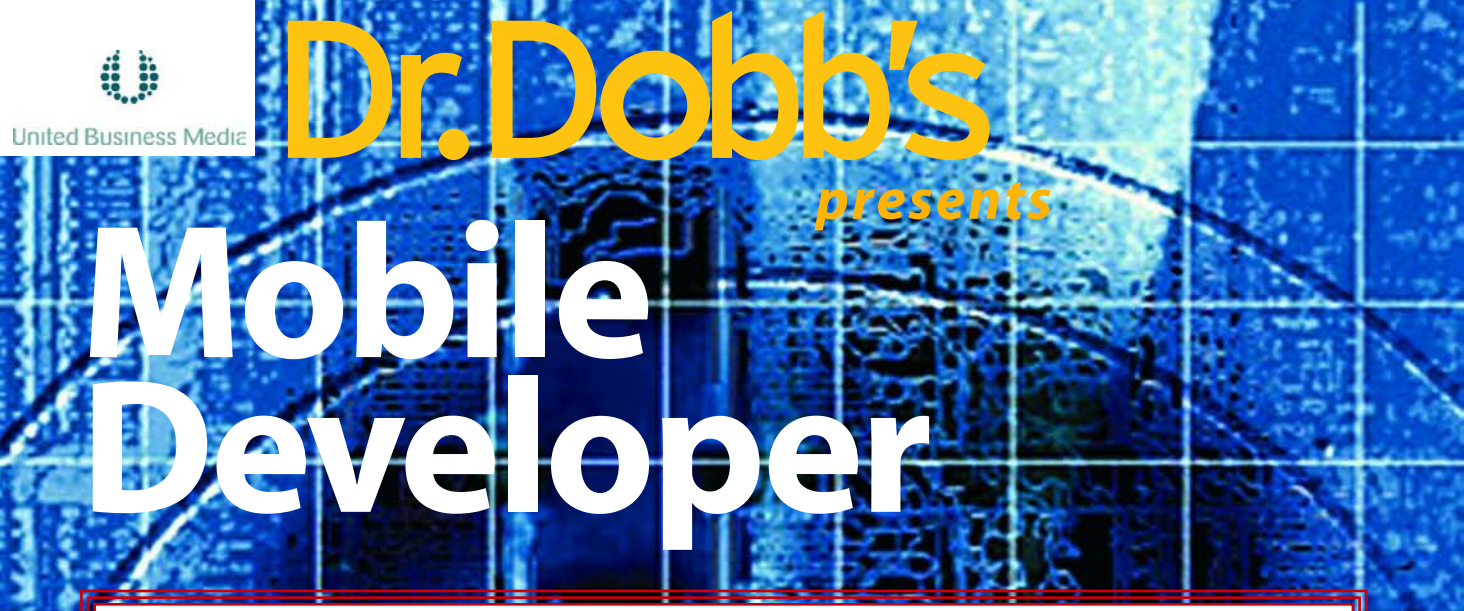


United Business Media

Dr.Dobb's presents

Mobile Developer

Mobile Platforms: A Developer's Perspective

For developers, the battle lines are forming in the mobile wars

by Tom Thompson and Eric J. Bruno

Apple's iPhone and iPad are either loved or loathed, for as many reasons as there are users. However, there's one fact that everyone can agree on about the device: Apple's iOS shook up the mobile industry. Despite Apple's coming late to the party in 2007, by the end of 2008 the iPhone had rocketed to second or third place in U.S. smartphone market and upstaged other vendors who have been selling smartphones for over a decade. Depending upon whose market report you consult, along with the surges and slides of smartphone sales, these positions are subject to change over the coming months, but there's no disputing that the iPhone has altered the landscape forever.

What is the secret sauce to the iOS's success? Feature-wise, the iPhone's hardware hardly makes a compelling case, as it lacks certain features found on other smartphones. The iPad, on the other hand, has an attractive and (for now) unique form-factor. And what both devices do, they do effectively and easily. In a single word, the features that the iPhone and iPad offer are "useable." To prove this, all you have to do is take a look at the picture submission statistics of Flickr, a popular photo-sharing site. The graphs on the site's Camera Finder page, which track the volume of photo submissions by device, reveal that the iPhone postings easily outpaced other smartphones with much better cameras. Furthermore, at the end of 2008, the iPhone temporarily matched the submission volume from several high-end digital cameras. What drove this volume was not the camera capabilities of the iPhone, but that its owner can snap and send photos easily and quickly to any website.

Another factor in the iPhone/iPad's success is its means of application distribution. Apple's App Store leverages the familiarity and infrastructure of Apple's iTune software, which many people already use to search for and purchase music. While network operators have had their own ways to distribute software, it's obvious that the App Store's quick access to the goods has made the difference, as the store was making $1 million per day a month after its launch in 2008.

What Apple has shown is that an easy-to-use platform, with ready access to applications, can carve out a section of the smartphone market filled with established players. Apple has fired the opening shot in the battle for the next stage in personal computing — the era of the smartphone. "Apple's iPhone radically redefined the concept of the smartphone, and RIM's Storm is obviously the first response," said Tom R. Halfhill, senior analyst for In-Stat's *Microprocessor Report*. "The huge software community that Apple is building around the iPhone is as revolutionary as the iPhone's hardware design. Google's Android could expand the concept even further by encouraging a more open approach to both hardware and software development."

The existing stewards of the smartphone market, such as Nokia, Microsoft, and Research In Motion, are not going to stand by and let upstarts like Apple and Google grab marketshare. Already they are beginning to counter with smartphones that offer a touch screen, accelerometers, and location services. However, the situation is complicated by the demands of the network operators, who want to be more than just dumb pipes carrying data and want to make money on services.

As the battle lines form, this is a time of opportunity for developers. Apple's App Store has shown that developers can write software that adds value to the platform — and just as important, distribute them in a way that they can make money. The industry stewards have countered Apple's move with their own application stores, so there's a huge opportunity to write the "killer app" for one of several smartphone platforms.

iPhone/iPad Mobile Application Development

When first announced in 2007, Apple fans declared the iPhone to be revolutionary, while some gave it little chance of dominating in an already-established market segment. Even the biggest of fans couldn't have predicted just how disruptive the iPhone has been to the smartphone industry. It's now seated comfortably in second place behind either Android or RIM (depending on whom you ask), and gaining ground quickly with the iPhone 4. Depending upon whose market report you consult, along with the surges and slides of smartphone sales, these positions are subject to change over the coming months, but there's no disputing that the iPhone has altered the smartphone landscape forever.

The iPhone's success can't be tied to its hardware; as stated previously, it matches what you'd find in many other smartphones. Additionally, some of the newer Android phones exceed even the iPhone 4's hardware specs. However, what the iPhone does do, it does effectively and easily. To sum it up, the features that the iPhone offer are just downright useable.

It's the Software!

By far, one of the largest factors in the iPhone/iPad's success is

Apple's Application Store, which leverages the familiarity and infrastructure of Apple's iTune software. In a way, iTunes and the iPod paved the way to the iPhone/iPad's success, but not because of music. They created a popular infrastructure to distribute bits (music, video, and now apps). While network operators have had their own ways to distribute software, it's obvious that the App Store's quick access to the goods has made the difference, as the store was making $1 million a day per month immediately after its launch in 2008. As of June 2010, there are over 225,000 applications in the App Store, with a total of over 5 billion downloads to date.

Apple proves that an easy-to-use platform with ready access to applications can carve out a section of the smartphone market previously filled with established players. In a nutshell: It's all about the software. With the iPhone, Apple fired the opening shot in the battle for the next stage in personal computing — the era of the smartphone. The iPad has further expanded Apple's pioneering of this new era of personal computing.

Apple iPhone Revisions

Apple introduced its first iPhone in September 2007, and 11 months later in July 2008 followed that with the iPhone 3G. By the end of 2008, the company sold over 10 million iPhones, to capture one percent of the mobile phone market. Not a bad beginning for a newcomer to the industry. This success occurred despite the fact that the device has a low-resolution 2 megapixel camera, and lacked some common smartphone features such video recording, voice dialing, and a to-do list.

Apple has continued its intense focus on the platform with the launch of the iPhone 3GS in 2009, and the iPhone 4 in 2010. As a result, this focus and intensity has spilled over into the marketplace. What other device do you know of that creates such hype that its consumers wait in lines in all sorts of weather to be the first to own them? Each subsequent year has seen a new iPhone device and updated software release that delivers huge new feature sets to meet the demand of its users. This continual focus on — and improvement of — the platform has allowed consumers to forgive Apple for the device's shortcomings.

Honestly, software isn't the only factor that has led to the iPhone's success. In a way, it's the lack of hardware that many say is its best feature; Apple dispensed with the phone keypad and physical buttons. The iPhone's touch screen, which displays whatever the user needs at the moment (a web page, the latest e-mail, a map displaying your current location, an iPod music selection screen, and so on) is what drives its users to use it more. Getting to a feature simply takes one or a few taps on the screen. As a result iPhone applications display large, uncluttered interfaces, and accomplish just one purpose.

Apple is able to integrate its software well with the device because the company builds the iPhone's hardware and its software stack, similar to the smartphone stewards.

Some of the best capabilities of the iPhone are:

- It's a web browser, with a best of breed experience
- It's an iPod, able to download, select and play music
- Advanced e-mail support and interface (especially with the iPad)
- It's an application platform able to run useful utilities, access online content, and play immersive games
- It has enterprise support, such as synchronization to Exchange, and the ability to establish a VPN connection
- Location-based APIs using the device's GPS feature provide position information, valuable for the creation of new forms of local presence applications and social networking
- APIs permit access to all of the hardware (i.e. accelerometer, gyroscope, and GPS) allowing for the development of novel applications and games
- HD video recording, powerful editing software on the device, and the ease of publishing make the iPhone a standalone video production system
- Two-way video conferencing via Facetime

Dr.Dobb's

EDITOR-IN-CHIEF
Jonathan Erickson

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

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

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

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

UBM LLC

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

TechWeb

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

techweb™

- Voice dialing and voice memo recording
- Accelerometer, gyroscope, and GPS make for revolutionary applications and games

However, both the iPhone and iPad present a split personality in terms of fantastic features and miserable limitations, such as:

- The iPhone has one of the best browsing experiences for a smartphone. Its WebKit-based Mobile Safari browser brings a desktop-like browsing experience to the device. It conforms to AJAX, ECMAScript 3, CSS 2.1 and partial CCS 3, DOM 2, HTML 4.01, and XHTML 1.0 standards, HTML5 new input types, and renders almost all web sites accurately. The multi-touch gestures interface lets you zoom in and interact with the content easily. Combined with the iPad's large screen, or the iPhone 4's Retina Display, web browsing on Apple's devices is a pleasure.
- Both the iPad and iPhone are also an iPod, with some unique features that make them the best iPods available. And again due to iPad's large screen and iPhone 4's display, watching video or viewing pictures is sheer joy on the devices.
- On the other hand, there's no Java or Flash support.
- You can't easily download files. Apple has intentionally hidden access to the file system on the iPhone and iPad, although they do lurk behind the UI curtain. Applications are available — with new ones appearing each day — which edit and display files on the devices, and sync them with the desktop.
- The iPad has no video or picture camera capabilities.
- The iPad doesn't support USB device connectivity besides those intended by Apple (i.e. you cannot connect an external video camera to the iPad).

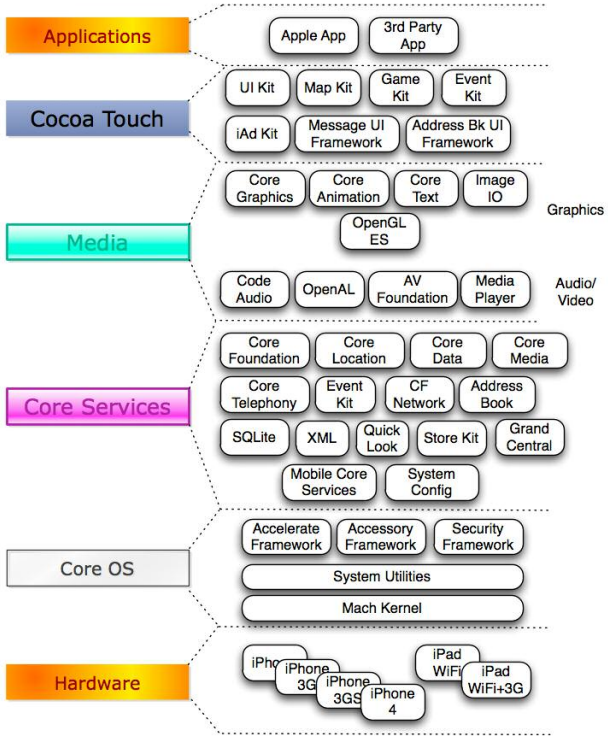


Figure 1: The IPhone/IOS software stack.

- A built-in e-mail program uses a virtual keyboard to type in messages with excellent error-correction, but you can view only certain types of e-mail attachments.
- While you can customize multiple screens full of applications, and even create folders to group them, overall the UI appears to have not been designed to handle as many applications as most users download to their devices.
- While the iPad displays its home screens in landscape mode (rotating the icons accordingly), the iPhone still does not.
- Although mutitasking on the iPhone is finally supported, accessing the feature via the home button can sometimes be clumsy.

