null)
  {
   _name.Text = recipe.Name;
   _ingredients.Text = recipe.Ingredients;
   _instructions.Text = recipe.Instructions;
  }
 }
}
```

If you review my January article, you'll remember how every route must specify a route handler. In RegisterRoutes, we are setting the handler for the "Recipe" route to an instance of the new PageRouteHandler class in ASP.NET 4.0. The routing engine will direct any incoming request URLs in the form of recipe/{name} to this route handler, where {name} represents a route parameter the routing engine will extract from the URL.

A Web Form has access to all of the route parameters the routing engine extracts from the URL, via a RouteData property. This property is new for the Page class in 4.0. The code in **Figure 5** will get the name parameter from RouteData and use the name to look up and display information about a recipe:

One of the great features of the routing engine is its bidirectional nature. Not only can the routing engine parse URLs to govern HTTP requests, but it can also generate URLs to reach specific pages. For example, if you want to create a link that will lead a visitor to the recipe for tacos, you can use the routing engine to generate a URL based on the routing configuration (instead of hard-coding the URL). ASP.NET 4.0 introduces a new expression builder you can use in your markup to generate URLs from the routing configuration table:

```
<asp:HyperLink NavigateUrl="<%$ RouteUrl:RouteName=recipe,name=tacos %>"
               Text="Titillating Tacos" runat="server">
</asp:HyperLink>
```

The preceding code shows the new RouteUrl expression builder in action. This expression builder will tell the routing engine to generate a link for the route named "recipe" and include a name parameter in the URL with the value "tacos". The preceding markup will generate the following HTML:

```
<a href="/recipe/tacos">Titillating Tacos</a>
```

The preceding URL is friendly, descriptive, and optimized for a search engine. However, this example brings up a larger issue with ASP.NET. Server controls for Web Forms often abstract away the HTML they produce, and not all the server controls in ASP.NET are search engine friendly. It's time we return to talk about HTML again.

HTML Mistakes

If we created a link to the taco recipe using a LinkButton instead of a Hyperlink, we'd find ourselves with different markup in the browser. The code for the LinkButton and the HTML it generates is shown here:

```
<asp:LinkButton runat="server" Text="Tacos"
  PostBackUrl="<%$ RouteUrl:RouteName=recipe,name=tacos %>">
</asp:LinkButton>

<!-- generates the following (excerpted): -->
<a href="javascript:WebForm_DoPostBackWithOptions(...)">Tacos</a>
```

We still have an anchor tag for the user to click on, but the anchor tag uses JavaScript to force the browser to postback to the server. The LinkButton renders this HTML in order to raise a server-side click event when the user clicks on the link. Unfortunately, JavaScript postback navigation and search engines don't work together. The link is effectively invisible to search engines, and they may never find the destination page.

Because server-side ASP.NET controls abstract away HTML, you have to choose your server controls wisely. If you want complete

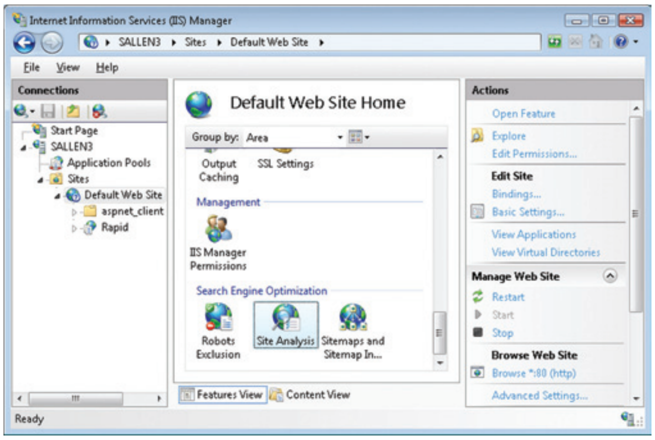


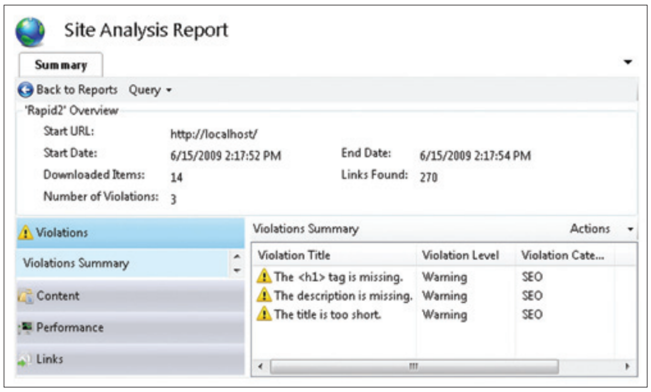
Figure 6 **IIS 7 Manager**

control over HTML markup in an ASP.NET environment, then you should consider using the ASP.NET MVC framework. Server controls are verboten when using the MVC framework, and the infrastructure and APIs are in place to work with only HTML markup.

If you are using ASP.NET Web Forms and are optimizing for search engines, you'll want to view the HTML source produced by server controls. Every Web browser will give you this option. In Internet Explorer, use the View -> Source command. Be careful with any control that renders a combination of HTML and JavaScript in navigational scenarios. For example, using a DropDownList with the AutoPostBack property set to true will require JavaScript to work. If you rely on the automatic postback to navigate to new content, you'll be making the content invisible to search engines.

Obviously, AJAX-heavy applications can present a problem for search engines. The UpdatePanel control and content generated by Web service calls from JavaScript are not friendly to search engines. Your safest approach for SEO work is to place content directly into your HTML to make it easily discoverable.

After you've tweaked your HTML, your keywords, and your URLs, how do you measure the results? Although your search engine ranking is the ultimate judge of your SEO effort, it would be nice if you could find any problems before a site goes live and a search engine crawls your pages. Although Visual Studio can tell you about HTML validation problems, it doesn't warn you about missing metadata and canonical URLs. This is the job of a new product—the IIS SEO Toolkit.


Site Analysis Report

Summary

Back to Reports Query

'Rapid2' Overview

Start URL:	http://localhost/		
Start Date:	6/15/2009 2:17:52 PM	End Date:	6/15/2009 2:17:54 PM
Downloaded Items:	14	Links Found:	270
Number of Violations:	3		

Violations | Violations Summary | Content | Performance | Links

Violations Summary Actions

Violation Title	Violation Level	Violation Cate...
The <h1> tag is missing.	Warning	SEO
The description is missing.	Warning	SEO
The title is too short.	Warning	SEO

Figure 7 **Report Summary**

The IIS SEO Toolkit

The IIS SEO Toolkit is a free download for IIS 7 and is available from iis.net/extensions/SEOToolkit. The toolkit includes a crawling engine that will index your local Web application just like a search engine, and provide you with a detailed site analysis report. The toolkit can also manage robots.txt and sitemap files. The robots file uses a standardized format to tell search engines what to exclude from indexing, while sitemap files can point search engines to content you want to include. You can also use sitemap files to tell the search engine the priority, rate of change and the date a resource changed.

For SEO work, the site analysis report is invaluable. The report will tell you everything about your site from the perspective of a search engine. After you've installed the toolkit, a Site Analysis option will appear for your sites in the IIS 7 Manager window, as shown in **Figure 6**. Double-clicking the icon will take you to a list of previously run reports, with an Action option of running a new analysis. Running an analysis is as easy as pointing the tool to a local HTTP URL and clicking OK. When the analysis is finished, the toolkit will open a report summary, as **Figure 7** shows.

The toolkit applies a number of rules and heuristics to make you aware of SEO- and performance-related problems. You can find broken links, missing titles, descriptions that are too short, descriptions that are too long and a host of other potential issues. The toolkit will analyze links and provide reports on the most linked pages, and the paths a visitor would need to follow to reach a specific page. The toolkit even provides a textual analysis of each page's content. You can use this analysis to find the best keywords for a page.

The IIS SEO Toolkit allows you to discover the SEO work you need to perform and to validate any SEO work you've already completed. At the time of writing, the toolkit is in a beta 1 release. You can expect that future versions will continue to add rules and analysis features, in addition to some intelligence that can automatically fix specific problems for you.

Easy and Effective

Even if you have the greatest content in the world, you need to make the content discoverable for search engines to bring you visitors. SEO is the practice of thinking like a search engine and making your site appeal to the crawlers and ranking algorithms. Visual Studio 2010 and ASP.NET 4.0 are introducing new features to make SEO work easier in .NET 4.0, while the IIS SEO Toolkit is a fantastic tool dedicated to making your site better for search engines. Using the three tools in combination can make your SEO work both easy and effective.

I want to thank Carlos Aguilar Mares and Matthew M. Osborn of Microsoft for their help in writing this article. ■

K. SCOTT ALLEN *is a member of the Pluralsight technical staff and founder of OdeToCode. You can reach Allen at scott@OdeToCode.com or read his blog at odetocode.com/blogs/scott.*

Charting with DataTemplates

For WPF programmers, one major revelation about the power of the DataTemplate comes with a demonstration of how a little piece of XAML (**Figure 1**) can turn business objects into bar charts (**Figure 2**).

Following the exhilaration at seeing the chart's rendering, the technique unfortunately seems to "stall out." Enhancing the simple bar chart becomes awkward, and using data templates for other types of common charts—such as pie charts and line charts—seems nearly impossible. That's too bad, because using data templates in this way is simply too powerful a technique to be relegated to unadorned bar charts.

In this article, I will show you a few techniques to get around the apparent limitations. There will be some code involved to help out with the details that XAML can't manage by itself, but the code will often be generalized enough to use for other applications. The goal is always to keep enough of the visual design in XAML so that changes and experimentation are easy.

Of course, the DataTemplate isn't a universal solution for charting. You might find an actual charting package more suited to your needs. You should also be aware that performance issues might result from an ItemsControl or ListBox with thousands of data items and a DataTemplate with many data bindings. I examined solutions to that problem in the article "Writing More Efficient ItemsControls" in the March 2009 issue of *MSDN Magazine* (msdn.microsoft.com/en-us/magazine/dd483292.aspx).

Basic Concepts

The downloadable code for this article consists of one Visual Studio solution named ChartingWithDataTemplates, containing one DLL project named ChartingLib and nine WPF application projects. The file shown in **Figure 1** is the Window1.xaml file from the SimpleBarChart project.

Some of the classes in the ChartingLib DLL take the form of business objects to supply the data for the sample programs. For this first example, the Doodad class implements INotifyPropertyChanged and includes the properties ModelName and BaseCost. The DoodadCollection class simply derives from ObservableCollection<Doodad> and contains a collection of Doodad objects. The DoodadPresenter class instantiated as a XAML resource in **Figure 1** defines a DoodadCollection property, creates all the random data, and includes a timer to change the data dynamically over time.

Figure 1 **XAML to Display a Bar Chart**