Everywhere you look, the iPhone exhibits this Jekyl-and-Hyde personality of performing most tasks very well, but there are those odd omissions. Until recently, copy-and-paste and multitasking were among those missing features. And have you tried to print from an iPad?

One of the smartest additions to the iPhone OS has been enterprise support. For example, the iPhone works with Microsoft Exchange Server. This allows it to support enterprise operations via ActiveSync technology to remotely manage iPhones, and allow users to securely log into the corporate intranet. It allows VPN connections that support Cisco IPSec, L2TP/IPSec, PPTP, and certificate-based authentication (PKCS1, PKCS12). ActiveSync's direct push feature sends e-mail, calendar, and contact updates to the device seamlessly. For non-Exchange users, Apple offers iPhone integration with MobleMe and Google apps.

With iOS 4, iPhone finally supports multitasking, but it's much different than on the desktop. Apple applications can run concurrently, but this concurrency is limited for third-party applications. The iPhone's multitasking is more like fast task switching, where an application is suspended (not shutdown as it used to be) so that it can be quickly resumed when the user returns to it. This compromise was made to maintain battery life and overall performance, but it's a big improvement over iPhone Os 3.x. This, combined with push notifications for applications that require them (i.e. VOIP, instant messengers, and so on), offers enough background processing power for most people.

Inside the iPhone/iPad Software: iOS

The iPhone/iPad OS and software has grown considerably since its debut, and has recently even changed its name. Let's dive into the iPhone/iOS software stack (Figure 1) and take a comprehensive look at what's inside for developers. There's a lot here, so let's discuss what each layer of the stack does:

- Cocoa Touch: The top layer of the stack, upon which applications run, consists of frameworks that manage the UI, such as capturing events, managing windows, and displaying graphics within these windows. Cocoa Touch is a subset of Apple's Cocoa, which are object-oriented frameworks written in Objective-C. Cocoa provides many classes, or components, from which you can build a full-featured application. However, Cocoa Touch frameworks are tailored for the constraints of the smartphone platform. These frameworks strike the right balance between

abstracting much of the iPhone's low-level hardware, while still enabling you to use device-specific features. For example, the Cocoa Touch layer handles tasks such as updating the user interface and playing media, yet its APIs allow you access to the accelerometer, camera, and GPS.

- Media layer: manages all graphics rendering, audio generation, and playback of audio or video files. While Cocoa Touch provides the high-level means to generate animations and graphics, you can use the frameworks in this layer to exert fine-grained control over the display of your content. It consists of the following main components:
  - OpenGL ES: conforms to the OpenGL ES 1.1 specification and displays three-dimensional objects. This framework uses the device's graphic accelerator to provide full-screen animations at high frame rates; a valuable capability for games.
  - Core Graphics (Quartz): a vector-based graphics engine that handles 2D drawing and special effects. The Quartz engine is identical to the one found in Mac OS X.
  - Core Animation: supports sophisticated animation and visual effects, where the image compositing required to accomplish them is performed in hardware. Playback and recording for audio files and streams is handled here, and a media player framework provides full-screen playback of video files of several format types.
  - Core Audio: handles stereo-based audio playback.
  - OpenAL: positional-based high-quality audio for games.
- Core Services: provides system services for the higher layers. It consists of frameworks and engines that support an address book, a SQL database (SQLite), location services (using GPS coordinates), networking services, telephony, and so on. A security framework manages the digital certificates, keys, and access policies that can protect an application's data. Let's look at some important ones:
  - Grand central: allows multithreading within applications seamlessly and efficiently. My prediction is that while this may be helpful with multitasking today, it's going to play a larger role with iPhones built around multicore processors and GPUs in the future.
  - CFNetwork: abstracts and simplifies network communications and protocols. This is especially important for a device with so many forms of communication (3G/Edge, Wifi, and BlueTooth).
  - Location: abstracts the positioning capabilities of the device for applications to use in their own unique ways.
  - Store Kit/In App Framework: for statistics about purchases, in-app purchases, and future application monetization (i.e. e-books, magazine issues, and so on).
- Core OS: implements basic OS services, and consists of the kernel, drivers, and OS interfaces. The kernel is based on Mach (as is OS X), and manages low-level functions such as virtual memory, POSIX threads, BSD sockets, math computations, filesystem access, and so on. Only a select few higher-level frameworks have access to the kernel and drivers. If necessary, an application can indirectly access some of these services through C-based interfaces provided in a LibSystem library.
- Hardware: all of the latest iOS features run on the iPhone 4, most run on the iPhone 3GS (i.e. no Facetime), a small subset runs on the iPhone 3G, and none of it runs on the original iPhone — iOS doesn't support the oldest hardware. iPad runs a special version of iPhone OS 3.x, but that's expected to change in the fall when Apple releases a special iOS version for iPad.

The iPhone OS stack has a rich legacy. Cocoa, the Mach kernel, and certain Core layer components had their start with the original 68030-based NeXTSTEP computer in 1989. Over the years, this software was ported to PowerPC- and Intel-based platforms, and finally to the iPhone's ARM processor. As a consequence, the code has been subjected to rigorous examination during these ports. It has also undergone extensive field-testing, since the Cocoa frameworks, Quartz, and Mach kernel are part of Mac OS X for millions of Macintosh computers. The iPhone OS can be considered a stripped-down version of Mac OS X (although it does have many unique features of its own), thus bringing some of that operating system's reliability and multitasking capabilities to the

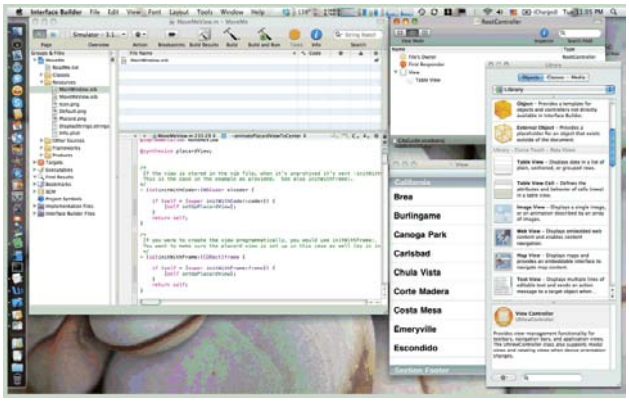


Figure 2: The XCode development environment, with Interface Builder.

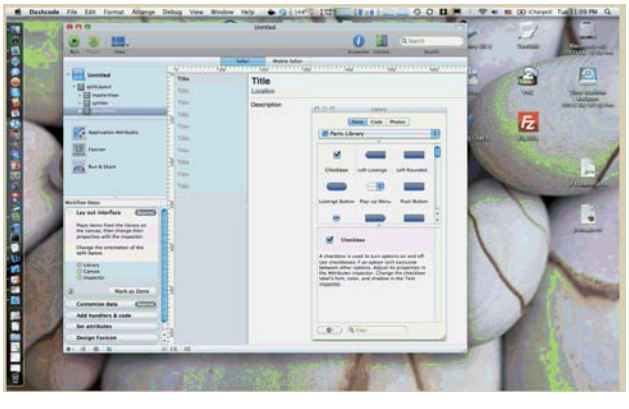


Figure 3: Dashcode allows you to build mobile HTML/JavaScript applications.

platform. In a nutshell, if you've developed applications of Mac OS X or even NeXTSTEP, you'll be at home with iPhone OS and iOS.

However, there are fundamental differences between the iPhone OS and Mac OS X, given that one runs on an embedded smartphone and the other on a desktop computer. One difference is the amount of RAM available, and even that varies greatly between devices:

- iPhone and iPhone 3G: 128M of RAM
- iPhone 3GS: 256M of RAM
- iPad: 256M of RAM
- iPhone 4: 512M RAM

Simply put, depending on the device and what it's currently running, there may not be much free memory available at any given time.

iPhone Development Tools

You write iPhone applications using XCode, which is an IDE developed and maintained by Apple (see Figure 2). Native applications are written in Objective-C, which is an object-oriented Smalltalk-like extension to C. Unlike C++, Objective- C is both a language and a runtime. Also, Objective-C's syntax for objected-oriented operations is significantly different from those used in C++. In short, it takes some effort to get familiar with the language's syntax and idiosyncrasies.

XCode's development tools allow you to debug iPhone/iPad applications in a device simulator, and you can trace and place breakpoints at the source code level. The SDK that provides the Cocoa Touch frameworks and simulator is a free download. However, to run your program on an actual iPhone, the application must be signed, which requires that you purchase a signing certificate. With an enterprise signing certificate, you can distribute provision profiles that allow internally developed applications to execute only on those iPhones that have installed the profile.

For web-based applications, the SDK provides Dashcode (see Figure 3), which is a framework based on a web page composed of HTML and JavaScript. You can use DashCode's simulator to write and test your web application. You can also use several other third-party frameworks to write web applications, and debug these with Aptana Studio's tools.

All iPhone applications are distributed via Apple's App Store, and they must be digitally signed. The App Stores uses the familiar iTunes front-end to both display and handle purchases. Applications can be obtained through iTunes on a desktop computer with the iPhone tethered to it by a USB cable, or over-the-air (OTA) via an App Store program on the iPhone or iPad. The revenue model is clear and simple: 70% of the revenue goes to the developer, and 30% goes to Apple. There have been several cases where developers have made small fortunes with highly popular applications. Other smartphone vendors have quickly adopted similar distribution and revenue arrangements.

Apple reviews the application you submit before publishing it in the App Store. This review can screen applications of questionable content or malicious code, but it's also been used to quash applications that "duplicate the functionality" of the iPhone. This is unfortunate, as developers could fill the gaps or improve the features of the device. For example, adding some useful Bluetooth profiles that supported stereo headsets, data synchronization, or the ability to implement multiplayer games would be useful and make money.

Beyond XCode, Interface Builder, and Dashcode, Apple provides other unique tools that allow you to debug and tune your application precisely for Apple's mobile devices. For instance, Apple offers the Instruments Application, which allows you to gather performance data such as CPU, memory, filesystem, and network bandwidth usage. It also measures your application's affect on the device's battery (whether it's abnormally draining it), and works in both the simulator and on a real device.

Futher, Apple's Shark application uses these statistics and goes even lower into the system to measure details such as multithreading issues, inefficient use of system calls, interrupt rates, and even the frame-rate at which your application updates the screen. In a nutshell, Shark supports low-level application and system profiling to ensure your application is running as efficiently as possible, and intended, on the iOS device you're targeting.

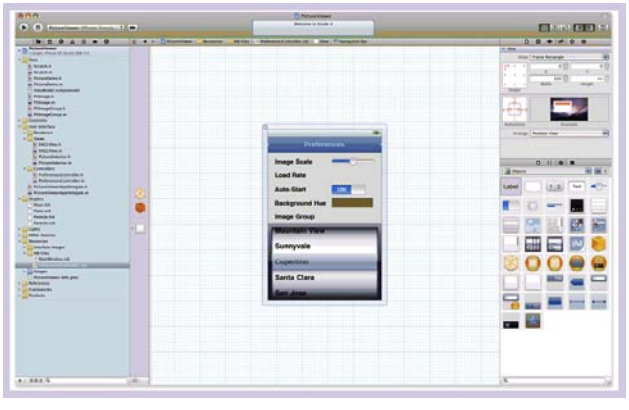


Figure 4: XCode 4's unified IDE window (picture courtesy Apple, Inc.)

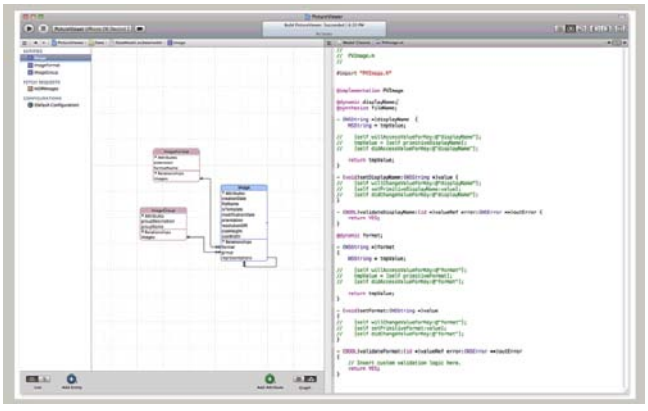


Figure 5: The new XCode Assistant (picture courtesy Apple, Inc.)

iPhone/iOS Platform Summary

Despite some missing features, the iPhone's ease-of-use advantage, elegant design, and integrated feature set has made it a wildly popular device. However, as this article points out, smartphone industry stewards have worked to level the playing field in terms of their devices and software. In turn, Apple continues to counter with additional software features and improved iPhone models.

Here are some iPhone's pros:

- Best handheld mobile platform that happens to be a phone
- Ease-of-use makes it a popular platform, overcoming its other shortcomings
- You can make some serious money with a successful app
- Millions of them out there and growing
- App store model has proven very successful
- Continual hardware and software improvements

Cons:

- The need to learn Objective-C (the only smartphone platform that uses it)
- Lots of applications in the App Store make it hard to stand out
- Competitors (i.e. Android) continue to catch up on the UI
- Operator lock-in (as of this writing)

The Future

At the 2010 World-Wide Developer Conference (WWDC), Apple announced that the next version of XCode (version 4) would sport a unified development environment. This means that code editing, debugging, and UI building (i.e. Interface Builder) will all be contained within one application window as opposed to being spread out as it is today (see Figure 4).

Additionally, a new XCode Assistant feature will provide a graphical representation of your code and its classes (see Figure 5). Here, you can visually explore an application's source code, and make code modifications directly via the graphical representation thanks to its full round-trip support.

Apple is planning to blast its compiler into the future with the release of its new open-source LLVM Compiler 2.0 for C, C++, and Objective-C. Not only does it support dramatically improved compilation times, but its integration into XCode will bring improved syntax highlighting, code suggestions, and code completion to the IDE.

Here's a summary of some other new features:

- Fix-It: brings common and not-so-obvious coding mistakes to your attention.
- Version Editor: a new built-in source-code management tool.
- Debugger: an open-source LLVM.org debugger that consumes fewer resources, performs better, and makes it easy to debug multithreaded parallel processing applications.

As if it weren't clear by its feature set alone, the power, depth, and breadth of its developer tools prove that Apple's iOS and its devices have become a mainstream application platform. What was once considered embedded application development has become the next personal computing programming environment, thanks a great deal to Apple's iPhone and other iOS devices.

Android Mobile Application Development

Like Apple, Google jumped into the smartphone business relatively recently. After months of rumors, the company finally launched its Android platform in late 2007. Android is a smartphone stack developed for the Open Handset Alliance, which is an association of 65 hardware, software, and telecommunications companies. The goal was to create a smartphone / mobile device software stack in the open, based on open source components, and driven by the open source community. In the fall of 2008, T-Mobile released the HTC G1 mobile device, which was the first smartphone to use the Android platform.

According to *Admob*, in January 2009 Android generated three percent of the smartphone network traffic — a significant jump considering the platform had only been on the market a few months. This astonishing trend certainly has continued, as there are now dozens of Android-powered devices currently in the marketplace and multiple operators supporting them. In fact, according to a recent Gartner report, Android edged out Apple's iOS-based devices into third place for the second quarter of 2010, behind only Symbian and RIM-based devices, first and second, respectively. According to some reports, Android usage stats (derived from web server logs) have been edging out even Apple iPhone usage stats, although this varies from month to month, and from source to source.

Inside the Android Platform

The Android platform is built upon the Linux 2.6 kernel, which can be traced back to the first Linux source code release in 1991. Since then Linux has evolved, and its code has been extensively reviewed and field-tested on just about every type of computer architecture, from embedded systems to mainframes. Linux therefore provides a solid foundation upon which the Android platform is built. In addition, Android is distributed under the Apache license, which means all of its software components are free and the source code available with few restrictions.

Some of the popular features of the Android platform are:

- It's built on Linux, a proven OS
- It's an open source platform
- There is a growing community of developers that contribute applications and features to the platform
- Multiple third-party applications can freely execute at the same time
- Applications can intercommunicate and share resources
- The SDK supports a wide range of hardware capabilities
- You use a Java-like language, set of libraries, and virtual machine to develop applications

Let's take a look at some of the software that's shipped with Android and most modern Android-based devices.

Android Applications

As you would expect from a smartphone platform, Android ships with a large number of useful applications, including:

- An e-mail program based on Google's GMail
- A mapping program based on Google Maps
- Its own unique browser that uses WebKit, with support for Chome's V8 JavaScript engine, HTML5, Flash, embedded video playback, and so on
- The Android Market (application store)
- Google Voice
- Google Sky Map
- Google Translate
- Google Shopper
- Google Listen (for podcasts)
- My Tracks (exercise companion application)
- and more

Some popular third-party applications that often ship with Android-based devices include social networking applications such as Facebook, Twitter, and MySpace. As of July of 2010, there were over 70,000 applications available in the Android Marketplace, with over 1 billion downloads to date.

As with all platforms, Android has its own shortcomings. For instance, until more recently, there was no support for on-screen keyboards, Microsoft Exchange, Bluetooth, or Adobe's Flash. However, all of these features, and them some, have been added to the platform over time.

Android Application Development

Being a Linux-based OS, Android supports C++ for native application development. However, most Android applications are written in a Java-like language, although Android is not Java ME, nor does it support such applications. In fact, Google and Android are the targets of a recent patent infringement lawsuit where Oracle states that Google did not properly license Java technology. Although we won't get into the legal issues in this article, we examine the complete software stack later to discuss the technical issues surrounding this issue.

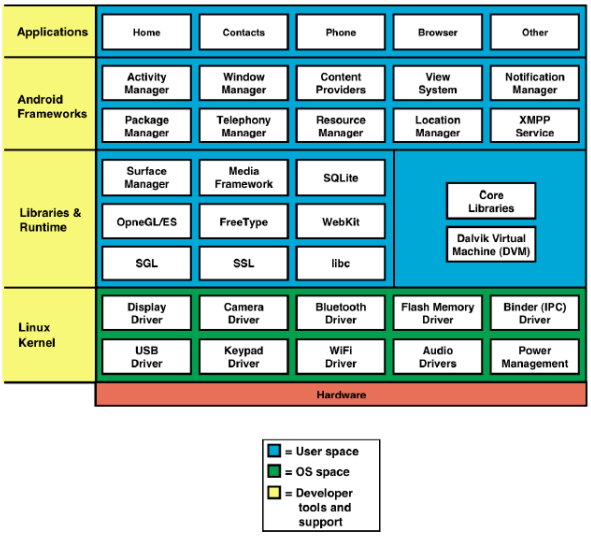


Figure 6: The Android software stack and APIs.

Like most smartphones, Android can handle changes in screen orientation, and provides access to camera, location, and accelerometer data. Android applications are designed to operate concurrently, with complete multitasking freedom provided to all applications. However, in practice, the heavy use of the smartphone's CPU and other resources when you run multiple applications concurrently can significantly shorten the device's battery life. This is the main reason that Apple has provided for the iPhone's lack of complete multitasking support.

The Android APIs support location-based services, a multitouch interface with gestures, access to all of the hardware such as GPS, camera, video, network, voice services, and so on. Let's now examine the Android software stack (Figure 6) to see how it implements the platform.

Here's how the layers are broken down:

- Applications: hosts all Android and third-party applications, where multiple third-party applications can execute concurrently. However, as noted, doing so can shorten battery life.
- Android Frameworks: consists of a Java-like library that provides application functions such as window management, window content display, application messaging, telephony, location services, and so on. The interface Android presents at this level is similar to the Java programming language, and because the source code is available, you can modify these classes to extend or override their behavior. Note that some of the lower levels in the stack present C++ interfaces.
- Libraries and Runtime: implements 2D and 3D graphics support, along with multimedia content decoding. This layer has engines to support application features, such as a SQLite database engine and WebKit, which handles web content rendering.
- Linux Kernel: consists of most of the Linux kernel and OS components to provide preemptive multitasking, system services such as threads, networking services, and process management, and to manage and provide access to all of the device's hardware.

Like Java ME, Android strives for hardware independence by using a bytecode interpreter to execute Android applications. However, it doesn't use Sun's JVM, but instead uses its own Dalvik Virtual Machine (DVM), which supports its own version of just-in-time compilation for fast code execution. Dalvik was designed to run multiple Android applications; each in their own protected memory space, and in some cases each in their own Dalvik instance. Everything about it is designed to run on low-memory devices efficiently. While this approach offers stability and a robust environment for running multiple applications, it does so at the expense of compatibility with Java ME applications.

Android Developer Tools

The programming language of choice for the Android platform is — no surprise here — Java, although you can use C++ if you're

writing code that might reside in the other layers. With the Android Development Tools (ADT) plug-in, you can use Eclipse to write and debug your applications. Other Java IDEs, such as IntelliJ, can be used. The external designer tool, DroidDraw (droiddraw.org), is an excellent design-time layout tool. The Android SDK (now at version 2.2) is a free download and contains a comprehensive set of development tools including a source-level debugger, a handset emulator, and software to support deployment and debugging directly on actual Android handsets.

Programming with the Android SDK requires the Java Development Kit, Apache Ant, and Python v2.2 or later, installed on Apple's Mac OS X, Microsoft Windows XP or Vista, or Linux. Android's core Java libraries are derived from the Apache Harmony project, and are as compliant with Java SE as Google could make them. Seasoned Java programmers will find the Android SDK an amalgam of Java SE and Java ME methods and classes, along with unique new ones that address the needs of mobile platforms.

In short, while Java programming experience is valuable, you still have to master the SDK's additional classes and methods. In terms of application software packaging, a manifest file serves a similar function as with Java ME, since it describes the contents of the archive file and its entry point. However, the Android manifest also specifies the application's permissions — required for sharing resources — and the events that the application can handle. Additional XML files describe the application's UI elements, and how they are laid out on the screen. The end result of these extensions means that the Java-like Android applications you write won't run on other Java SE or ME platforms. However, this can also be said of Java applications written for RIM BlackBerry devices.

The developer tools compile the Java code to generate Dalvik bytecode, saved in files with the extension .dex. These files, along with the manifest, graphics files, and XML files, are packaged into a .apk file, which is similar to a Java archive (JAR) file. Fortunately, you can use the existing Java signing tools — keytool and jarsigner — to create keys and sign your Android application. All Android applications must be signed or the platform won't execute them.

While the SDK provides a developer key for use during code test, you must generate a private key to sign the application when it is ready for public distribution. The private key is required to identify the author of the application, and to establish relationships between applications so that they can share code or information. The certificate that you use to generate the private key does not require a signing authority, and you can use self-signed certificates for this purpose.

The Android Developer Phone

At the end of 2008, Google announced the availability of the Android Developer Phone 1, which is an HTC phone with the Android platform installed. The dev phone is an unlocked device that is designed for advanced developers, without the cash outlay or contract commitments often required when developing for other mobile platforms. To purchase one, you must register as an Android developer on the Android Market site, which currently costs $25, and is something you'll eventually need to do anyway. There is also a limit of 10 devices per developer account.

The Android Dev Phone version 1 is a special version of the HTC Dream, with a touch screen; touch ball navigation device; camera; GPS; WiFi; Bluetooth; 3G CDMA chipset; and a physical sliding keyboard. The Android Dev Phone version 2 is a version of the HTC Magic. In summary, there's more memory, a faster processor, additional features such as USB support, and support for multiple cell networks (beyond just T-Mobile).

The Android Market

Google launched the Android Market, where developer's can make their applications available to the public for purchase. The requirements are much more relaxed when compared to Apple's iPhone App Store and approval process. The only requirements are that the application must be signed, and you must pay a $25 USD application fee. As with Apple's App Store, 30% of your applications' revenues go to Google, and you keep the remaining 70%. The Android Market offers you the ability to copy protect the applications you sell, thereby preventing end users from freely duplicating your application on other devices.

Unlike Apple, Google doesn't review the applications submitted to Android Market, although the user community can rate the application. This arrangement gives you a shot at purchasing or selling an application that is truly fantastic, and enhances the platform. On the other hand, it's possible to purchase or obtain a malicious application before the user community detects it.

As of Android version 1.5, Google opened up the AppWidget framework to all developers, allowing anyone to create applications that install as home screen widgets alongside those that ship with the platform. Additionally, Google recently announced the availability of App Inventor for Android, which is a web-based IDE for beginning Android application developers. This development environment gives you access to most of the device's capabilities, including location services, phone functions, SMS, contacts, storage, and cloud-based services.

The Future

Google is expected to announce the new feature set included in its upcoming version 3.0 Gingerbread release. Confirmed already are new features such as WebM video playback, and much improved copy-paste functionality. Rumored features include Android Market music purchases, the playback of media streamed from a desktop computer, support for larger and higher resolution screens, video chat, enhanced voice commands, and a new and easier to use home screen and overall UI.

The real question is what's going to happen between Oracle and Google, and how might this affect the Android SDK? One can only speculate at the moment, but with the huge amount of momentum Android currently has, you can only hope that these two industry giants find some way to work things out to the benefit of all parties

involved: Google, Oracle, the handset manufacturers, and the developers investing in the platform.

Android Summary

Like the iPhone, Android does an excellent job of meeting users' expectations in terms of applications and overall functionality, despite its missing or still-evolving features. Where Android truly excels, over the iPhone in particular, is in its integration with cloud-based services, beginning with Google's own cloud-based applications. Additional cloud integration includes Amazon's APIs as well.

Here are some of Android's pros:

- It's an open source, open platform, based on the very liberal Apache licensing
- Community-based development and improvement (i.e. you can find alternatives to just about any application if you don't like one that ships with the platform)
- Quick releases by Google and Open Handset Alliance means new features, faster
- Dozens of Android devices are available from multiple vendors, on more than one cell carrier network
- New devices with greater capability are being released constantly
- More competition in the Android marketplace is beneficial to the consumer
- Full multitasking support
- A Java-like language and development environment
- Support for existing Java libraries and tools
- Linux underpinnings provide a robust, full-featured, desktop-like OS environment

Cons:

- Many of the mobile Java classes are specific to the platform, and tie applications that you write to it. However, this is also true of RIM's Blackberry classes, or Apple's Objective-C classes.
- Running multiple applications has its advantages, but can also shorten battery life significantly
- Difficult to upgrade existing devices
- Lack of application approval process places users at risk of virus infections
- Copy protection and software pirating is still an issue that results in lost revenue
- Fragmentation is occurring across of the market of Android-based devices. This makes it difficult to ensure your application will run on all (or even a wide-range of) devices
- Recent legal activities between Oracle and Google over Java patent infringement can make you nervous

Nokia Mobile Development: S60 and WRT

Without a doubt, Nokia is the steward of the mobile phone market. It dominates the smartphone industry, both in marketshare and sheer volume of devices sold. The Finnish company sold its first mobile phone in 1992, and it currently retains 34.2% of the marketshare (2Q2010). This is down slightly from 2008, where it owned 38.6% of the world marketshare, according to ABI Research. Symbian, the OS platform for S60-based Nokia smartphones, is still in first place with 41.2% of the market. However, all smartphone manufacturers are pressuring Nokia at the moment, namely Apple's iPhone and Android-based devices (HTC and Motorola in particular).