```
<Window x:Class="SimpleBarChart.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:charts="clr-namespace:ChartingLib;assembly=ChartingLib"
        Title="Simple Bar Chart">
    <Window.Resources>
        <charts:DoodadPresenter x:Key="doodadPresenter" />
    </Window.Resources>

    <ItemsControl
            ItemsSource="{Binding
                            Source={StaticResource doodadPresenter},
                            Path=DoodadCollection}"
            VerticalAlignment="Center">
        <ItemsControl.ItemTemplate>
            <DataTemplate>
                    <Rectangle Height="{Binding BaseCost}"
                           VerticalAlignment="Bottom"
                           Fill="Blue"
                           Margin="3">
                        <Rectangle.ToolTip>
                            <StackPanel Orientation="Horizontal">
                                <TextBlock Text="{Binding ModelName}" />
                                <TextBlock
                                Text="{Binding BaseCost,
                                           StringFormat=': {0:C0}'}" />
                            </StackPanel>
                        </Rectangle.ToolTip>
                    </Rectangle>
            </DataTemplate>
        </ItemsControl.ItemTemplate>

        <ItemsControl.ItemsPanel>
            <ItemsPanelTemplate>
                <UniformGrid Rows="1" IsItemsHost="True" />
            </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
    </ItemsControl>
</Window>
```

It is important that the business object used for the items implement the INotifyPropertyChanged interface so that changes in the underlying data are reflected visually in the chart. The collection object should implement ICollectionChanged so that changes to the collection itself—such as items added to or removed from the collection—are reflected in the ItemsControl. The ObservableCollection class implements ICollectionChanged and is a very popular choice for this purpose.

Send your questions and comments to mmnet30@microsoft.com.

Code download available at code.msdn.microsoft.com/mag200909Foundations.

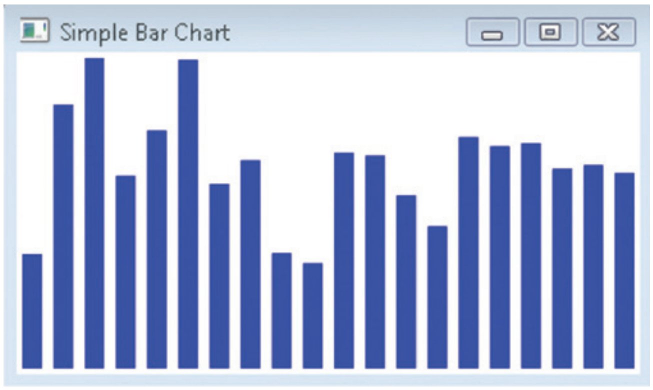


Figure 2 **Bar Chart Displayed by the SimpleBarChart Project**

The sample programs for this article use an ItemsControl for displaying the items. You can alternatively use a ListBox if you need to provide a way for the user to select an item. Each item is displayed based on the DataTemplate set to the ItemTemplate property of the ItemsControl. The DataTemplate contains bindings to the properties of the object stored in the collection, in this example Doodad.

Notice the ToolTip, which also contains bindings to the properties of the Doodad class. The ToolTip is definitely the easiest way to include textual information about each item in the chart.

Besides the DataTemplate, you'll also need to set the ItemsPanel property of the ItemsControl to an ItemsPanelTemplate containing an appropriate panel. For bar charts arranged horizontally, you can use either a UniformGrid with the Rows property explicitly set to 1, or a StackPanel with the Orientation property set to Horizontal.

The single-row UniformGrid and the horizontal StackPanel seem similar, but they actually have quite different layout behaviors. The single-row UniformGrid sizes its children so that they all have the same width and fit within the dimensions of the panel. The horizontal StackPanel bases its own size on the composite width of its children. Use UniformGrid if you want to maintain a constant width for the chart regardless of how many items are displayed. Use StackPanel if you want to maintain a constant size on each bar, which you'll need to explicitly set. A chart that uses a StackPanel for its ItemsPanel will change size depending on the number of items, so you might want to put the ItemsControl in a ScrollViewer to prevent the chart from taking up too much space on the screen.

Utility and Aesthetics

The SimpleBarChart program displays bars indicating the value of the BaseCost property defined by the Doodad class. The Doodad class also defines AdditionalCost and ExtraCost properties, so one possible enhancement is making a stacked bar chart to display all three of these properties, as demonstrated in the StackedBarChart program. The DataTemplate is shown in **Figure 3** and the result in **Figure 4**.

Notice that a StackPanel encloses the three Rectangle elements; it is this StackPanel that is given the VerticalAlignment and Margin properties. I've eliminated the ToolTip just to make the XAML simpler; if you add it back in, it should be attached to the StackPanel rather than to any of the Rectangles.

Figure 3 **DataTemplate for a Stacked Bar Chart**

```
<DataTemplate>
    <StackPanel VerticalAlignment="Bottom"
                Margin="3">
        <Rectangle Height="{Binding ExtraCost}"
                   Fill="Red" />
        <Rectangle Height="{Binding AdditionalCost}"
                   Fill="Green" />
        <Rectangle Height="{Binding BaseCost}"
                   Fill="Blue" />
    </StackPanel>
</DataTemplate>
```

The primary colors are a bit garish, but you can easily tone them down by putting the StackPanel in a single-cell grid, which then contains another Rectangle to overlay the other three. This fourth Rectangle is colored with a LinearGradientBrush with various degrees of transparent white. The StackedBarChartWithGradientBrush project uses this technique for the visuals in **Figure 5**.

One of the great benefits that result from defining the bar chart visuals in XAML is the ease of replacing the bar with some other type of graphic, or of coloring the Rectangle with an image brush or drawing brush. Suppose the Doodad is actually a toy robot and you want to use a graphical representation of that toy in the bar chart. **Figure 6** shows a DataTemplate containing a Path with a simple figure defined by coordinates between 0 and 1. A ScaleTransform gives each of these figures a uniform width. The horizontal scaling is bound to the BaseCost property. The result is shown in **Figure 7**.

Is it possible to use different colors in the bar chart? Absolutely. The Doodad class contains an integer property named ProductType with values from 0 through 3. You can base the color of the bar on this ProductType by providing a simple binding converter that converts an integer to a Brush. The IndexToBrushConverter defines a public property named Brushes of type Brush array:

```
public Brush[] Brushes { get; set; }
```

To keep this converter generalized, the array is not required to have a fixed number of brushes. Instead, a modulo operation is applied to the incoming integer value based on the size of the array:

```
return Brushes[(int)value % Brushes.Length];
```

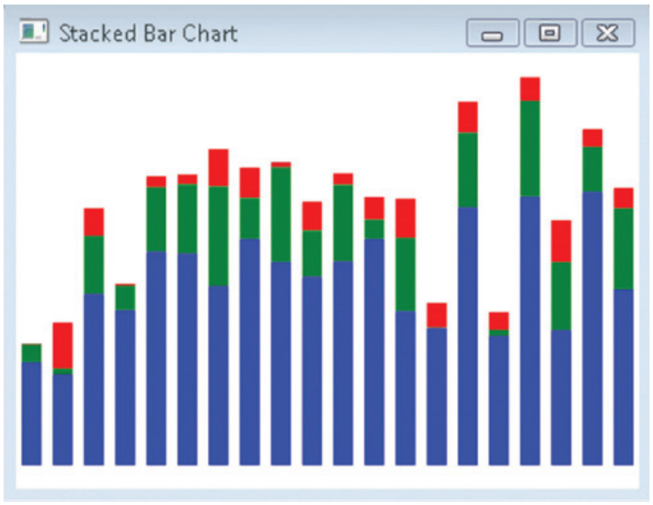


Figure 4 **The StackedBarChart Program Display**

In the XAML file, the IndexToBrushConverter is instantiated as a resource and initialized with an array of brushes:

```
<charts:IndexToBrushConverter x:Key="brushConv">
    <charts:IndexToBrushConverter.Brushes>
        <x:Array Type="Brush">
            <x:Static Member="Brushes.Orange" />
            <x:Static Member="Brushes.Purple" />
            <x:Static Member="Brushes.SkyBlue" />
            <x:Static Member="Brushes.Pink" />
        </x:Array>
    </charts:IndexToBrushConverter.Brushes>
</charts:IndexToBrushConverter>
```

The Fill property on the Rectangle element is bound to the ProductType property of Doodad but converted to one of these brushes:

```
Fill="{Binding ProductType,
    Converter={StaticResource brushConv}}"
```

Perhaps you want a series of repeating colors—not to indicate anything about the item, but just for a little variety. For example, you might use orange for the first item, the fifth, the ninth, and so on.

At first this seems impossible, because the DataTemplate would need access to the index of the particular data item in the collection. But it's easy enough to include this information in the business object. For example, the Doodad class might define a property named ChildIndex, and the DoodadCollection property would be responsible for setting that property for each of its members.

Another option requires deriving the child index from the visual tree and passing that to the IndexToBrushConverter. This job requires a little knowledge about the visual tree that makes up the ItemsControl. An ItemsControl begins with a Border, which contains an ItemsPresenter, which contains the panel you specify in the ItemsPanelTemplate that you set to the ItemsPanel property of the ItemsControl. This panel contains a number of children, equal to the number the items in the data collection; these children are of type ContentPresenter, where the Content property is set to the visual tree you've defined by the DataTemplate set to the ItemTemplate property of the ItemsControl.

The ChartingLib DLL contains a class named ChildIndexProvider that derives from Decorator, defines a dependency property named Index, and installs a handler for the LayoutUpdated event. This handler checks if its parent is a ContentPresenter, and if the parent of that ContentPresenter is a Panel. If so, it then finds the child index of that ContentPresenter within the Panel.

The ChildIndexPresenter is intended to be the root element of a DataTemplate, as demonstrated by the BarChartWithRepeatingColors project:

```
<DataTemplate>
    <charts:ChildIndexProvider Name="indexProv">
        ...
    </charts:ChildIndexProvider>
</DataTemplate>
```

The Fill property of the Rectangle is then bound to the Index property of this element using the IndexToBrushConverter:

```
Fill="{Binding ElementName=indexProv,
               Path=Index,
               Converter={StaticResource brushConv}}"
```

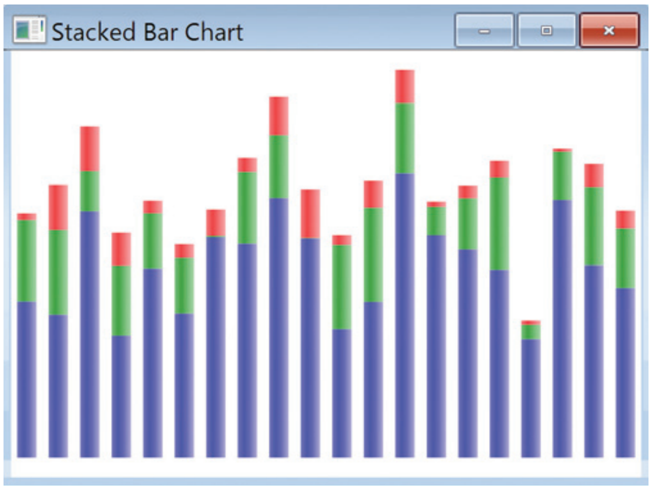


Figure 5 **The StackedBarChartWithGradientBrush Display**

Figure 6 **A DataTemplate with a Path**

```
<DataTemplate>
    <Path VerticalAlignment="Bottom"
          Margin="2"
          Stroke="Black"
          Fill="Silver">
        <Path.Data>
            <PathGeometry
                Figures="M 0 0 L 1 0, 1 0.2, 0.6 0.2, 0.6 0.25,
                        1 0.25, 1 0.6, 0.9 0.6, 0.9 0.3, 0.8 0.3,
                        0.8 0.95, 1 0.95, 1 1, 0.55 1, 0.55 0.5,
                        0.45 0.5, 0.45 1, 0 1, 0 0.95, 0.2 0.95,
                        0.2 0.3, 0.1 0.3, 0.1 0.6, 0 0.6, 0 0.25,
                        0.4 0.25, 0.4 0.2, 0 0.2 Z
                        M 0.2 0.05 L 0.4 0.05, 0.4 0.1, 0.2 0.1 Z
                        M 0.6 0.05 L 0.8 0.05, 0.8 0.1, 0.6 0.1 Z
                        M 0.2 0.15 L 0.8 0.15">
                <PathGeometry.Transform>
                    <ScaleTransform ScaleX="25"
                                    ScaleY="{Binding BaseCost}" />
                </PathGeometry.Transform>
            </PathGeometry>
        </Path.Data>
    </Path>
</DataTemplate>
```

A Fake 3-D Effect

For a really fancy bar chart, you might want to use some 3-D effects for the visuals. It is indeed possible for the DataTemplate to have a root element of type Viewport3D and for each data item to be represented by an entire 3-D scene. But this seems a little extravagant to me, and probably not quite what you want.

What you really want is a single 3-D scene with a single camera and lighting, where each data item is a ModelVisual3D in that scene. But this route would require some major re-architecting of the ItemsControl and its templates. A better solution is to fake the

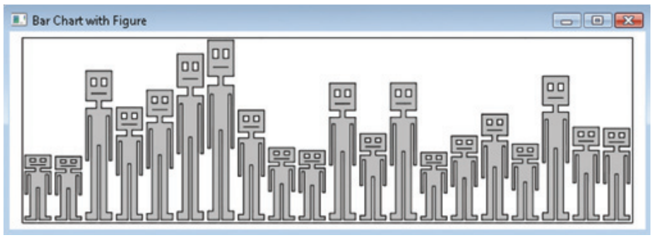


Figure 7 **The BarChartWithFigure Display**

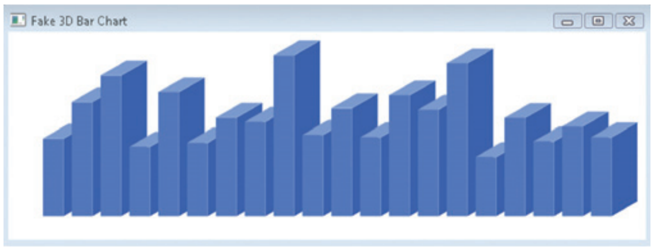


Figure 8 The Fake3DBarChart Display

3-D visuals by using a couple of additional rectangles with skew transforms, as shown in **Figure 8**.

Each of the bars in the chart consists of three rectangles, one in the front, one on the right side, and one on top, so the overall effect looks like a solid block. The overlapping of the items is governed by the order of the blocks in the panel. To create the effect in **Figure 8**, you can't simply cobble together multiple rectangles, because the panel containing these blocks will lay them out based on the *overall* width of each block, and they will not overlap.

There are basically two ways you can persuade an element to appear outside the boundaries recognized by the layout system. The first and most obvious is to apply a RenderTransform. As the name implies, RenderTransform changes the rendering of a visual object without affecting how it's interpreted in layout. But RenderTransform by itself is sometimes awkward. In this particular example, the *untransformed* rectangles for the right side and top must be defined so as not to exceed the width of the front rectangle.

Another approach to persuading the layout system to ignore certain elements is to make them children of a Canvas. Unless a Canvas is given an explicit height and width, it always reports zero dimensions to the layout system. For example, if you put a Rectangle and a Canvas in a single-cell Grid, then any child of that Canvas will effectively be ignored by the layout system.

Figure 9 shows the DataTemplate for the Fake3DBarChart program. A single-cell Grid is given an explicit Width and a Height based on the BaseCost property of the Doodad object. That Grid contains a Rectangle for the front of the block and a Canvas. The Canvas contains two additional single-cell Grids for the right side and top. These Grids have RenderTransforms applied for the skewing effect, but even the untransformed Grids are ignored by the layout system.

Notice how the colors are shaded: The basic color is a resource in the DataTemplate named "brush." The front Rectangle gets that color straight up. The right side is actually two Rectangles within the single-cell Grid. The first one is colored with the "brush" resource; the second one lies on top of the first and is colored with a partially transparent black brush. This causes the overall color to be darkened. For the top, one rectangle colored with "brush" is covered with a partially transparent white brush, making the overall color lighter. This is an excellent technique to create different shades of a particular fixed color or a color obtained through a data binding.

The Scaling Issue

All the examples I've shown so far have been based on the Doodad class, which very conveniently has sales figures in the low two figures and hence is just about ideal for binding directly to the heights of Rectangles (or other elements) in a bar chart.

Values of real data are generally not so compliant, and you really need some way to convert data values to appropriate display dimensions. You'll probably also want to display a little scale at the side of the chart that shows the actual values corresponding to the heights of the bars. Moreover, if the maximum value of some collection of data items is 76,543 (for example), you probably want the scale to go up to a rounded value such as 100,000.

Issues like these often cause programmers to abandon the DataTemplate approach to charting and embrace custom charting controls. But there's no reason to be so hasty. There are a couple of solutions to scaling bars in a chart. If you know beforehand the approximate range of values you'll be charting, you can apply a ScaleTransform in the XAML. For example, if the values you need to graph range from 0 to 10, you might apply a ScaleTransform with a ScaleY property set to 15 to create bars up to 150 units tall.

Another approach (which I'll be demonstrating here) requires additional properties to be added to the business objects. The scaling calculation is performed in code, but the scaled values can be accessed in XAML.

The Gizmo class in ChartingLib has a Name property and a Revenues property, but it also contains properties named Scaling-

Figure 9 The Fake3DBarChart Data Template

```
<DataTemplate>
    <DataTemplate.Resources>
        <SolidColorBrush x:Key="brush" Color="#4080FF" />
    </DataTemplate.Resources>
    <Grid VerticalAlignment="Bottom"
          Margin="4"
          Width="20"
          Height="{Binding BaseCost}">

        <!-- Front -->
        <Rectangle Fill="{StaticResource brush}" />

        <Canvas>
            <!-- Right Side -->
            <Grid Canvas.Left="20"
                  Height="{Binding BaseCost}"
                  Width="25"
                  VerticalAlignment="Bottom">
                <Grid.RenderTransform>
                    <SkewTransform AngleY="-30" />
                </Grid.RenderTransform>

                <Rectangle Fill="{StaticResource brush}" />
                <Rectangle Fill="#40000000" />
            </Grid>

            <!-- Top: 14.4 = 25 * tan(30) -->
            <Grid Canvas.Top="-14.4"
                  Height="14.4"
                  Width="20">
                <Grid.RenderTransform>
                    <SkewTransform CenterY="14.4" AngleX="-60" />
                </Grid.RenderTransform>