What Is Haptic Feedback?

In the strictest sense, haptic feedback is a device's ability to provide a touch-based response to the user's actions. Such an example is when a fly-by-wire airplane system provides force feedback to the pilot via the stick when a dangerous maneuver is attempted, or an aerodynamic stall is anticipated. For a cell phone, haptic feedback can come in the form of vibration, or tactile response on the touch screen to simulate objects (such as keyboard keys). With such a display, the user can feel with his finger what he sees on the screen. Nokia has demonstrated this on more than one occasion, and has built support for a full-featured haptic system. For instance, instead of just simulating the feel of a keyboard, Nokia's haptic system can simulate the feel of the key actually being depressed.

Admob, which tracks online requests to websites by platform, pegged Nokia's share in mobile network traffic at 41%. In terms of devices shipped, the numbers are impressive: In 2005, the billionth (yes, that's with a "b") Nokia phone was sold. Nokia's smartphone platform, the S60, was introduced in 2001 and the first handset that incorporated the platform was the Nokia 7650. While smartphones are a subset of the Nokia's product line, the numbers are formidable: Nokia shipped over 105 million S60-bases devices in 2009 (again, down from 180 million in 2008).

The S60 platform, currently in its fifth revision and based on Symbian OS v9.4, is a smartphone reference design that provides

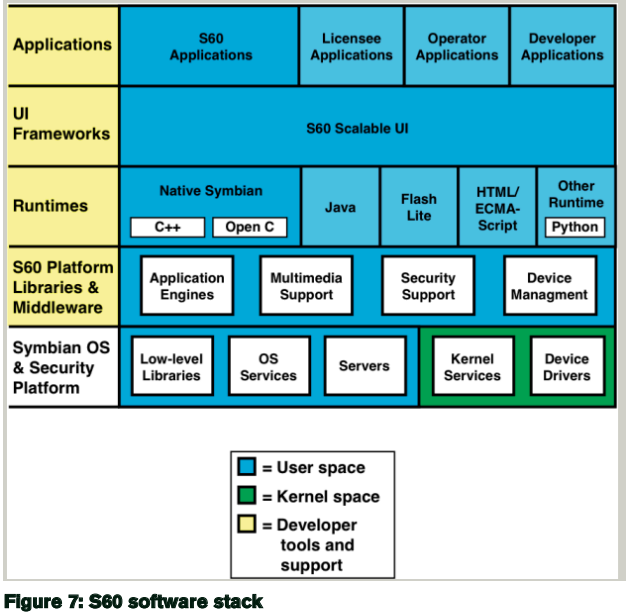


Figure 7: S60 software stack

a consistent interface and execution environment across the various devices made by Nokia and its licensees, which currently include Siemens, Samsung, Lenovo, Panasonic, and LG. Some of the S60 Platform's key features are:

- Multitasking operating system that executes multiple third-party apps simultaneously
- A real-time kernel
- Memory protection that implements a robust and secure execution environment
- Power management software conserves energy on battery-powered devices
- Bluetooth TCP/IP connectivity
- Asian language support
- Multi-touch interface with haptic feedback (available in the fifth edition)
- Can execute Java ME applications
- Has enablers for enterprise support, such as push-based e-mail and protocols for synchronizing Exchange e-mails and contacts
- Numerous avenues to develop S60 applications, using different programming languages and several sets of tools
- Support for high-resolution (640 x 360) touch screens

To see how the S60 platform implements these features, a tour of its software stack is in order. The platform's layered structure (Figure 7) allows it to address the different needs of Nokia, the network operator, and the developer.

The topmost layer is the Applications Layer, which is where native S60, licensee, operator, and third-party applications execute. Beneath this layer is the S60 Scalable UI layer, which is developed and licensed by Nokia, and consists of UI frameworks such as Nokia's Qt. The visible components of these frameworks adjust for different screen sizes, or when the device switches between portrait and landscape screen orientations. Scalable Vector Graphics – Tiny (SVG-T) and relative positioning are used to achieve this capability. The UI components also adopt the device's internationalization settings, such as language/font sets, which direction to draw text, and how to display the calendar and time. For the developer, the UI framework provides a toolkit of ready-to-use components that implement lists, editable forms, grids, notification screens, content viewers, and other visual UI elements.

The next layer in the S60 platform stack is the Runtimes layer. As its name implies, this layer contains the runtime libraries that support various programming languages, interactive content display, and web rendering. For example, the runtime libraries for Symbian C++, Open C/C++, and Java ME all reside here. Web support for the S60's web browser and for Web Runtime (WRT) (http://www.forum.nokia.com/Develop/Web/Tools/) widgets is provided via a WebKit rendering engine. The Runtimes layer also supports bindings for other execution modules such as Flash Lite, Python, and Perl. We examine the WRT later in this article.

The Platform Libraries and Middleware Layer contains the frameworks used to implement specific services for layers higher in the stack, or expose lower level OS APIs in the stack to the developer. It contains various application engines that manage Personal Information Management (PIM) data, messaging, and data synchronization. Multimedia content rendering — such as video decoding — is performed here. Camera and audio recording interfaces are provided for multimedia applications. A security framework manages the security certificates and keys used for secure data sessions.

Inside the S60's Kernel

The Symbian OS and Security Platform Layer resides at the lowest part of the stack. The Symbian OS provides fundamental system services for the S60 Platform. The OS is microkernel-based, with the microkernel residing in kernel space, while the rest of the OS and software stack resides in user space. This provides security and memory protection for vital low-level services. In addition, the OS places each application into its own separate address space that isolates it from the OS and other applications for security and reliability. The Security Layer helps an IT department manage its smartphones, allowing an enterprise to lock or wipe a lost smartphone.

The OS also supports the concept of Data Caging to control access to the file system and ensure data remains secure. To implement this, each application is given its own private file store, which cannot be accessed by other applications unless explicitly granted to do so. Other parts of the file system are accessible in order to promote resource sharing, although some parts are read-only to ensure a level of data integrity and security.

The Symbian OS has undergone extensive field-testing, having evolved from the 32-bit EPOC OS in 1994. The Nokia S60 Platform uses the latest Symbian OS v9, which has an EKA2 real-time kernel. The real-time capability is required to properly manage a device's telephony and multimedia functions. The Symbian OS has a plug-in framework that enables the easy addition of new peripherals, where drivers and supporting frameworks can be "snapped" into the system without having to modify the OS or other software.

A client-server mechanism provides access to low-level system resources, and in fact the kernel itself is a server that parcels out resources to those applications that need them. This transaction scheme allows applications to exchange data without requiring direct access to the OS space.

Web Runtime (WRT) Development

The Nokia Web Runtime is one of two key development environments for the S60 platform; the other being Qt. A WRT widget is essentially a web page containing HTML and ECMAScript, stored in a .zip file. The widget's markup language and scripts execute locally on the phone, analogous to an Ajax application in a browser. In the latest release of the S60 platform, WRT widgets can, through JavaScript extensions, directly access the device's functionality, such as contact lists or the location API. The goal of WRT is to make simple and lightweight, yet useful, applications that look, feel, and act as native S60 applications — a goal that is very well achieved.

Although some applications still require a native implementation (i.e. games), the WRT will suit most applications. In fact, WRT applications have an advantage since they're easily shared and distributed (i.e. an e-mail with the application archive file is all that's required), they can access the device's capabilities, they can share both device and web-based data, and they benefit from the WRT's built-in internationalization support. The WRT is template-driven to allow for quick application assembly, even by hobbyists who want to try something new. Additionally, Nokia supports hybrid applications, where the WRT and native code is combined.

As for application and developer support, Nokia maintains an online developer forum to not only support learning, but also the sharing of code and entire WRT applications. There are online discussions, video tutorials, pod casts, and other material to help. Beyond that, the Symbian OS has an independent following of its own, and an ecosystem of developers that can be turned to for support if needed.

Nokia's Ovi Store (https://store.ovi.com/) is a marketplace for WRT applications that supports multiple versions of the WRT, and close to 100 devices that support them all.

Tools for S60 Platform Development

As you can see, when it comes to developing S60 platform applications, you have many choices. The platform supports several programming languages, notably C/C++ for porting existing UNIX applications, and Java to port Java ME MIDlets. As mentioned previously, the software stack offers several runtimes that offer application development using WRT widgets, Flash, and Python.

However, the primary programming language for the platform is Symbian C++, which is integrated with Symbian's object-oriented interfaces and frameworks. It has special language idioms for embedded system support, such as a clean-up stack. On the negative side, these idioms and the language's behavior makes for a steep learning curve with the language, although this is no different than learning Objective-C for the iPhone. Additionally, over the years, the platform has accumulated hundreds of API classes with a relatively large number of methods. It can take time to sift through these when learning to write an S60 platform application. You can use any of the following development environments to build native S60 applications:

- Carbide.c++ v1.0 — v2.0: an Eclipse-based IDE and toolset
- Carbide.c++ Express v1.0 — v2.0: freely available
- Microsoft Visual Studio .NET 2003/2005
- Metrowerks Codewarrior: this will be phased out by Nokia in favor of Carbide.c++

The platform also supports Open C/C++, which implements a vendor-neutral C programming interface. Open C's runtime consists of an implementation of selected POSIX libraries, while Open C++ uses STL/Boost libraries. These libraries allow ports of Linux applications to the platform. Using the Open C SDK, the Carbide.c++ IDE can be used to write Open C/C++ programs. The SDK can also be used with GCCE and RVCT compilers.

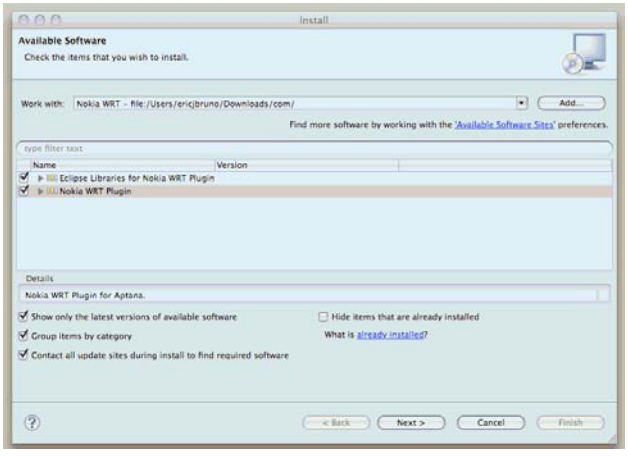


Figure 8: Installing Nokia's WRT plug-in for Eclipse.

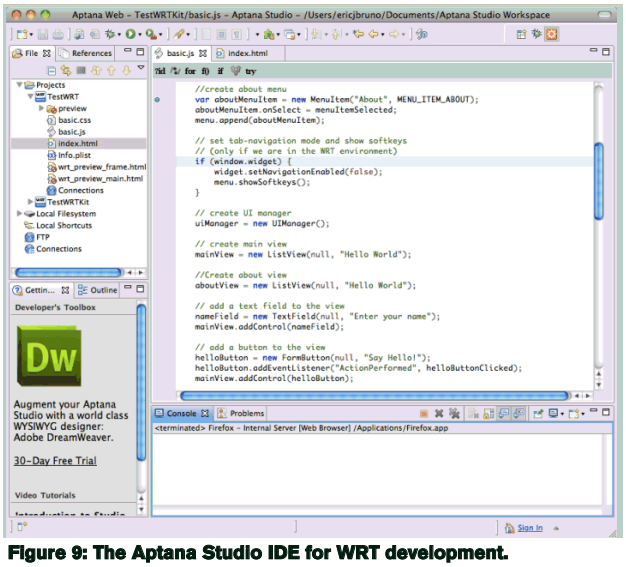


Figure 9: The Aptana Studio IDE for WRT development.

Python language support is available as an installable runtime. Python is valuable as a rapid prototyping environment, and it can be extended to function as a code wrapper that invokes S60 platform APIs. Perl development is supported through the ActivePerl software download.

WRT Development

WRT widgets can be written using a number of web authoring tools. Nokia has developed an Eclipse plug-in that works with Aptana's Studio, a development environment used to debug HTML, JavaScript, CSS, and Ajax code. Other IDEs supported are the standalone version of Aptana (which itself is based on Eclipse), Dreamweaver, and even MS Visual Studio; the choice is yours. Nokia's WRT tools plug-in page provides the tools and downloads needed to work in all of these environments.

If you already use Eclipse as we do, and prefer to stay in that environment, you can install the Eclipse/Aptana plug-in with the following steps:

1. Download Aptana or Eclipse with Aptana plug-in (and Nokia WRT plug-in)
2. To find the Aptana and Eclipse downloads and complete instructions, see http://tools.ext.nokia.com/wrt/prod/aptana/plugin/
3. Download the plug-in here: http://tools.ext.nokia.com/wrt/prod/aptana/plugin/com.nokia.wrt.2.3.0.v20091028223022.zip
4. Unzip plug-in archive to a known directory
5. In Eclipse, select the Install New Software menu option from the Help menu
   a. Clock Add, and then clock Local.
   b. Provide the path to the plug-in directory (where you unzipped it) and provide a name
6. When asked what to install, select both Nokia WRT Plugin and Eclipse Libraries for Nokia WRT Plugin (see Figure 8).

Alternatively, to avoid the potential hassle involved if you don't have the correct version of Eclipse, and cannot or do not want to upgrade, you can install the Aptana Studio IDE from http://apatana.com. The IDE, being based on Eclipse, looks and works as you'd expect from an Eclipse-based development tool (see Figure 9).

You can download device emulators, or use the built-in Firefox support to run and test your WRT applications on the desktop. There are also options to verify and deploy your WRT applications directly to a device when you reach that point.

Java Development

Java ME development can be done with the usual suspects: either Eclipse or the NetBeans IDEs. The Runtimes Layer implements the Java ME profile/configuration MIDP 2.0/CLDC 1.1. The platform also provides access to the device's capabilities to Java applications, such as the touch screen, the gesture API, location-based services, and so on.

For Java applications, all user interface manipulation on the Nokia platform can be done through the following:

- LCDUI: Liquid Crystal Display User Interface (LCDUI) is part of MIDP, and takes a screen-based approach where a single Displayable (or a UI control such as a listview, text box, and so on) is always active at a time in the application user interface.
- eSWT: Embedded Standard Widget Toolkit (eSWT) provides a very rich GUI experience for Java ME applications on the S60 platform
- Scalable 2D Vector Graphics API: a W3C XML standard for describing 2G graphics
- Mobile 3D Graphics API: real-time 3D graphics and animations, supported through OpenGL or Direct3D
- Nokia UI API: part of Qt, and an extension to the standard MIDP functionality

Java applications also have access to the S60 platform's APIs for multimedia support, security, wireless messaging (i.e. MMS), networking (Java IO, Web, Bluetooth, SIP, IAP), application launching, device sensors, the Touch UI API, and the Location API.

Application Deployment

Once you've developed and tested that killer S60 mobile app, you'll need to focus on deployment. Applications written in Symbian C++ and Open C/C++ are packaged as SIS files. Symbian SIS files contain installation metadata, program code, and resources. The smartphone's software installer reads the metadata to determine if the application requires specific hardware that might be absent on the target device, such as an accelerometer or Bluetooth. The installer can also perform a security check at this time.

For the S60 platform, the SIS file must be digitally signed. This is required by Symbian OS v9.x for the security check and trusted code mechanism to work. Security certificates can be used to permit or block application execution in the marketplace, within the enterprise to control in-house mobile applications, or via the network operator to manage smartphone features and grant access to network services.

S60 applications can be distributed over-the-air (OTA) with wireless technology such as Bluetooth, or PC connectivity software. Traditionaly, Handango has managed the wide-scale distribution of Nokia applications. A couple of years ago, Nokia announced plans to launch its Ovi Store, which sells applications, videos, games, podcasts and other content, similar to Apple's App Store. The store will be accessible by all S60 platform-based smartphones. As with other mobile provider application stores, Nokia offers developers 70 percent of the revenue from sales.

Finally, the Symbian Foundation will take on the governance of Symbian and Symbian-based platforms such as the S60 Platform,

Figure 10: The keys on a typical BlackBerry (an 8800 in this case) resemble seeds.

and move them to open source in the future. The source code for these platforms will be available to developers for free.

S60 Platform Summary

The Nokia S60 platform is currently one of the largest deployed mobile platforms in the world, but others are catching up. Therefore, the S60 as a development platform has many positives.

Here are some of the S60's pros:

- Nokia has the largest smartphone marketshare.
- The S60 platform supports a diverse set of languages and tools to develop applications (C , C++, Python, WRT, Java, Flash, Perl)
- Good enterprise support
- Excellent security
- UNIX-like OS
- Stable, real-time enabled kernel

Cons, and other things to consider:

- Non-standard Symbian C++ has steep learning curve (i.e. there are special idioms to master)
- Large number of Symbian and Qt APIs to learn: it contains hundreds of classes and thousands of member functions
- Developer tools can be costly
- Too many development options can lead to confusion

Bottom line: The S60 is a mature application environment with many features, a large install base, and wide programming language and tool support.

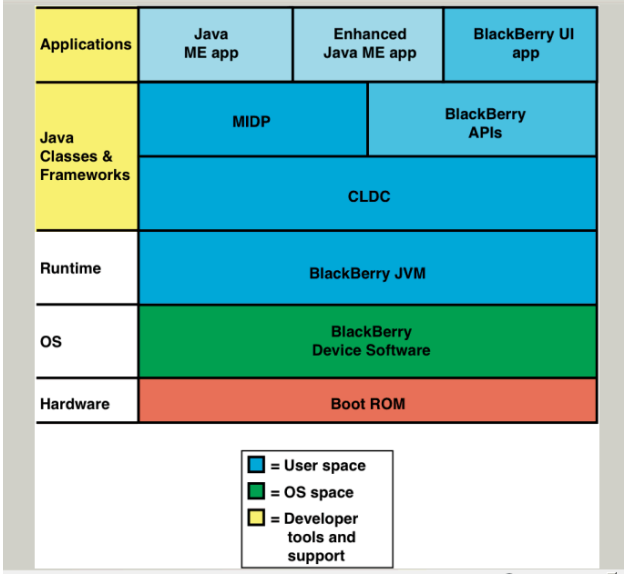


Figure 11: RIM BlackBerry software stack.

RIM Blackberry Mobile Application Development

If there is one smartphone widely recognized for its enterprise support and communication capabilities, it's Research In Motion's (RIM) line of BlackBerry smartphones. In 1999, the company introduced its first device: a wireless two-way pager; and in 2002 it launched its first BlackBerry-branded smartphone. BlackBerrys feature their trademark QWERTY-style keypad with which owners can write e-mails much more easily than with a simple numeric keypad. In fact, it was this keyboard that inspired the name "Blackberry" for the device.

Early beta reviewers of the device often said it resembled a strawberry, with the keys on the keyboard being the "seeds." However, RIM decided that strawberry wasn't an appropriate name, but BlackBerry, which matched the device's color, was available and ideal (see Figure 10 for a typical BlackBerry device).

With lots of useful apps installed out of the box, wireless support, and a server infrastructure that pushes Microsoft Exchange e-mails to the device, BlackBerry smartphones enable on-the-go workers to keep in touch from anywhere. With arguably one of the best e-mail clients around, support for MMS and SMS messages, as well as Facebook, Twitter, and other social media, BlackBerrys have been a leader of communication devices for many years.

As a long-time smartphone contender, RIM mounted a rapid response to the iPhone threat with the launch of the Storm, which is a full touch-screen version of its platform. With a wide range of devices such as the Curve (an affordable consumer device), the Pearl (with a compact form factor), the Bold (with one of the best mobile keyboards ever), the Storm (with an enhanced touch interface), and more recently the Torch (with a slide-out keyboard and smoking performance), RIM has positioned itself well into every niche in the smartphone marketplace. Some of the BlackBerry platform's features are:

- Application multitasking
- Java ME underpinnings provide memory protection and trusted code execution
- Supports Java ME applications with specialized development environment
- A touch interface
- Manages multiple e-mail accounts including:
  - SMPT
  - Exchange
  - POP3
  - IMAP
- E-mail file attachment support
- Browser support for various media types
- FIPS 140-2 compliant, and supports AES or Triple DES encryption

Like Nokia and Apple, RIM owns and builds the entire hardware and software stack for the BlackBerry smartphone. This gives them the ability to tightly integrate software features to hardware functions, and closely manage the overall user experience.

At its core, the BlackBerry's complete operating environment and its application are 100% Java ME compliant, and the platform can execute Java ME MIDlets. However, RIM's software has enhanced the capabilities of the platform with its own Java virtual machine (JVM), along with new Java classes that offer multitasking capabilities, messaging, and UI enhancements to go beyond the capabilities of Java ME. Let's take a look at the software stack (Figure 11) to see how RIM accomplished this.

Inside the BlackBerry OS

Java ME applications (MIDlets) and BlackBerry UI applications execute at the topmost Application Layer. You can also take existing Java ME code and add specific BlackBerry classes to make a hybrid Java ME application. For example, you might invoke a BlackBerry API call to select an audio output device (speakers or headphones), and then use a standard multimedia player class to play the audio content. Hybrid applications are possible as long as you don't intermix MIDP 2.0 and BlackBerry API calls that perform either screen drawing or application management.

Drilling down, the next stack layer is the Java Classes and Frameworks Layer. Java ME MIDlet programmers will be on familiar ground here, as this layer resembles the Java ME platform, which makes sense since that's what it is. Here you'll find the usual MIDP MIDlet classes that manage the UI and application lifecycle, and beneath that resides the CLDC classes that provide access to low-level resources. For BlackBerry smartphones running BlackBerry Device Software 4.0 or later, the platform's Java profile/configuration is MIDP 2.0/CLDC 1.1 compliant, otherwise it uses MIDP 1.0/CLDC 1.0.

This layer also implements useful Java Specification Request (JSR) API packages, such as JSR-75 (PIM and FileConnection services), JSR-135 (multimedia capture and playback), JSR-82 (Bluetooth support), JSR-120 (wireless messaging), and JSR-179 (location services), to name a few. All of these classes and those of the application are loaded and executed by the BlackBerry JVM; RIM licensed the rights to build and market a Java-compliant VM on its devices many years ago.

The BlackBerry API extensions in this layer enhance the platform in several ways. First, they provide UI APIs for custom menu, widgets, and screens. Next, an application class, Application, enables an application to persist and continue to run in the background, unlike the MIDP Midlet class, which requires that the application terminate when it is closed. Other APIs handle the set-up and tear-down of network sessions, and manage I/O to servers. Finally, these APIs provide hooks into the device's camera, media player, and web browser. An application can be written using only CLDC and BlackBerry APIs. Such an application has access to all of the device's features including, for example, Bluetooth, the accelerometer, and the touch screen on the Storm. It can also run concurrently with other applications. Finally, the application can be launched when the BlackBerry starts up and then run in the background. The catch to writing an application that uses BlackBerry API extensions is that it ties the application to this smartphone. However, this is no worse than using the unique Java classes found in Google's Android.

The BlackBerry Device Software Layer implements a low-level multitasking, multithreaded operating system. It uses listener threads to monitor for specific device events. For example, these listener threads manage push support for e-mail and messages. The BlackBerry Device Software allows an IT manager to configure applications and disable certain smartphone features, and to blank a lost device remotely.

Programming BlackBerry Devices

You have choices when developing applications for BlackBerry devices. The premier platform and environment is Java ME, but RIM now supports application development with Ajax-like web standards known as BlackBerry web applications, as well as applications that contain mostly graphics, animations, and other media content known as BlackBerry Theme applications. Let's examine all three of these environments now.

BlackBerry Java Development

With the BlackBerry platform being a Java ME platform, you can use a number of Java development tools, such as Eclipse and NetBeans, to write applications for BlackBerry smartphones. Officially, Windows is the only supported development environment, but the tools work on Linux and Mac OS X computers as well.

RIM also provides a BlackBerry Java Development Environment (JDE), which is a set of standalone tools. The JDE consists of an IDE that allows you to write, build, and test Java ME and BlackBerry smartphone applications at the source-code level. The JDE has phone simulators that allow you to view and interact with your application. It also comes with a Signing Authority Tool to sign your applications, which is required to access the BlackBerry APIs. To thoroughly test your custom BlackBerry applications, RIM offers simulator software for features such as e-mail and communicating with BlackBerry Enterprise Servers.

For specialized client/server applications, RIM provides rapid application development (RAD) tools to decrease the time you spend in development and testing. With these tools, you can prototype services for UI frameworks, data management, and client-server I/O. The RAD tools are available as a plug-in for Microsoft Visual Studio, or as a standalone Mobile Development System (MDS) for use inside of Eclipse.

BlackBerry Web Development

For web development, you can use standards-based HTML, JavaScript, and CSS. The BlackBerry web browser is web 2.0 standards-based, and on BlackBerry Device Software 4.6 or later, the browser adheres to AJAX, JavaScript 1.5 – 1.8, CSS 2.1 – 3.0, DOM L2, and HTM4.01 to HTML5 standards. The platform also supports a messaging arrangement termed "Web Signals". When an update is made to a website, its server can use RIM's push infrastructure to

send a notification to a BlackBerry device. The pushed message includes a hyperlink back to the updated content.

Your applications can deeply integrate with RIM's push communication infrastructure, which supports sending messages to browser-based applications via many protocols. The Web API supports various back-end server technologies (i.e. Java Servlets in popular web and application servers), and is optimized to help reduce round-trip message frequency and sizes.