                <Rectangle Fill="{StaticResource brush}" />
                <Rectangle Fill="#40FFFFFF" />
            </Grid>
        </Canvas>
    </Grid>
</DataTemplate>
```

Factor and ScaledRevenues. Whenever Revenues or ScalingFactor changes, the class simply multiplies the two values and sets the result to ScaledRevenues. (ScaledRevenues is really a height in device-independent units, but that's a detail that Gizmo really needn't bother itself with.)

Somebody has to keep track of the maximum value of the Revenues properties of all the Gizmo objects. Probably the best spot for this logic is the GizmoCollection class that derives from ObservableCollection<Gizmo>. Because Gizmo itself implements the INotifyPropertyChanged interface, GizmoCollection can install PropertyChanged event handlers on all the Gizmo items in the collection. If the Revenues property on any Gizmo object changes, GizmoCollection calculates a new maximum that it stores in a property named MaximumRevenues. GizmoCollection also defines a MaximumRevenuesChanged event that it fires whenever MaximumRevenues changes.

The remainder of the logic is handled in the GizmoPresenter class. The GizmoPresenter class defines a property named Gizmos of type ObservableCollection<Gizmo>, creates that collection, and installs a handler for the MaximumRevenuesChanged event.

GizmoPresenter defines a property named DisplayHeight, which will be set in XAML to the maximum desired height of the bars in the bar chart. It also defines two read-only properties, named RevenuesBase and HalfRevenuesBase, intended to be of assistance in creating a visual scale for the bar chart.

When the MaximumRevenuesChanged event is fired, the GizmoPresenter gets the new MaximumRevenues value from the GizmoCollection object and calculates a RevenuesBase value.

RevenuesBase is basically MaximumRevenues rounded up to some value with a lot of zeroes. It's the Revenues value that corresponds to DisplayHeight. But you probably don't want to restrict RevenuesBase to a power of 10 (for example, 10, 100, 1000, and so forth) because if MaximumRevenues is 1,001, RevenuesBase would be 10,000, and the tallest bars would only be about one-tenth the height of the chart. It's preferable to use a geometric sequence, such as a 1-2-5 series, so that Revenues base will equal a value from the series 1, 2, 5, 10, 20, 50, 100, 200, 500, and so forth.

The ChartingLibHelper class has a static method named GetRoundedMaximum to help out with this calculation. (It's fairly simple, but logarithms are involved.) You simply pass in a calculated maximum and an ordered series of values between 1 and 10, for example, 2 and 5.

GizmoPresenter uses this RevenuesBase value and the DisplayHeight to set the ScalingFactor value in each of the Gizmo objects.

The ScaledBarChart project demonstrates this technique. The height of each bar is bound not to the Revenues property of Gizmo but to the ScaledRevenues property. In addition, enough information is provided from GizmoPresenter to construct a simple scale at the left entirely in XAML. The result is shown in **Figure 10**.

If you run this program and wait for the random-number generator to make one of the Revenues values greater than 100,000, you'll see the scale shift to showing a maximum of 200,000, and all the bars decrease in height by half.

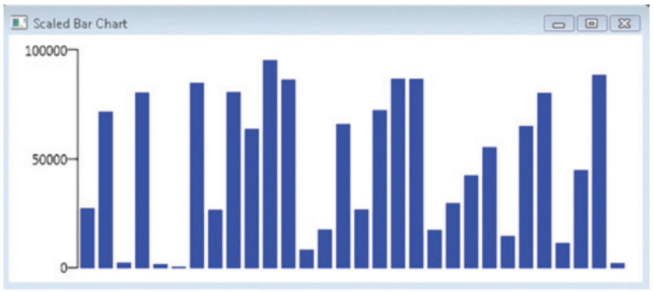


Figure 10 **The ScaledBarChart Display**

Tackling the Pie Chart

At first a pie chart seems quite unsuited to realization by an ItemsControl, but it's really not too bad. You don't even need a custom panel, but you will need a custom element to display the slices of the pie.

The PieSlice class in ChartingLib derives from Shape using techniques I discussed in my column "Vector Graphics and the WPF Shape Class" in the March 2008 issue of *MSDN Magazine* (msdn.microsoft.com/en-us/magazine/cc337899.aspx). It has a Center property of type Point, a Radius property of type double, and StartAngle and SweepAngle properties, also of type double. The StartAngle is relative to a vertical line extending up from the center, with positive angles going clockwise.

PieSlice also overrides the MeasureOverride method to report a size as if the slice encompassed the entire circle and not just one part of it. This allows all the slices to be put into a single-cell Grid.

The pie chart I'll be showing uses a DataTemplate to display seven objects of type Product. These are collected in a class named ProductLineup that derives from ObservableCollection<Product>. The Product class has Name and Sales properties, as well as two properties that help out with the display: a Percentage property, which is a percent of the total sales, and an AccumulatedPercentage, which is the accumulated percentage of sales of items in the collection preceding this item. These properties are set by the collection class. Obviously, AccumulatedPercentage is intended to be

Figure 11 **DataTemplate for a Pie Chart**

```
<DataTemplate>
    <charts:ChildIndexProvider Name="indexProvider">
        <Grid>
            <charts:PieSlice
                Name="pieslice"
                StartAngle="{Binding Accumulated,
                                 Converter={StaticResource angleConv}}"
                SweepAngle="{Binding Percentage,
                                 Converter={StaticResource angleConv}}"
                Fill="{Binding ElementName=indexProvider,
                                 Path=Index,
                                 Converter={StaticResource brushConv}}"
                Stroke="Black"
                StrokeThickness="1"
                Center="200 200"
                Radius="200">
            </charts:PieSlice>
            ...
        </Grid>
    </charts:ChildIndexProvider>
</DataTemplate>
```

bound to the StartAngle of the pie slice, and Percentage is bound to the Sweep angle, as shown in the portion of the DataTemplate shown in **Figure 11**.

The "angleConv" resource key references the PercentageToAngleConverter that simply multiplies values by 360. The ItemsPanelTemplate is a single-cell Grid.

There are no ToolTips defined here. Instead, I wanted actual labels. For this feature, I added a read-only property in PieSlice named CenterAngle. Its value is calculated as the StartAngle plus one-half the SweepAngle. It's fairly easy to add a Line element and a TextBlock to the template and rotate them by CenterAngle degrees, but you'll get some upside-down text, and that's not good. You really want the text to be consistently horizontal.

I decided another piece of code was in order, a Decorator derivative called CircleShifter that is intended to arrange a child element around a circle. CircleShifter has a property named BasePoint of type Point and another named RotateTransform of type RotateTransform. CircleShifter positions its child at BasePoint rotated by RotateTransform—but differently if the Angle property of the RotateTransform is between 0 and 180 degrees (the right of the pie) or between 180 and 360 degrees (the left of the pie). **Figure 12** shows the additional XAML and **Figure 13** shows the result.

If the slices get too small, there's no logic to prevent the labels from running into each other. However, CircleShifter might be enhanced so that labels toward the top and bottom of the pie chart are more centered.

It's tempting to try to derive some transforms to make an "exploded" pie chart, where one or more slices are pulled away from the center. This feature is probably best handled within the PieSlice class itself. An additional property would indicate an offset from the center (for purposes of the MeasureOverride method), and a Boolean property would govern whether that offset should actually be applied to the rendered slice.

A 3-D pie chart can be desirable, but it presents some challenges. Visually, the best results would be obtained from putting all the pie slices in a single Viewport3D, but as I mentioned earlier, that doesn't allow you to use an ItemsControl with a DataTemplate. The DataTemplate itself can be a Viewport3D containing a GeometryModel3D defining a 3-D pie slice, but these multiple Viewport3D elements will be stacked from background to foreground in the same order as their child indices in the Grid. The only foolproof way to avoid odd overlapping effects is to explode the slices sufficiently from the center so that they don't overlap at all.

The fake 3-D effect I used for the bar chart was simplified by the uniform shape of the sides and tops and the uniform left-to-right overlapping. A fake 3-D effect might be implemented in the PieSlice class, but each slice would need to look a little different depending on its position, and the overlapping would still be a problem.

Figure 12 **DataTemplate for Pie Chart Labels**

```
DataTemplate>
    <charts:ChildIndexProvider Name="indexProvider">
        <Grid>
            ...
            <Canvas>
                <Line X1="200" Y1="50" X2="200" Y2="-25" Stroke="Black">
                    <Line.RenderTransform>
                        <RotateTransform
                            Angle="{Binding ElementName=pieslice,
                                            Path=CenterAngle}"
                            CenterX="200" CenterY="200"/>
                    </Line.RenderTransform>
                </Line>

                <charts:CircleShifter BasePoint="200 -25">
                    <charts:CircleShifter.RotateTransform>
                        <RotateTransform
                            Angle="{Binding ElementName=pieslice,
                                            Path=CenterAngle}"
                            CenterX="200" CenterY="200"/>
                    </charts:CircleShifter.RotateTransform>