The BlackBerry Widget SDK 1.0 consists of a full application development environment (with its own debugger, simulator, HTTP proxy, and application packager), putting it almost on par with Java ME-based BlackBerry applications. BlackBerry Widget applications can be developed with Eclipse or Microsoft Visual Studio, along with the BlackBerry Widget SDK and the Java SE 6 Development Kit (JDK 6).

Applications can be distributed over the air (OTA), via the Web, or through a web loader application on a PC with a BlackBerry smartphone tethered to the computer with a USB cable. RIM supports commercial application distribution via Handango, and the newer BlackBerry App World storefront. Developers retain 80% of the revenue made from sales of the application. See the External References section for more on the App World store and other BlackBerry resources.

BlackBerry Theme Development

BlackBerry Theme applications are developed using the Theme Studio, previously known as the Plazmic Content Developers Kit. This allows you to create BlackBerry smartphone themes —as the name implies —as well as media-rich applications on SVG-based content. The resulting applications can be seamlessly integrated into the BlackBerry device's overall user interface, using existing Adobe Flash and Photoshop files for the design.

The Future: BlackBerry 6

RIM unveiled its latest BlackBerry device OS, version 6.0, this summer, along with the 6800 Torch. This latest OS is clearly a continued response to the competitive threats from both the iPhone and Android-based devices. While keeping the spirit of the traditional BlackBerry experience, RIM has packed quite a punch with v6.0, including the following features:

- A unique UI to integrate social networking feeds and RSS feeds, all in one view
- An updated WebKit-based browser with faster page loading, and easy-to-use tabs to switch from one page to another quickly
- Support for HTML5.0, CSS 3.0, and JavaScript 1.8 (passes Acid3 100/100 tests)
- An innovative universal search feature that comprehensively searches throughout your device, the Web, and networked applications to find what you request
- Enhanced multimedia capabilities with new applications such as built-in YouTube access
- Simplified setup and personalization — something we thought was confusing in the previous versions, but is now much improved
- Personalization extends to the entire UI look and feel
- A simplified, smooth UI with lots of eye candy, but with better organization
- An improved device title bar packed with information about connectivity, battery, and application-specific entries
- Inertial touch-based scrolling with enhanced support for gestures
- Third-party application API that allows deep integration with the device's native capabilities (camera, touch screen, location services, and so on)
- A new Animation API with support for OpenEL ES 1.1 3D and OpenVG 1.1, both with hardware acceleration on devices that support it
- New enterprise features such as:
  - Advertising Service SDK
  - Payment Service SDK
  - 1D and 2D barcode reading support
- Built-in SQLite database for application data

RIM Acquisition of QNX

In April 2010, RIM announced its intent to acquire QNX from Harman International. The QNX real-time OS represents a growth opportunity for RIM to expand into device markets beyond just smartphone devices. This includes automotive and peripherals, perhaps integrated with the entire BlackBerry experience.

"In addition to our interests in expanding the opportunities for QNX in the automotive sector and other markets, we believe the planned acquisition of QNX will also bring other value to RIM in terms of supporting certain unannounced product plans for intelligent peripherals, adding valuable intellectual property to RIM's portfolio and providing long-term synergies for the companies based on the significant and complementary OS expertise that exists within the RIM and QNX teams today," said Mike Lazaridis, RIM's cofounder and president.

BlackBerry Summary

The BlackBerry offers a truly unique smartphone experience that has kept a loyal customer base for over a decade. Offering a comprehensive communication platform, here are some of the device's pros:

- Proven, secure, reliable, and global
- Arguably the best keyboards in the industry
- Best of breed Java ME platform for enterprise communications
- Infrastructure for push e-mail and messaging in place and proven
- Enterprise security and capability throughout
- Platform supports third-party application multitasking
- Java-based platform, but with proprietary BlackBerry's API extensions

Cons:

- To use the platform to its best advantage, you must use RIM's own APIs and classes
- BlackBerry applications won't necessarily run on other Java ME devices due to the BlackBerry API extensions
- Late in offering touch and gesture-based UI

The MeeGo Mobile Platform

MeeGo is a Linux-based platform built especially for netbooks with limited computing resources and screen space, handheld devices, in-car infotainment systems, and soon, tablet computers as well. It's the result of Intel and Nokia's efforts to merge their Moblin and Maemo Linux offerings, respectively, and is fully supported and hosted by the Linux Foundation. Recently, the MeeGo partnership has grown with Novel's announcement that it will soon support and ship its own SUSE MeeGo distribution. The goal is to see MeeGo installed on all sorts of devices (from a variety of vendors) over the coming year.

Although ARM processors are supported, the main target for MeeGo are Intel Atom-based netbooks and devices, which the partners claim Windows 7 is not well suited for. In fact, before embarking on the research project to prepare for this article, we purchased an Asus EeePC with an Intel Atom 270 CPU, 1GB of RAM, and Windows 7. We've used netbooks with different versions of Windows in the past, and the EeePC performed to expectation, and generally what you would expect from a smaller, less expensive, netbook computer. Overall, Asus makes a nice product, but Windows 7 can be a bit taxing for its limited resources.

For the next step, we downloaded and installed MeeGo v1.0 for Netbooks (more on this later) on the EeePC — goodbye Windows. The result was a pleasant surprise in both performance and usability. In the past, we had read all about MeeGo, viewed the screenshots on the website, and had an idea of what to expect from a typical Linux distribution. The reality of MeeGo far exceeded our expectations. Not only did the EeePC performance seem to double with MeeGo — everything on the netbook is now quite snappy — the user interface paradigm is quite natural and comfortable.

The research project continued by using the MeeGo netbook on train rides to write articles and blogs, and to read PDFs and other documents Eric had saved on it. All in all, the smaller size is perfect for commuting. Additionally, being a Linux distribution at its core, you can install all of the necessary software you need easily, including Open Office, Java, and the GNU toolset. Combined with the built-in support for e-mail, Twitter, Facebook, appointment tracking, reminders, and even a media player, the MeeGo/EeePC experience is as enjoyable as an iPad. And as a bonus, it supports Flash! Next, let's take a look at the MeeGo editions available, how to install and configure it to suit your needs, and what it's like to use it regularly.

Figure 12: MeeGo integrates applications, a calendar, web pages, and social network updates on the home screen.

Inside the MeeGo Editions

MeeGo refers to its various editions as User Experiences, or simply UX. Each edition shares the same MeeGo/Linux core deliverable, with a different UX layered on top. For instance, here is the ever-growing list of UX editions you can download and install:

- MeeGo v1.0 for Netbooks UX
- MeeGo Handset Day1 Developer Preview UX
- MeeGo v1.0 for In-Vehicle Infotainment (IVI) UX
- Upcoming: MeeGo Tablet UX

The MeeGo website refers to specialized distributions — targeted at original device manufactures (ODMs) — such as MeeGo Connected TV and the MeeGo Media Phone. The Connected TV edition is an OS/software stack optimized for devices you'd likely find in your living room or media center. This includes Blu-ray players, TV set-top boxes, and digital television sets. The idea is to enable the integration of multi-media players, the Internet, and computing devices, all within your living room.

The MeeGo Media Phone edition is an answer to the realization that telephones have gone well beyond simple voice communication devices (just look at the iPhone or Android devices and you'll see this is true); but what about the desktop phone? MeeGo aims to turn desktop phones into entertainment systems and Internet access points, just as smartphones have for mobile phones. Built-in applications that support high-definition video conferencing over the Web represent the practicality of this integration.

Although these specialized editions are currently only available to ODMs, let's take a look at the other editions that are available to consumers and developers to download and use today.

MeeGo v1.0 for Netbooks

MeeGo for Netbooks is available to download in production form, and can be installed on virtually any Intel Atom-based netbook. The following netbooks have been approved and tested by MeeGo (check http://MeeGo.com for updates):

- Asus EeePC (901, 1000H, 1005HA, 1008HA, 1005PE)
- Asus Eeetop (ET1602)
- Dell mini10v, Inspiron Mini 1012
- Acer Aspire One D250, AO532-21S, Revo GN40, Aspire 5740-6025
- Lenovo S10
- MSI U130, AE1900
- HP mini 210-1044
- Toshiba NB302

Figure 13; The MeeGo Handset UX provides a familiar mobile user experience.

Users of Intel Moblin will find the MeeGo Netbook UX very familiar, if not quite a bit more refined. It's also the first mainstream Linux distribution to support the Btrfs Linux file system out of the box. Additionally, Nokia's Qt framework is included, which allows both Qt applications and GTK applications to run natively on the platform.

Overall, the Netbook UX provides a very rich user experience, designed to run well on the small screens of netbook computers. MeeGo displays tasks and calendar appointments synchronized with Google, a list of recently opened web pages, another list of your favorite applications, and real-time social networking updates all on the home screen. Placing e-mail, Twitter, Facebook, and other web content on the home screen where you can easily access and update it means less time spent switching between applications and URLs (see Figure 12).

Google Chromium (with the latest in Ajax and HTML5 support) is installed by default, along with Evolution for e-mail, and an easily accessible media player for music and video support. (Note that there is an alternative netbook download that includesGoogle's Chrome browser in place of Chromium.) All of this is available in many languages, and with the best performance of any OS we've tested on a netbook to date.

As far as connectivity and interoperability goes, my MeeGo EeePC connected seamlessly and quickly to my wireless network via the easily accessible network connection screen. We were able to connect and share files with other computers running Windows, Mac OS X, and Ubuntu Linux without issue. We also tested USB thumb drives formatted for OS X and Windows, and had no problems reading or writing data on them. I'll go into more details on using MeeGo on a netbook later in this article.

MeeGo for Handsets

Recently, MeeGo released a preview of the MeeGo Handset edition source code with binary kickstart downloads available for Intel Moorestown-based devices, as well as Nokia's ARM-based N900 handheld Internet tablet (for an article outlining an experience with the previous generation of this device — the N810 — see http://www.drdobbs.com/java/208801979).

Of all the MeeGo editions, the Handset UX represents the Moblin/Maemo merger the best, as it combines the look-and-feel and goals of both distributions. At the foundation is the MeeGo core, with the Nokia Qt framework, MeeGo Touch UI framework, core MeeGo applications, and specialized mobile applications built on top. The status bar is changed to appear as you would expect on a mobile device: it includes cell strength, Bluetooth state, and battery status, along with the familiar date and time.

The home screen contains a grid of icons for applications such as a dialer, SMS messenger, web browser, calendar, contacts, maps, video, and so on. As with Android or iPhone, the UX is geared towards Internet connectivity with an excellent mobile browsing experience, and online media support. The screen layouts are clean and arranged well, with support for both portrait and landscape modes (see Figure 13).

The MeeGo Handset edition includes advance previews of the new multi-touch and gesture support that's been added to the platform. Also new are advanced geo-location services, including Intel's Geoclue, which uses multiple location sources to provide postal information for your current location. There are also APIs available for all of this technology so you can build customer applications around them.

You can experience all of this for yourself by either downloading a preview image already built, the source code to build yourself, or a video of the UX running on a handset, all available on the MeeGo website under the Handset link.

MeeGo for Tablet Devices

Although very few details have yet been provided, Intel and Nokia are planning make a preview version of the MeeGo Tablet UX available to developers in October 2010. This UX will be similar to the Handset UX, with the same framework for multi-touch and gesture support, but optimized for devices with larger screens. For instance, sneak peak videos of the Tablet UX show a grid layout of the home screen, where columns can be scrolled up and down to show various content, and the rows can be scrolled back and forth to reveal other categories of content.

Overall, the UX will adjust to the way you work, keeping a comprehensive history of the applications and content you frequently use, while still keeping important system information (battery charge, connection status, external devices, and so on) at your fingertips at all times.

There will be a high level of familiarity to both MeeGo netbook and handset users, since the common application set, and touch and gesture APIs, will remain the same. Working with and developing for the Tablet UX is meant to be a natural extension to MeeGo for other platforms.

MeeGo for In-Vehicle Infotainment (IVI) Systems

In April of this year, Intel and Nokia announced the support of MeeGo for in-vehicle infotainment systems by the GENIVI alliance, which is a consortium that consists of BMW, GM, Delphi, Intel, Wind River, Peugot, and many others. Having already standardized on Moblin and Qt, GENIVI officially announced their roadmap recently, which includes moving to MeeGo IVI.

Although no official screenshots or demonstration videos have been made, the following details are known:

- The home screen will consist of a taskbar located on the right or left
- The GUI will conform to Automotive Center Console HMI requirements
- Navigation is supported through any combination of touch screen, mouse, or a scroll device
- A navigation system will be included
- Bluetooth, hands-free dialing will be included
- Audio management will be included
- Data connectivity will be an option

As for hardware support, the target systems are Intel Atom based, with reference platforms including the Intel Russellville eMenlow system and Intel Crown Bay Tunnel Creek system.

MeeGo Licensing

Meeting all of the technical requirements for a large market segment or vertical is only half the battle; licensing is just as important for commercial success. This is especially true in embedded systems deployment, such as with handsets and infotainment systems, where licensing viral effects can infect derivative works.

As a result, Intel and Nokia are trying to maintain an open-source development and adoption model for MeeGo, while also trying to simplify the process of building derivative works around it. The MeeGo License Model breaks the software out into two pieces: the core OS component set, and the user experience component set built on top. Overall, the goal is to not impose any further limitations on top of the licensing models of MeeGo's constituent components.