                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Name}" />
                        <TextBlock Text="{Binding Sales,
                                   StringFormat=': {0:C}'}" />
                    </StackPanel>
                </charts:CircleShifter>
            </Canvas>
        </Grid>
    </charts:ChildIndexProvider>
</DataTemplate>
```

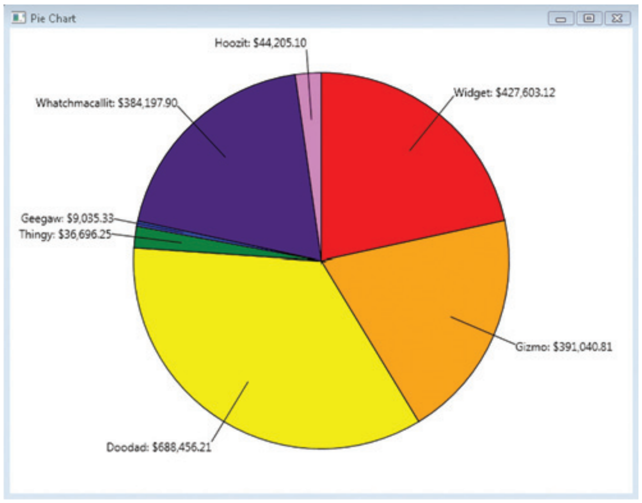


Figure 13 **The PieChart with Labels Display**

Connecting the Points

When implementing a line chart, it really becomes questionable whether an ItemsControl and DataTemplate are the way to go. The big problem is that a line chart needs to deal with two variables, and the data must be scaled both horizontally and vertically. Also, it's often desirable to connect the points with a line or smoothed curve, which involves something external to each DataTemplate getting information from all the DataTemplates. Still, the advantage of defining the visual elements of the line chart entirely in XAML makes the effort worthwhile.

Next, I'll explore the code support necessary to make a line-charting DataTemplate a reality. ■

CHARLES PETZOLD *is a longtime contributing editor for* MSDN Magazine. *His most recent book is "The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine" (Wiley, 2008). His Web site is charespetzold.com.*

Drawing with Direct2D

In the June issue of *MSDN Magazine* (msdn.microsoft.com/en-us/magazine/dd861344), I introduced Direct2D, a brand new 2-D graphics API designed to support the most demanding and visually rich desktop applications with the best possible performance. In that article, I described where Direct2D fits in among the various graphics APIs on Windows, its architecture and principles. In particular, I described in detail the fundamentals of using Direct2D reliably and efficiently for rendering inside of a window. This included creation of device-specific resources as well as device-independent resources and their respective lifecycles. If you haven't already done so, I'd encourage you to read that article before continuing on here, as this article very much builds upon the foundation laid out there.

Rendering and Control

It is helpful to think about Direct2D as a hardware-accelerated 2-D rendering API. Of course, it supports software fallback, but the point here is that Direct2D is about rendering. Unlike other graphics APIs on Windows, Direct2D takes a componentized approach to graphics. It does not provide its own APIs for encoding and decoding bitmaps, text layout, font management, animation, 3-D and so on. Rather, it focuses on rendering and control over the graphics processing unit (GPU) while providing first class hooks to other APIs that focus on things like text layout and imaging. Direct2D does, however, provide primitives for representing different types of brushes as well as simple and complex shapes, the building blocks for any 2-D graphics application.

In this article, I'm going to show you how to draw with Direct2D. I'll begin by introducing Direct2D's color structure and then show you how to create various types of brushes. Unlike most of the other graphics APIs on Windows, Direct2D doesn't provide a "pen" primitive, so brushes are pretty important as they're used for all outline and filling tasks. With that out of the way, I'll show you how to draw primitive shapes.

Colors

Direct2D uses a simple structure that represents colors with floating-point color components. The D2D1_COLOR_F type is actually a typedef for the D3DCOLORVALUE structure used by Direct3D to describe color values. It includes individual floating-point values for the red, green, blue and alpha channels. Values range from 0.0 to 1.0 with 0.0 being black for the color channels and completely transparent for the alpha channel. Here's what it looks like:

```
struct D3DCOLORVALUE
{
    FLOAT r;
    FLOAT g;
    FLOAT b;
    FLOAT a;
};
```

Direct2D provides the ColorF helper class in the D2D1 namespace that inherits from D2D1_COLOR_F and defines some common color constants, but more importantly provides a few helpful constructors that initialize the D2D1_COLOR_F structure. You can, for example, define red as follows:

```
const D2D1_COLOR_F red = D2D1::ColorF(1.0f, 0.0f, 0.0f);
```

Another constructor accepts a packed RGB value and converts it to the individual color channels. Here's red again:

```
D2D1::ColorF(0xFF0000)
```

This is exactly the same as using the following enum value:

```
D2D1::ColorF(D2D1::ColorF::Red)
```

Although concise, using the packed RGB representation does eat up a few more CPU cycles as the different color channels need to be extracted and converted to their floating point equivalents, so use it with care.

All of the constructors accept an optional alpha value that defaults to 1.0, or fully opaque. Thus, you can express semi-transparent blue as follows:

```
D2D1::ColorF(0.0f, 0.0f, 1.0f, 0.5f)
```

Apart from clearing a render target's drawing area, however, there is little you can do with a color directly. What you need are brushes.

Brushes

Unlike simple color structures, brushes are resources exposed through interfaces. They are used to draw lines, to draw and fill shapes, and to draw text. Interfaces that derive from ID2D1Brush represent the different types of brushes provided by Direct2D. The ID2D1Brush interface itself allows you to control opacity of a brush as a whole, rather than changing the alpha channel of the colors used by the brush. This can be particularly helpful with some of the more interesting types of brushes.

Send your questions and comments to mmwincpp@microsoft.com.

Here's how you might set the opacity of the brush to 50 percent:

```
CComPtr<ID2D1Brush> brush;
// Create brush here...

brush->SetOpacity(0.5f);
```

ID2D1Brush also allows you to control the transform applied to the brush since, unlike Windows Presentation Foundation (WPF), brushes adopt the coordinate system of the render target rather than that of any particular shape they may be drawing.

The various render target methods for creating brushes all accept an optional D2D1_BRUSH_PROPERTIES structure that can be used to set the initial opacity and transform.

All of the brushes provided by Direct2D are mutable and can efficiently change their characteristics, so you don't need to create new brushes with different characteristics. Keep this in mind as you design your applications. Brushes are also device-dependent resources, so they are bound by the lifetime of the render target that created them. In other words, you can reuse a particular brush for as long as its render target is valid but you must release it when the render target is released.

As its name suggests, the ID2D1SolidColorBrush interface represents a solid color brush. It adds methods for controlling the color used by the brush. A solid color brush is created with a render target's CreateSolidColorBrush method:

```
CComPtr<ID2D1RenderTarget> m_target;
// Create render target here...

const D2D1_COLOR_F color = D2D1::ColorF(D2D1::ColorF::Red);
CComPtr<ID2D1SolidColorBrush> m_brush;

HR(m_target->CreateSolidColorBrush(color, &m_brush));

This brush's initial color can also easily be changed using the SetColor
method:

m_brush->SetColor(differentColor);
```

I'm going to leave a complete discussion of shapes for the next section, but for the sake of having something to see, I'll just fill a window's render target using a rectangle based on the size of the render target. Keep in mind that Direct2D uses device-independent pixels (DIPs). As a result, the size reported by a particular device, such as a desktop window's client area, may not match the size of the render target. Fortunately, it's very easy to get the size of the render target in DIPs using the GetSize method. GetSize returns a D2D1_SIZE_F structure that Direct2D uses to represent sizes with two floating point values named width and height. I can then provide a D2D1_RECT_F variable to describe the area to fill by using the RectF helper function and plugging in the size reported by the render target. Finally, I can use the render target's FillRectangle method to do the actual drawing.

Here's what the code looks like:

```
const D2D1_SIZE_F size = m_target->GetSize();
const D2D1_RECT_F rect = D2D1::RectF(0, 0, size.width, size.height);

m_target->FillRectangle(rect, m_brush);
```

Figure 1 shows what the window looks like with a solid green brush. Very exciting indeed!

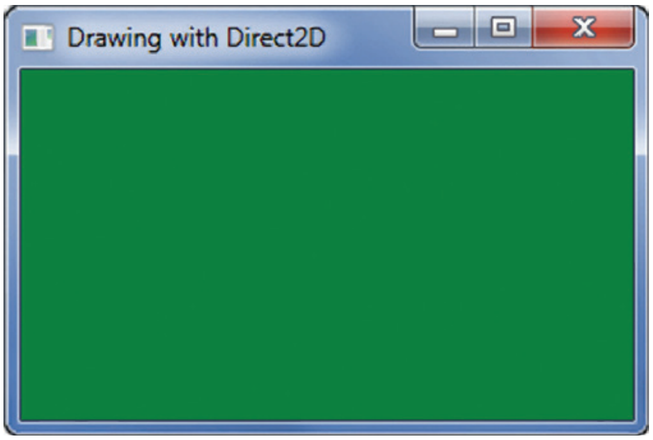


Figure 1 **Solid Color Brush**

You can reuse a particular brush for as long as its render target is valid but you must release it when the render target is released.

Direct2D also provides two types of gradient brushes. A gradient brush is one that fills an area with colors blended along an axis. A linear gradient brush defines the axis as a straight line with a start and end point. A radial gradient brush defines the axis as an ellipse, where the colors radiate outward from some point relative to the center of the ellipse.

A gradient is defined as a series of relative positions from 0.0 to 1.0. Each has its own color and is called a gradient stop. It is possible to use positions outside of this range to produce various effects. To create a gradient brush, you must first create a gradient stop collection. Start by defining an array of D2D1_GRADIENT_STOP structures.

Here's an example:

```
const D2D1_GRADIENT_STOP gradientStops[] =
{
    { 0.0f, color1 },
    { 0.2f, color2 },
    { 0.3f, color3 },
    { 1.0f, color4 }
};
```

Next, call the render target's CreateGradientStopCollection method to create a collection object based on the array of gradient stops, as follows:

```
CComPtr<ID2D1GradientStopCollection> gradientStopsCollection;

HR(m_target->CreateGradientStopCollection(gradientStops,
                                          _countof(gradientStops),
                                          &gradientStopsCollection));
```

The first parameter is a pointer to the array and the second parameter provides the size of the array. Here I'm just using the standard _countof macro. The CreateGradientStopCollection method

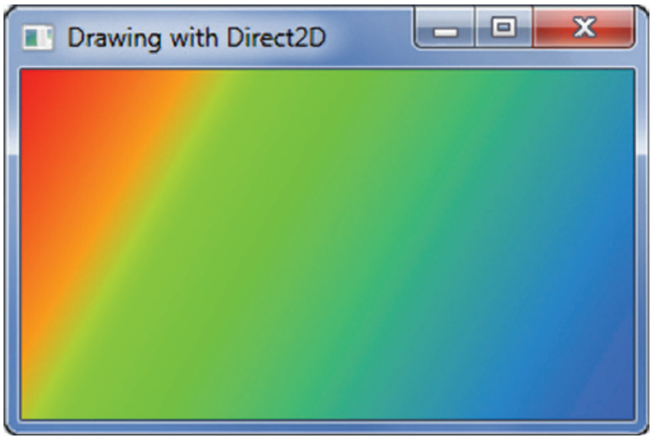


Figure 2 **Linear Gradient Brush**

returns the new collection. To create a linear gradient brush, you also need to provide a D2D1_LINEAR_GRADIENT_BRUSH_PROPERTIES structure to indicate the start and end point of the gradient's axis. Unlike the gradient stop positions, these points are in the brush's coordinate space which is usually that of the render target unless a transform is set on the brush.

For example, you could create the brush with an axis that runs from the top-left corner of the render target to the bottom-right corner as follows:

```
const D2D1_SIZE_F size = m_target->GetSize();

const D2D1_POINT_2F start = D2D1::Point2F(0.0f, 0.0f);
const D2D1_POINT_2F end = D2D1::Point2F(size.width, size.height);
const D2D1_LINEAR_GRADIENT_BRUSH_PROPERTIES properties =
  D2D1::LinearGradientBrushProperties(start, end);

HR(m_target->CreateLinearGradientBrush(properties,
                                       gradientStopsCollection,
                                       &m_brush));
```

LinearGradientBrushProperties is another helper function provided by Direct2D to initialize a D2D1_LINEAR_GRADIENT_BRUSH_PROPERTIES structure. The CreateLinearGradientBrush method accepts this along with the gradient stop collection shown earlier and returns an ID2D1LinearGradientBrush interface pointer representing the new brush.

Of course, the brush won't know if the size of the render target changes. If you wanted the end point of the axis to change as a window is resized, you could easily do so with the brush's SetEndPoint method. Similarly, you can change the axis' start point with the SetStartPoint method.

Here's what the drawing code looks like:

```
const D2D1_SIZE_F size = m_target->GetSize();
const D2D1_RECT_F rect = D2D1::RectF(0, 0, size.width, size.height);

m_brush->SetEndPoint(D2D1::Point2F(size.width, size.height));
m_target->FillRectangle(rect, m_brush);
```

Figure 2 shows what the window looks like with the linear gradient brush.

To create a radial gradient brush, you need to provide a D2D1_RADIAL_GRADIENT_BRUSH_PROPERTIES structure. This structure defines the ellipse as well as an offset relative to the center of the ellipse that represents the origin out of which the brush "radiates" color based on the gradient stop collection. The following example produces an ellipse that is centered in the render target, has an X and Y radius to match that of the render target, and an origin half way toward the bottom-right corner:

```
const D2D1_SIZE_F size = m_target->GetSize();

const D2D1_POINT_2F center = D2D1::Point2F(
  size.width / 2.0f, size.height / 2.0f);
const D2D1_POINT_2F offset = D2D1::Point2F(
  size.width * 0.25f, size.height * 0.25f);
const float radiusX = size.width / 2.0f;
const float radiusY = size.height / 2.0f;

const D2D1_RADIAL_GRADIENT_BRUSH_PROPERTIES properties =
  D2D1::RadialGradientBrushProperties(center,
                                      offset,
                                      radiusX,
                                      radiusY);

HR(m_target->CreateRadialGradientBrush(properties,
                                       gradientStopsCollection,
                                       &m_brush));
```

RadialGradientBrushProperties is another helper function provided by Direct2D to initialize a D2D1_RADIAL_GRADIENT_BRUSH_PROPERTIES structure. The CreateRadialGradientBrush method accepts this along with the gradient stop collection shown earlier and returns an ID2D1RadialGradientBrush interface pointer representing the new brush.

You can also change the brush's ellipse and origin at any time, as follows:

```
m_brush->SetCenter(center);
m_brush->SetGradientOriginOffset(offset);
m_brush->SetRadiusX(radiusX);
m_brush->SetRadiusY(radiusY);
```

Figure 3 shows the radial gradient brush in action.

Direct2D also provides a bitmap brush, but I'll leave a discussion of Direct2D bitmaps for a future article.

Shapes

Thus far I've shown you only how to fill a rectangle so that I could focus on brushes, but Direct2D can do so much more than plain rectangles.

For starters, primitives are provided for rectangles, rounded rectangles and ellipses. By primitives, I just mean that there is a plain old data structure for each of these.

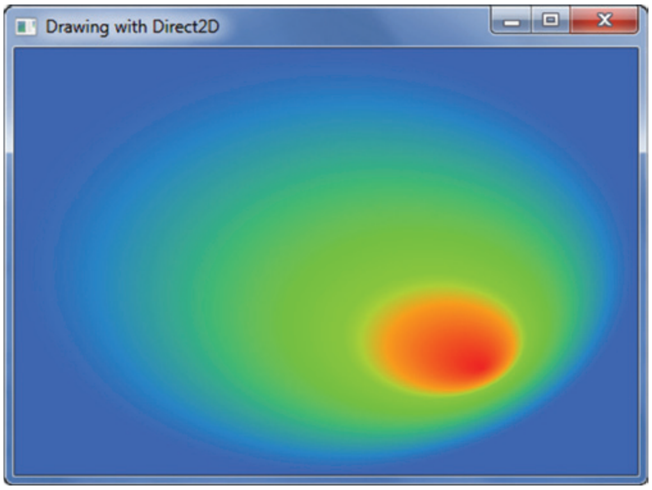


Figure 3 **Radial Gradient Brush**

D2D1_RECT_F represents a floating-point rectangle and is itself a typedef for D2D_RECT_F:

```
struct D2D_RECT_F
{
    FLOAT left;
    FLOAT top;
    FLOAT right;
    FLOAT bottom;
};
```

D2D1_ROUNDED_RECT represents a rounded rectangle and is defined as follows, with the radiuses defining the quarter ellipses that will be used to draw the corners:

```
struct D2D1_ROUNDED_RECT
{
    D2D1_RECT_F rect;
    FLOAT radiusX;
    FLOAT radiusY;
};
```

D2D1_ELLIPSE represents an ellipse and is defined with a center point as well as radiuses:

```
struct D2D1_ELLIPSE
{
    D2D1_POINT_2F point;
    FLOAT radiusX;
    FLOAT radiusY;
};
```

Although these structures may not seem too exciting, there are two reasons I mention them up front. First, they are used to create more sophisticated geometry objects. Second, if all you need is to fill or draw one of these primitives, you should stick with them as you will typically get better performance if you avoid the geometry objects.

Render targets provide a set of Fill- and Draw- methods for filling and outlining these primitives. So far I've shown you how to use the FillRectangle method. I won't bore you with examples of the FillRoundedRectangle and FillEllipse methods, as they work in exactly the same way. In contrast to the Fill- methods, the Draw- methods can be used to outline a particular shape and are quite versatile. Like the Fill- methods, the Draw- methods covering the three primitives all work in exactly the same way, so I'll cover only the DrawRectangle method.

In its simplest form, you can draw the outline of a rectangle as follows:

```
m_target->DrawRectangle(rect,
                        m_brush,
                        20.0f);
```

The third parameter specifies the width of the stroke that is drawn to outline the rectangle. The stroke itself is centered on the rectangle, so if you had filled the same rectangle you would see that the fill and the stroke overlap by 10.0 DIPs. An optional fourth parameter may be provided to control the style of the stroke that is drawn. This is useful if you need to draw a dashed line or just want to control the type of join at a shape's vertices.

Stroke style information is represented by the ID2D1StrokeStyle interface. Stroke style objects are device-independent resources, which mean they don't have to be re-created whenever the render target is invalidated. In that event, a new stroke style object is created using the Direct2D factory object's CreateStrokeStyle method.

If you just want the stroke to use a particular type of join, you can create a stroke style as follows:

```
D2D1_STROKE_STYLE_PROPERTIES properties = D2D1::StrokeStyleProperties();
properties.lineJoin = D2D1_LINE_JOIN_BEVEL;

HR(m_factory->CreateStrokeStyle(properties,
                                0, // dashes
                                0, // dash count
                                &m_strokeStyle));
```

The D2D1_STROKE_STYLE_PROPERTIES structure provides a variety of members to control various aspects of the stroke style, in particular the shape, or cap, at each end of an outline or dash as well as the dash style itself. The second and third parameter to the CreateStrokeStyle method are optional, and you need to provide them only if you are defining a custom dashed stroke style. To define a dashed stroke style, make sure you specify the dashes in pairs. The first element in each pair is the length of the dash and the second is the length of the space before the next dash. The values themselves are multiplied by the stroke width. You can specify as many pairs as you need to produce the desired pattern. Here's an example:

```
D2D1_STROKE_STYLE_PROPERTIES properties = D2D1::StrokeStyleProperties();
properties.dashStyle = D2D1_DASH_STYLE_CUSTOM;

float dashes[] =
{
    2.0f, 1.0f,
    3.0f, 1.0f,
};