For the core OS, in summary, the licensing policy is that all components must be under OSI-compatible licenses, with the use

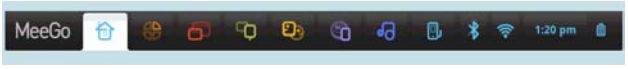


Figure 14: The MeeGo taskbar along the top of the screen.

Figure 15: Applications are listed according to category to make them easier to find.

of GPL v2 licensing strongly encouraged and favored. Further, each component's licensing must support linking to proprietary extensions. For the UX, components must be under a permissive OSI-compatible (BSD-style) license, which does not mandate code modifications to be released as open source.

Where possible, licensing efforts are made to guard against fragmentation for any particular MeeGo UX edition, and exceptions to the above licensing requirements have been made for a few individual components. Therefore, you should read through the licensing on the MeeGo site to be sure you're in agreement with its requirements.

Installing MeeGo for Netbooks

We began my research project with MeeGo by downloading the MeeGo Netbook v1.0 binary image onto a Mac OS X computer. My intention was to install the image onto an Asus EeePC from a USB thumb drive. To do this, we followed these steps after downloading:

1. Insert the thumb drive and erase its contents
2. Unmount the thumb drive in OS X by running the command:
   diskutil unmountDisk /dev/diskN
3. Copy the bootable image:
   sudo dd if=/path/to/downloaded.img of=/dev/diskN bs=1m
4. Eject the thumb drive:
   diskutil eject /dev/diskN
5. Place the thumb drive into the netbook's USB port, and set the BIOS to boot up first in its book order
6. Follow the MeeGo installation instructions, choosing to either wipe your hard drive clean, or repartition to support dual booting with the existing OS.

For Windows, the steps are similar, but you'll need to use a software package that supports disk image (ISO) burning, such as Launchpad's Image Writer for Windows, or Astonsoft's DeepBurner.

As for hardware requirements, any Intel Atom-based based netbook should work, but MeeGo does specify some tested models (listed earlier in this article and on the MeeGo website). Let's take a look at what it's like to use MeeGo on an everyday basis.

Using MeeGo on a Netbook

The MeeGo main screen (see Figure 12 again) organizes itself to list all of your frequently used items, including web pages, documents, applications, and social network updates. The toolbar along the top (see Figure 3), which hides itself when you're viewing anything other than the home screen, contains links to other parts of your system. These are, in the order they appear from left to right:

- Home: The home screen
- Applications: The applications installed on your system
- Status: Social network status updates (i.e. Twitter, Facebook, and so on)
- People: Instant messaging client
- Internet: Browser, bookmarks, and browsing history
- Media: A multimedia player for music and videos
- Devices: Settings for your netbook, and access to USB-connected devices
- Bluetooth: Bluetooth settings and access to devices connected or available
- Networks: Wired and wireless network settings, connections, and available networks to connect to
- Date and time: Access the clock, calendar, and tasks
- Charge: The power and charging status of your netbook

The toolbar makes it quick and easy to navigate around your MeeGo system, while remaining unobtrusive since it hides itself when you're working with an application. You can customize the toolbar to add and remove tabs as you wish. Not shown here are additional icons for a cut-and-paste pasteboard tool, gadgets (for Mac OS X like widgets), and others.

Configure Favorite Applications

The Home screen contains a list of favorite applications, with four initial applications listed in the panel by default: Media Player, MeeGo Help, Mail, and a browser. You can remove, replace, or add applications to this list as you please. To do so, click on the Applications icon in the toolbar to view the categorized list of installed applications (Figure 15).

When you find an application that you would like to add to the Favorites list on the Home screen, simply click on its pin icon to "pin it" to the Home screen. Subsequently, you need to "un-pin" an application in this view to remove it from the Favorites list.

Installing New Applications

Being a Linux-based system, to install applications in MeeGo you simply download and install the appropriate Linux packages. For instance, we installed OpenOffice for Linux by downloading the Linux RPM package and following the installation procedure. For Java, we downloaded the self-extracting executable Linux package, added executable permissions to the file, and executed it as a shell script.

Afterwards, both OpenOffice and Java worked as expected from the command-line. However, in the case of OpenOffice, we wanted to add an icon to the Applications screen to launch it from the GUI. This proved to be a bit more challenging, but after figuring it out (and getting some help from Intel) the steps are quite simple. First, know that each application listed in the Applications screen has a corresponding *<appname>.desktop* file in the /usr/share/applications folder. To add a new one, follow these steps:

1. Start with an existing .desktop file and copy it
2. Rename the copied file to something appropriate for your new

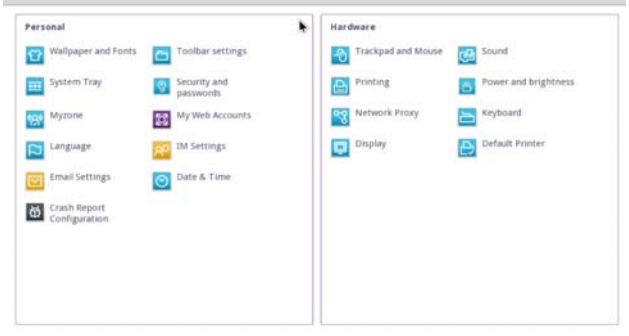


Figure 17: MeeGo settings, for both software and hardware, are configured here.

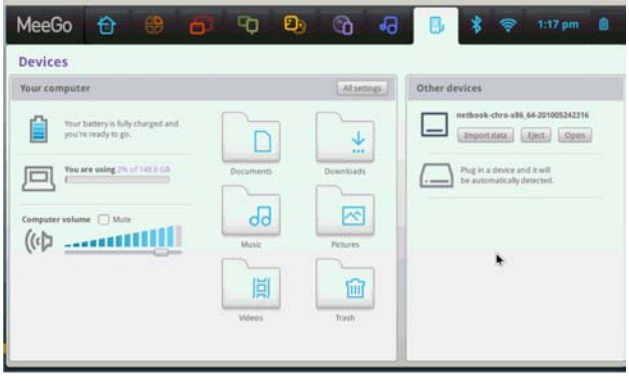


Figure 16: The Devices screen gives you access to external devices and system settings.

Figure 18: Zones allows you to organize and switch between running applications.

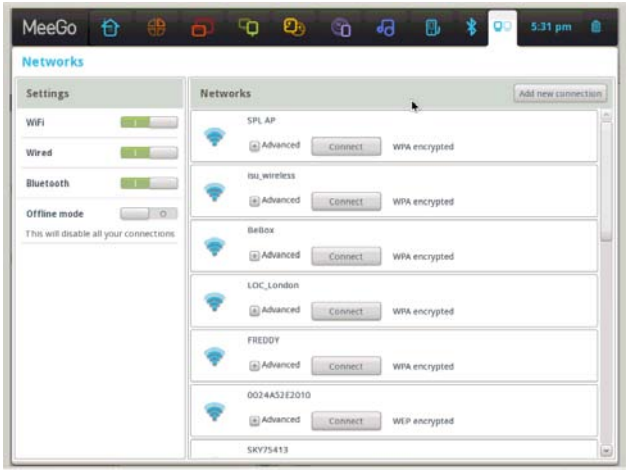


Figure 19: The Networks screen makes it easy to join a network.

application; i.e. openoffice.desktop

3. Modify the following values accordingly:
   a. Name: the name you'd like to see in the Applications screen; i.e. OpenOffice
   b. Comment: any you'd like to see when the mouse hovers over the icon
   c. Icon: the path to a picture file for the application's icon
   d. Categories: make sure the list of category names includes the name of the category you'd like to see the icon listed under; i.e. Office. Other options are Accessories, Games, Internet, Media, and System Tools.
   e. Exec: the command-line string to use to launch the application; i.e. /home/ebruno/opt/openoffice.org3/program/soffice

If all goes well, you should see the application's icon listed in the category you chose, with icon you specified, and it will launch when clicked on.

E-Mail and Social Networking

You can configure your default e-mail account(s) shown on the Home screen by selecting Email Settings in the Settings page. To do this, go to the Devices screen (via the MeeGo toolbar), and then click on the All Settings button (see Figure 16). In a nutshell, the Devices screen allows you to access external devices and common folders, and shows system statistics such as disk space usage and speaker volume.

We admit that accessing system settings through the Devices screen may not be intuitive at first, but once you know it's there, you should remember it. In the Settings screen, you'll see entries to set your wallpaper, fonts, the toolbar, date and time, printer, keyboard, display, network, and so on (see Figure 17 for all of them).

The entry we're interested in now is Email Settings. You can configure more than one e-mail account, which will be summarized on the Home screen, by following the wizard steps presented. Supported protocols include IMAP, IMAP+, POP, Novell Groupwise, USENET, UNIX mail, and various local directory and delivery options. Configuring your instant messaging accounts can be accomplished via the IM Settings entry in the Settings panel as well.

Configuring social network accounts, such as Twitter, is done through the Status screen, available from the Status tab on the toolbar. From here, click on the Add New Web Account button, and configure your social network account settings; i.e. Twitter. You will see your Twitter updates displayed both in this tab and the Home screen. However, you must visit the Status screen to update your social network status; i.e. enter a new tweet.

Zones and Application Switching

You can switch between multiple running applications by pressing the familiar Alt-TAB key combination, but you can further group applications using MeeGo's zones concept. First, click on the Zones tab within the task bar to view running applications, each in their own zone by default (see Figure 18).

When viewing zones, you can use the mouse to select an application thumbnail (such as OpenOffice on the right) and drag it to another zone. This allows you to easily group and switch between running applications without wasting too much time finding it in a list.

Network Configuration

The Networks screen, available from the toolbar, allows you to easily discover and configure wireless networks, as well as wired Ethernet networks (see Figure 19).

In this panel, you'll see all available networks listed by type (WiFi, Wired, and Bluetooth), where you can select and enter credentials to join. Selecting Offline mode from the left will turn off all of your connections and wireless devices in case you're on an airplane or you need to conserve power.

Summary

Hopefully after reading about and working with MeeGo, you'll gain an appreciation for what Intel, Nokia, and partners are attempting to achieve. Additionally, MeeGo offers an excellent opportunity for developers targeting the netbook market, which is growing due to the convenient size and price of the devices available. The power of Intel's Atom processor, the growing number and type of devices built around it, and the Intel application store (AppUp Center) should make this platform a viable market for custom applications for many years to come.

— Tom Thompson is the head of Proactive Support for embedded products at Freescale Semiconductor. He can be reached at tom_thompson@lycos.com. The views stated in this article are the author's, and don't necessarily represent Freescale's positions, strategies, or opinions. Eric J. Bruno is a contributing editor at Dr. Dobb's Journal. *He can be contacted at www.ericbruno.com.*

Return to Table of Contents

Developing a Silverlight UI for Windows Phone 7

How to build a UI from scratch

by Gaston Hillar

Silverlight for Windows Phone, the application development platform for Windows Phone 7 Series, supports core Silverlight features while providing access to the phone's unique capabilities through managed .NET code. Developers with some experience in XAML and managed .NET code will be able to use the new developer tools to create Windows Phone 7 Series applications. In this article, I show how to build a UI from scratch using the new development tools, SDKs, and the mobile device emulator, assuming that you are familiar with Silverlight and Visual Studio 2010 basics. First, you must download and run the Silverlight for Windows Phone Developer Tools installer. These Developer Tools are available in their Beta version, and therefore, it is convenient to download the installer by clicking the Download button in the Silverlight for Windows Phone page at http://www.silverlight.net/getstarted/devices/windowsphone/. You will always find the latest version available in the link included in this page.

Once you finish the installation process, you will notice a new folder in Start -> Programs, Microsoft Visual Studio 2010 Express. If you had any version of Visual Studio 2010 installed, you don't need to start Microsoft Visual Studio 2010 Express for Windows Phone, included in the Visual Studio 2010 Express folder. You can develop Silverlight for Windows Phone applications in your Visual Studio 2010 version that was installed before running the installer for the new tools.

Start Visual Studio 2010, select File -> New -> Project and the new "Silverlight for Windows Phone" item will appear within the Installed Templates list for Visual C#. Figure 1 shows the three types of projects that you can create targeting Silverlight for Windows Phone:

- Windows Phone Application. An application without navigation support.
- Windows Phone List Application. An application with navigation support.
- Windows Phone Class Library. A class library that you can use in other applications.

If you select "XNA Game Studio 4.0" within the Installed Templates list for Visual C#, you will notice that there are new project types related to Windows Phone. In fact, Silverlight for Windows Phone can use the XNA Framework features for audio capture and playback, access the media library, and Xbox LIVE.

Introducing Silverlight for Windows Phone

When you create a new Windows Phone application, the new solution includes the Silverlight MainPage.xaml page. However, the design view for this page shows a preview of the controls that compose the user interface within a Windows Phone 7 screen, as in Figure 2. You can drag and drop controls to the design surface and check the layout preview for a Windows Phone 7 screen. As happens when working with Silverlight in Visual Studio 2010, if you make changes to the XAML code, these changes will be reflected on the design surface that emulates a Windows Phone 7 screen.