HR(m_factory->CreateStrokeStyle(properties,
                                dashes,
                                _countof(dashes),
                                &m_strokeStyle));
```

You can choose from any of a number of different dash styles from the D2D1_DASH_STYLE enumeration instead of defining your own. **Figure 4** shows some different stroke styles at work. If you look closely, you'll see that Direct2D automatically alpha blends with per-primitive antialiasing for the best looking results.

There's so much more that Direct2D can do for you, from complex geometries and transformations, to drawing text and bitmaps, and much more. I hope to cover these and more in future columns. ■

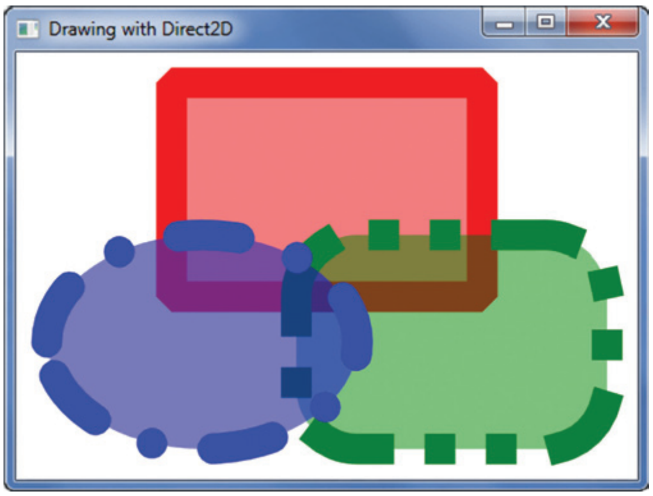


Figure 4 **Drawing Primitives**

KENNY KERR *is a software craftsman specializing in software development for Windows. He has a passion for writing and teaching developers about programming and software design. Reach him at weblogs.asp.net/kennykerr.*

An Introduction to IPsec VPNs on Mobile Phones

This afternoon, while I was away from the office, I got an e-mail on my phone. The message had a link to a document I was supposed to read—a document on a SharePoint site available only through my company's intranet. This was a bummer because I had to wait until I could fire up my laptop, put my smart card in the smart card reader, get a Wi-Fi connection at a coffee shop, connect my laptop to the corporate VPN, and log on before I could read the document.

Life would be much easier if I could just use the phone to access the SharePoint site. Of course, my phone would need some magic way of connecting securely to the corporate network and authenticating me. In other words, like my laptop, my phone would need the capability to start a VPN connection to the network. Many commercial phone models, including Windows phones, come with a VPN client built in. But there are shortcomings in the most widely available VPN clients because they are based on version 1 of a specification called Internet Key Exchange (IKEv1). IKEv1 is a stable part of the IPsec framework and is great for wired devices or devices, like laptops, with relatively large batteries that don't move around too much. However, it is not ideal for mobile phones.

Sure, I use a wireless connection on my laptop to go from my office to a conference room and back again. I might switch from a wireless connection to a wired one and expect no loss of connectivity. But we don't move around nearly as much with a laptop as we do with a phone. A phone travels with you through traffic and in and out of buildings, transitioning into and out of the roaming state. Unless you use a cellular modem, when was the last time your laptop told you it was roaming? Generally, a phone changes its point of network attachment with a frequency and complexity that a laptop never has to worry about. The guys who thought about the IKEv1 spec didn't have to worry about mobile phone scenarios either, because in the late 1990s, when RFC 2409 was being written, smart phones just weren't prevalent in the marketplace. Since then, the use of mobile phones has skyrocketed, and the importance of the mobile phone has begun to approach the importance of other, larger computing devices, like the workstation or the laptop.

IKEv1 isn't well suited to a highly mobile style of computing because IKEv1 doesn't have a good way to cope with a host that might change its point of network attachment several times in a few seconds. When IKEv2 was drafted, a set of extensions called the Mobility and Multihoming (MOBIKE) protocol was also drafted to accommodate the mobile phone scenario. Using these extensions, a mobile phone VPN becomes far more practical. Several products on the market support IKEv2 and MOBIKE, including Microsoft Systems Center Mobile Device Manager (SCMDM).

In this article, I'll cover some of the basics of the technology behind IKEv2 and MOBIKE. I assume you have a working knowledge of IPv4 and networking, some familiarity with mobile phones, and a basic understanding of cryptography. This article is not going to cover IPsec in detail and is not going to discuss other sorts of VPN technologies, such as SSL-based VPNs. I also won't discuss IPv6. IPsec and IKE are extensions to IPv4, but they are baked into IPv6, so some of the things we'll touch on here will still be applicable on an IPv6 network. However, IPv6 introduces enough complexity and new terminology that it wouldn't be possible to cover it in sufficient detail.

Say 'AH'

Three protocols make up the core of IPsec: Authentication Header (AH), Encapsulating Security Payload (ESP), and IKE. To address IKE, I first need to discuss AH and ESP.

In a nutshell, AH makes sure that the packets we send aren't tampered with. It protects the integrity of our packets. AH also ensures that the packets are sent from whomever claims to have sent them. It ensures the authenticity of our packets. AH does not, however, provide any measure of privacy or encryption. An attacker who gains access to the network could still sniff packets that are guarded by AH and extract their content. He would not, however, be able to masquerade as one of the authenticated parties nor be able to change our packets in transit. AH is defined in RFC 4302.

For instance, Alice and Bob have established a VPN connection between themselves and have chosen to protect their packets with AH. Charlie inserts himself onto the network and starts intercepting packets. Charlie is able to reconstruct Alice and Bob's conversation because he can see the content of their packets and discern what kind of traffic is passing between them. However, he can't forge a message from Alice to Bob without getting caught, and he can't alter Bob's messages to Alice, either. Both of these things are prevented by AH.

Send your questions and comments to goplaces@microsoft.com.

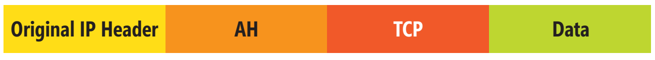


Figure 1. **A packet with an AH header.**

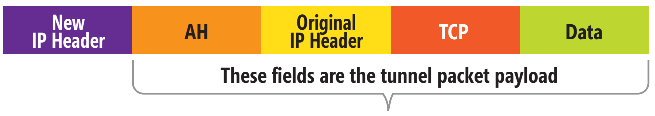


Figure 2. **A packet with AH in tunnel mode.**

AH can be used in transport mode or tunnel mode. In transport mode, the payload of a packet is protected, and packets are routed directly from one host to another. In tunnel mode, the entire packet is protected by AH, and packets are routed from one end of an IPsec "tunnel" to another. Tunneling is accomplished through encapsulation of the original packet. At either end of the tunnel is a security gateway. The gateway that is sending a packet is responsible for encapsulating that packet by adding an "outer" IP header and address. This outer address routes the packet to the security gateway at the other end of the tunnel. The gateway that receives the packet is responsible for processing the AH to determine that the packet is from a valid sender and hasn't been tampered with, and then it routes the packet to its final destination.

In transport mode, an AH gets added to the packet right after the IP header. The AH comes before the next layer protocol in the packet (such as UDP or TCP) and also before any other IPsec headers in the packet, such as the ESP header. The IP header that precedes the AH must have the value 51, which is the magic number assigned by the IANA to AH that tells the application processing the packet that the next thing it will see is an AH. **Figure 1** shows the shape of a packet after having an AH added to it.

In tunnel mode, the AH gets added after the new IP header. The encapsulated IP header is treated as part of the payload, along with everything else. This is shown in **Figure 2**. The first 8 bits of a packet protected by AH specify the protocol ID of the payload that comes after the AH. This tells the receiver what to expect after the AH payload. For instance, if the next layer protocol is TCP, the protocol ID will be set to 6. This field is called the Next Header. (You might wonder why it's not called Protocol in a manner consistent with other headers in IPv4. The reason is consistency and compatibility with IPv6. Fortunately, you don't really need to know much about IPv6 to understand how AH works on your IPv4 network, but if you were wondering why it's called Next Header, that's why.)

Next comes the 7-bit length of the AH payload. Since there's only 7 bits, the payload's size is limited to 128 bytes. Then comes a long string of zeroes—2 bytes of them, in fact. These 2 bytes are reserved for future use according to the RFC, so until that future use is defined, we've got 2 empty bytes that are just along for the ride.

Following these 2 bytes is the Security Parameter Index (SPI). This is a 32-bit number that is used to determine which IPsec Security Association (SA) the AH is associated with. I'll talk in detail about SAs when I discuss IKEv2. This number is followed by a 32-bit sequence number, which is incremented with each packet sent and is used to prevent replay attacks.

After the sequence number comes the Integrity Check Value (ICV). The ICV is calculated at the sender by applying a hashing function, such as SHA-2, to the IP header, the AH, and the payload. The receiver checks that the packet hasn't been tampered with by applying the same hashing function and confirming that the same hash is produced.

I Have ESP

Like AH, Encapsulating Security Payload (ESP) can provide integrity and authentication. Unlike AH, however, ESP provides authentication and integrity only for the payload of the packet, not for the packet header. ESP can also be used to provide confidentiality by encrypting the packet payload. In theory, these features of ESP can be enabled independently, so it is possible to have encryption without authentication and integrity, or to have integrity and authentication without encryption. In practice, however, doing one without the other doesn't make a great deal of sense. For instance, knowing that a message has been sent to me confidentially doesn't do me any good if I can't also be completely sure who sent it. ESP is defined in RFC 4303.

ESP, like AH, can be enabled in tunnel mode or transport mode. The ESP header must be inserted after the AH. In transport mode, ESP encrypts the payload of the IP packet. In tunnel mode, ESP treats the entire encapsulated packet as the payload and encrypts it. This is illustrated in **Figure 3**.

The encryption algorithm is chosen through a process of negotiation between the peers that set up the IPsec SA. Advanced Encryption Standard (AES) is a common choice of encryption al-

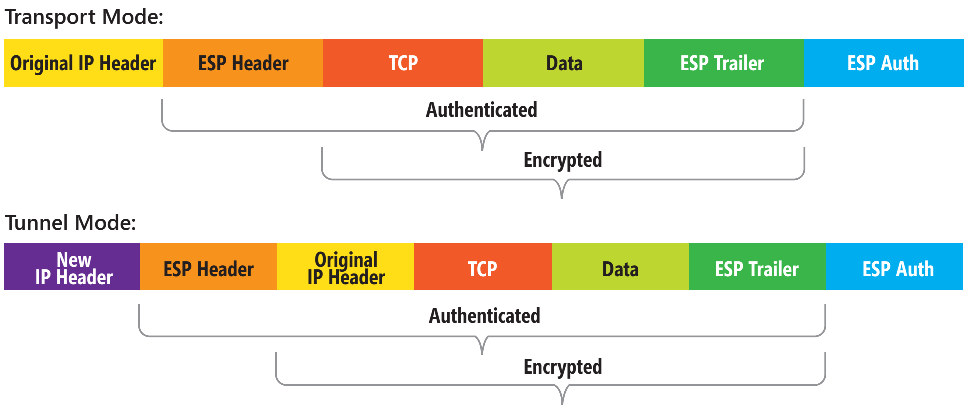


Figure 3. **ESP header in transport mode and tunnel mode.**

gorithm for modern implementations. Of course, to use AES, you must first have a shared secret for the two hosts that are about to start a secure communication. Manually giving the shared key to the host is an impractical approach because it doesn't scale, but you can use Diffie-Hellman (DH) key exchange to provision the secret.

Diffie-Hellman Groups

DH is a protocol that allows two parties to share a secret over an insecure channel, and it is integral to the negotiation that takes place in IKE. The shared secret that is communicated via DH can be used to create a communication channel that is securely encrypted. The math that goes into DH is complex, and I will not go into it in detail here. If you're interested in learning more about it, check the resources that cover modular exponential (MODP) groups and their application to DH. For our purposes, it suffices to say that a DH group is a specific collection of numbers with a mathematical relationship and a unique group ID.

A DH group is specified when a secure connection is being set up with IPsec, during the IKE negotiation. In this negotiation, the two peers trying to establish a secure connection need to find a DH group that they both support. DH groups with higher IDs have higher cryptographic strength. For instance, the first DH groups that are called out in the original IKE specification had about the same cryptographic strength as a symmetric key, with between 70 and 80 bits. With the advent of stronger encryption algorithms such as AES, more strength was required from DH groups to prevent the DH groups from becoming a weak link in the cryptographic chain. Therefore, newer DH groups specified in RFC 3526 provide estimated strength between 90 and 190 bits.

The downside to these newer DH groups is that their greater strength comes at a cost: with the stronger groups, more processing time is required. This is one of the reasons why peers need to negotiate a mutually acceptable DH group. For instance, my phone might not have enough processing power to deal with DH group 15, so it wants to support only DH group 2. While trying to establish an IPsec connection with a server, my phone will propose DH group 2, and if the server supports DH group 2, that group will be used—even if the server could potentially have used a stronger DH group. Of course, if the two peers can't agree on a common DH group, they won't be able to communicate.

That's enough background. Let's talk about IKE.

I Like IKE (and So Should You)

IKE is used to establish an IPsec connection between peers. This connection is called a Security Association (SA). There are two kinds of SAs, the IKE_SA and the CHILD_SA. The IKE_SA is set up first. It is where the shared secret is negotiated over DH and where encryption and hashing algorithms are also negotiated. The CHILD_SA is where network traffic is sent, protected by AH, ESP, or both.

Every request in IKE requires a response, which makes it convenient to think in terms of pairs of messages. The first message pair is called IKE_SA_INIT and is used to decide what cryptographic algorithm and DH group the peers should use. Since cryptography is still being determined during this exchange, this message pair is not encrypted. The next pair of messages is called IKE_SA_AUTH. This exchange authenticates the messages sent during IKE_SA_INT, and proves the identity of both the initiator and the responder. This step is necessary because the first message was sent in the clear—having established a secure channel, the peers now need to prove to each other that they really are who they say they are and that they really meant to start this conversation. The IKE_SA_AUTH message exchange also sets up the first CHILD_SA, which is frequently the only CHILD_SA created between the peers.

A CHILD_SA is a simplex—or one-way—connection, so CHILD_SAs are always set up in pairs. If one SA in a pair is deleted, the other should also be deleted. This is handled in IKE through INFORMATIONAL messages. Per RFC 4306, an INFORMATIONAL message contains zero or more Notification, Delete or Configuration messages. Let's say we have an initiator that's a PC and a responder that's a server. The PC decides to close the CHILD_SA connection

Every request in IKE requires a response, which makes it convenient to think in terms of pairs of messages.

and terminate the VPN. It sends an INFORMATIONAL message with a Delete payload to the server, which identifies the SA to delete by its SPI. The server then deletes this incoming SA and sends a response to the PC to delete its half of the SA. The PC receives this message and deletes its half of the SA, and everything is great.

Of course, things might not always work this way. Either an initiator or a responder might end up with an SA in a "half-closed" state, where one member of the SA pair is closed but the other is still open. The RFC specifies that this is an anomalous condition, but it doesn't allow for a peer to close these half-open connections by itself. Instead, the peer is supposed to delete the IKE_SA if the connection state becomes sufficiently unstable —but deleting the IKE_SA deletes alf the CHILD_SAs that were created beneath it. Either case would be painful on a phone because keeping the CHILD_SA open would consume scarce system resources, as would having to tear down and rebuild the IKE_SA.

Also, it is possible for one peer at the end of an SA to disappear completely, without telling the system on the other side of the SA that it's good. This is a circumstance that is particularly prone to occur with mobile phones. For instance, consider a scenario in which a mobile phone user has established an IPsec VPN connection with a server. The mobile phone user goes into a basement, and loses her radio signal. The server has no way of knowing that the phone has disappeared into a black hole so it continues to send messages on the CHILD_A, but receives no response. It would likewise be possible for the phone to start sending messages into a black hole because of routing issues on the cellular network. Anything that could cause one peer to lose track of another can lead to this condition, but the cost on the phone is greater than the cost on the server because the resources on the phone are more scarce.

Because sending a message on the radio costs power, and power on the phone is limited, the phone needs to detect these black-hole situations and deal with them to conserve system resources. This is usually handled through a process called Dead Peer Detection (DPD), in which a peer that suspects that it might be talking to a black hole sends a message demanding proof of liveness. If the target of this request does not respond in an appropriate amount of time, the sender can take appropriate action to delete the IKE_SA and reclaim the resources being spent on it. In general, it's preferable to send DPD messages only when there's no other traffic traveling through the SA and the peer has reason to suspect that its partner on the SA is no longer there. While there's no requirement to implement DPD this way, it doesn't make much sense to confirm the liveness of a peer that's currently sending you other sorts of network traffic.

Another situation that can cause trouble on a VPN connection is a host changing its IP address. The IP address of a host is used along with the 32-bit SPI to identify a particular host and associate it with an SA. When a host loses its IP address, this association is also lost, and the SA needs to be torn down and re-created with the new IP address.

As we've said before, this is not much of a problem with desktops or laptops. A PC might lose a DHCP lease and get a new IP address, but most DHCP implementations make it quite likely that the PC will be assigned the same IP address it had before. In other words, desktop PCs don't change IP addresses very often. Laptops, because they are mobile, can change their point of network attachment and therefore get a new IP address, forcing any SA they have open to be destroyed and re-created. However, the rate at which laptops switch IP addresses is still relatively infrequent when compared with the rate at which a phone does. For instance, a phone that has the ability to transmit data via Wi-Fi and cellular channels might switch networks every time a user walks in or out of her office building as the phone changes from a Wi-Fi to a GPRS con-

Why Is It Bad to Tear Down and Rebuild the SA?

The short answer is that tearing down and rebuilding the SA uses expensive resources that the phone can't afford to waste. Specifically, CPU is consumed in performing cryptographic calculations and while the radio is in use sending and receiving large amounts of data, both of which cost battery life. The large amount of data being transferred also consumes bandwidth, which costs money—especially in places where data plans charge by the kilobyte.

nection and back again. Unlike a laptop moving around an office building, which might remain on the same network link and therefore continue to have a topologically correct network address without a change, the phone has switched between two fundamentally different networks, so it is virtually guaranteed to change IP addresses. This results in an interrupted connection on the phone whenever a handoff between networks occurs. The same thing can happen while the phone is on the cellular network alone. The phone might enter roaming mode and switch from its home network to a foreign network owned by a different mobile operator. The phone might move from one area of coverage to another and become attached to a completely different part of the mobile operator's network.

There are any number of other reasons controlled by the mobile operator that could cause an interrupted connection. This frequent tearing down and rebuilding of the SA would make the mobile VPN intractable were it not for the extensions to IKEv2 known as MOBIKE.

When a host loses its IP address, the SA needs to be torn down and re-created with the new IP address.

MOBIKE

The IKEv2 MOBIKE protocol is defined in RFC 4555. It allows peers in an IPsec VPN to advertise that they have multiple IP addresses. One of these addresses is associated with the SA for that peer. If the peer is forced to switch its IP address because of a change in network attachment, one of the IP addresses previously identified, or a newly assigned address, can be swapped in without having to tear down and rebuild the SA.

To indicate its ability to use MOBIKE, a peer includes a MOBIKE_SUPPORTED notification in the IKE_SA_AUTH exchange. The IKE_AUTH exchange also includes the additional addresses for the initiator and the responder. The initiator is the device that started the setup of the VPN by sending the first IKE message, and is responsible for making decisions about which of the IP addresses to use from among those it has available and those offered to it by the responder.

As the RFC points out, the initiator is generally the mobile device because the mobile device has more awareness of its position on the network, and as a result it is better suited to make decisions about which addresses to use. However, the RFC does not specify how these decisions should be made. Generally, one end of the IPsec VPN will be a mobile device, and the other end will be a stationary security gateway server. The specification doesn't require this implementation, and does allow both ends of the gateway to move—but it doesn't provide a way for the two ends of the gateway to find each other again if they move at the same time. That is, if one peer updates its address and the other peer does the same thing at the same time, there is no opportunity to communicate this change to either peer, and the VPN connection will be lost.

The initiator uses the responder's address list to figure out the best address pair to use for the SA. The responder doesn't use the initiator's addresses, except as a means of communicating to the initiator that the responder's address has changed.

For instance, when the initiator sees that its address has changed, it notifies the responder of this fact with an INFORMATIONAL message that contains an UPDATE_SA_ADDRESSES notification. This message uses the new address, which also starts being used in the peer's ESP messages. The receiver of the update notification records the new address and optionally checks for return routability to be sure that the address belongs to the other mobile node as is claimed. Following this, the responder starts using the new address for its outgoing ESP traffic.

Of course the initiator or responder might not know all the IP addresses it will ever have for the lifetime of the SA. A peer can advertise a change in the list of addresses it supports with an INFORMATIONAL message. If the peer has only one address, this address is present in the header, and the message contains the NO_ADDITIONAL_ADDRESSES notification. Otherwise, if the peer has multiple addresses, one of these addresses is put in the header of the INFORMATIONAL message, and the others are included in an ADDITIONAL_IP4_ADDRESSES notification.

This list is not an update—it is the whole list of addresses that the peer wants to advertise at that time. In other words, the whole list is sent every time, but this cost is still lower than the cost of tearing down and rebuilding the SA every time the phone changes its point of network attachment.

Wrapping Up

You should now have a basic idea of how an IPsec VPN with MOBIKE would function on a mobile phone.

The growing prevalence of smart phones in the home and workplace is going to make these solutions more important as users begin to demand an experience that matches that on more resource-rich computing devices. For the time being, people are willing to accept phones that can't connect to the corporate network, that have only a day of battery life, and that suffer from the other shortcomings of the smart phone we are all familiar with. This won't last. Competition and the emergence of better hardware will force the adoption of more complete, end-to-end solutions that enable experiences for the phone that are on par with the laptop and desktop.

And then I, along with everyone else, will be able to browse corporate SharePoint sites just by clicking a link on my phone.

Special thanks to Melissa Johnson for her suggestions and technical review of this article. ■

RAMON ARJONA *is an SDET lead at Microsoft.*