According to the information provided in the design resources, all Windows Phone 7 devices will have WVGA screens at 800 x 480 pixel resolution, no matter the screen size. This single resolution makes it simpler to design the UI. The great drawback is that you cannot emulate multitouch gestures with a mouse or a touchpad in your developer workstation.

However, if you don't have a multitouch monitor, you still have a chance to test some multitouch gestures without having to deploy the project to the phone. There is an interesting project on CodePlex, Multi-Touch Vista (http://multitouchvista.codeplex.com), that allows you to work with multiple mice to emulate two fingers on the screen and their multitouch gestures. In fact, the latest version of Multi-

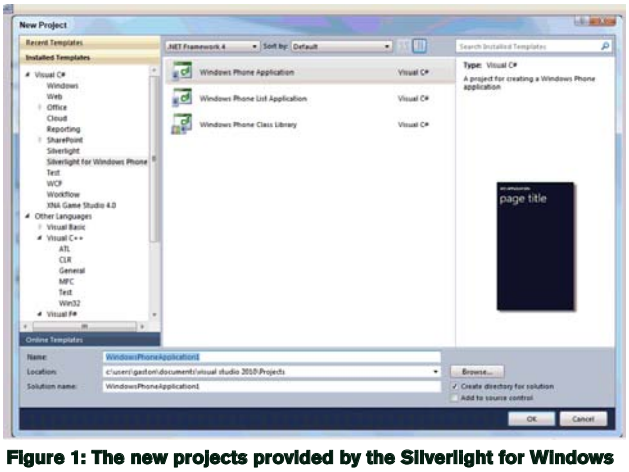


Figure 1: The new projects provided by the Silverlight for Windows Phone template.

Touch Vista provides a Windows 7 compatible driver that enables multiple mice and is compatible with the Windows Phone 7 emulator. For example, you can use a laptop's touchpad as one of the pointers and a USB mouse connected to the same laptop as the second pointer. (If you're interested in working with Multi-Touch Vista, you can read an excellent step-by-step tutorial written by Michael Sync at http://michaelsync.net/2010/04/06/step-by-step-tutorial-installing-multi-touch-simulator-for-silverlight-phone-7. This tutorial explains how to install and configure the driver to work with the Windows Phone 7 emulator.)

The structure of a Windows Phone application project is similar to a classic Silverlight application project. However, Windows Phone adds some proprietary references and files. Figure 3 shows the project structure for a sample WPBusinessApp project. This is the default structure that Visual Studio 2010 creates for a Windows Phone application.

You will notice the WMAppManifest.xml file within the Properties folder. This XML file defines the resources and capabilities of the application from the point of view of the operating system. The following code snippet shows the default contents for this file, considering that the project's name is *WPBusinessApp*:

```
<?xml version="1.0" encoding="utf-8"?>

<Deployment
xmlns="http://schemas.microsoft.com/windowsphone/2009/deployment"
AppPlatformVersion="7.0">

  <App xmlns="" ProductID="{41bdea14-687b-4815-93b3-a51759f18a09}"
Title="WPBusinessApp" RuntimeType="Silverlight" Version="1.0.0.0"
Genre="apps.normal"  Author="WPBusinessApp author"
Description="Sample description" Publisher="WPBusinessApp">

    <IconPath IsRelative="true"
IsResource="false">ApplicationIcon.png</IconPath>

    <Capabilities>

      <Capability Name="ID_CAP_NETWORKING" />

      <Capability Name="ID_CAP_LOCATION" />

      <Capability Name="ID_CAP_SENSORS" />

      <Capability Name="ID_CAP_MICROPHONE" />

      <Capability Name="ID_CAP_MEDIALIB" />

      <Capability Name="ID_CAP_GAMERSERVICES" />

      <Capability Name="ID_CAP_PHONEDIALER" />

      <Capability Name="ID_CAP_PUSH_NOTIFICATION" />

      <Capability Name="ID_CAP_WEBBROWSERCOMPONENT" />

    </Capabilities>

    <Tasks>

      <DefaultTask  Name ="_default"
NavigationPage="MainPage.xaml"/>

    </Tasks>

    <Tokens>

      <PrimaryToken TokenID="WPBusinessAppToken"
TaskName="_default">

        <TemplateType5>

          <BackgroundImageURI IsRelative="true"
IsResource="false">Background.png</BackgroundImageURI>

          <Count>0</Count>

          <Title>WPBusinessApp</Title>

        </TemplateType5>

      </PrimaryToken>

    </Tokens>

  </App>

</Deployment>
```

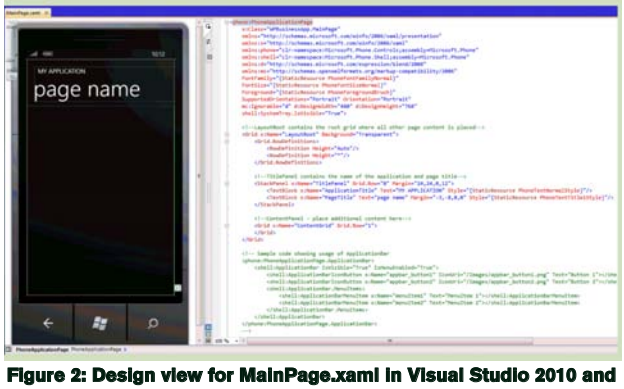


Figure 2: Design view for MainPage.xaml in Visual Studio 2010 and its XAML code on the right side.

The WMAppManifest.xml file includes the list of capabilities that the application needs from the phone in *<Capabilities>*. By default, the manifest requests all the capabilities. However, you should remove the unnecessary ones in order to create a more secure application. If you try to use a feature that corresponds to a capability that the application doesn't request in the manifest, you

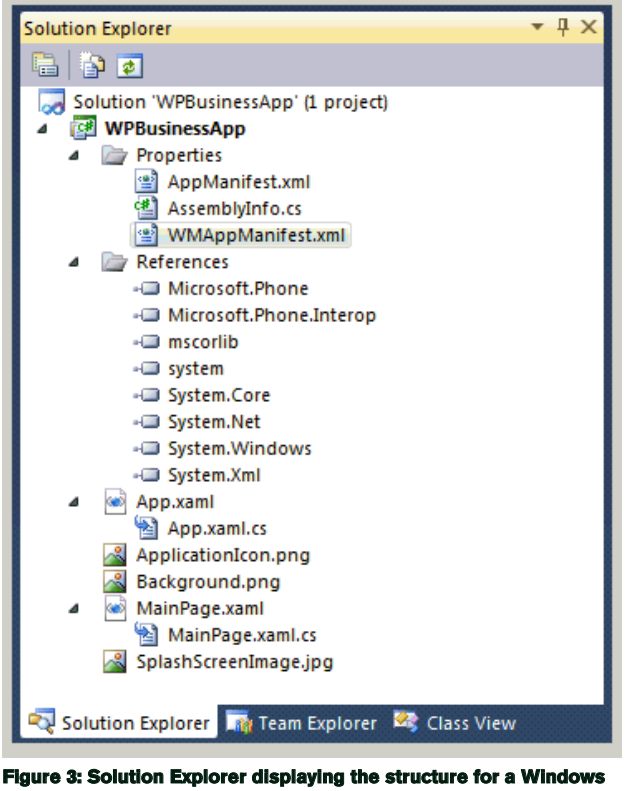


Figure 3: Solution Explorer displaying the structure for a Windows Phone Application.

Windows Phone

Set your creativity in motion

Silverlight

XNA Game Studio

Visual Studio

7

Download the free tools

The experience is elegant. The platform is powerful. One free download gives you the tools to build new apps and games, or to tweak existing Silverlight and XNA apps for Windows Phone 7. Discover the possibilities today.

Download the free tools now!

will get an *UnauthorizedAccessException* exception. If the application is installed, it will have the capabilities that it requests.

The names for the capabilities are very easy to understand. For example, *ID_CAP_LOCATION* means that you need to access location services included in the *System.Device.Location* namespace. However, some capabilities are related to many namespaces and classes. Jaime Rodriguez wrote a very interesting post about the Windows phone capabilities security model that includes detailed information about the namespaces related to each capability at http://blogs.msdn.com/b/jaimer/archive/2010/04/30/windows-phone-capabilities-security-model.aspx.

The WMAppManifest.xml file also defines the icon, the background image, and the title for the application. Luckily, you can specify icons as PNG bitmap files. The default icon is *ApplicationIcon.png* and the default background is *Backgroud.png*. You can edit XML code or you can change the values for these properties in the Application page in the project's properties, as in Figure 4.

By default, the project includes two Windows Phone related references, *Microsoft.Phone* and *Microsoft.Phone.Interop*. *Microsoft.Phone* provides access to *Microsoft.Phone.Controls* and *Microsoft.Phone.Shell*. If you have to work with sensors, you must add *Microsoft.Devices.Sensors* as a new reference.

Understanding the Code-Behind File for App.xaml and the Splash Screen

If you have some experience with Silverlight and C#, you will be familiar with App.xaml and its code-behind file, App.xaml.cs. The C# code adds some phone-specific initialization code. The *App* class provides easy access to the root frame with the public *RootFrame* property:

```
public PhoneApplicationFrame RootFrame { get; private set; }

PhoneApplicationFrame is
MicrosoftPhone.Controls.PhoneApplicationFrame. App.xaml.cs uses
the Microsoft.Phone.Controls and Microsoft.Phone.Shell
namespaces.
```

The *App* class constructor calls the *IntializePhoneApplication* method, which adds phone-specific initialization code that displays a splash screen. This code snippet shows the code for this method with the classic Silverlight initialization and the new phone-specific line:

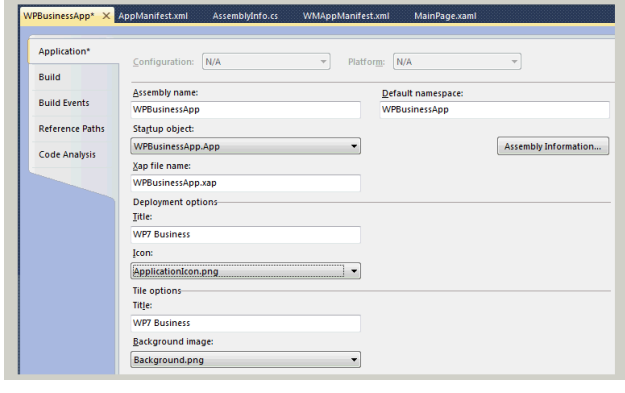


Figure 4: Project's properties displaying the Application page.

```
public App()

{

    // Global handler for uncaught exceptions.

    // Note that exceptions thrown by ApplicationBarItem.Click
    // will not get caught here.

    UnhandledException += Application_UnhandledException;


    // Standard Silverlight initialization

    InitializeComponent();


    // Phone-specific initialization

    InitializePhoneApplication();

}
```

The following lines show the code for the *InitializePhoneApplication* and *CompleteInitializePhoneApplication* methods. The *InitializePhoneApplication* method creates the new frame and keeps the splash screen visible until the application is ready to render. The *CompleteInitializePhoneApplication* method sets the new frame as *RootVisual* because it is attached as the event handler for *RootFrame.Navigated*. Don't make changes to these methods.

```
private bool phoneApplicationInitialized = false;

// Do not add any additional code to this method

private void InitializePhoneApplication()

{

    if (phoneApplicationInitialized)

        return;


    // Create the frame but don't set it as RootVisual yet;
    // this allows the splash screen to remain active until the
    // application is ready to render.

    RootFrame = new PhoneApplicationFrame();

    RootFrame.Navigated += CompleteInitializePhoneApplication;


    // Handle navigation failures

    RootFrame.NavigationFailed += RootFrame_NavigationFailed;


    // Ensure we don't initialize again

    phoneApplicationInitialized = true;

}

// Do not add any additional code to this method

private void CompleteInitializePhoneApplication(object sender,
NavigationEventArgs e)

{

    // Set the root visual to allow the application to render

    if (RootVisual != RootFrame)

        RootVisual = RootFrame;

    // Remove this handler since it is no longer needed

    RootFrame.Navigated -= CompleteInitializePhoneApplication;

}
```

While the application is loading, the emulator and the device will show a splash screen. This splash screen is a 480 x 800 pixel resolution bitmap, *SplashScreenImage.jpg*, with a 24-bit color depth and included in the project. You can replace this JPEG bitmap with your desired splash screen but you have to make sure that you use the same name, set its Build Action to Content and Copy to Output Directory to Do Not Copy. Figures 5 and 6 show the default splash screen and a customized version. The splash screen bitmap must use the 24-bit color JPEG format to work properly. If you use a PNG file instead of a JPEG file, the splash screen won't be displayed. Remember that the user can rotate the device before designing the customized splash screen.

Understanding MainPage.xaml in a Windows Phone Application

The following code snippet shows the original code for MainPage.xaml. These lines shows the code that provides an example of the *ApplicationBar* buttons uncommented. By default, these lines appear commented, and therefore, you don't see the application bar buttons in the design view. If you uncomment the code that begins with *<phone:PhoneApplicationPage.ApplicationBar>*, you will see the application bar buttons. Figure 7 shows the document outline for MainPage.xaml. The document outline allows you to understand the different controls that compose the basic UI.

```
<phone:PhoneApplicationPage

    x:Class="WPBusinessApp.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"

    xmlns:shell="clr-
namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"

    FontFamily="{StaticResource PhoneFontFamilyNormal}"
```

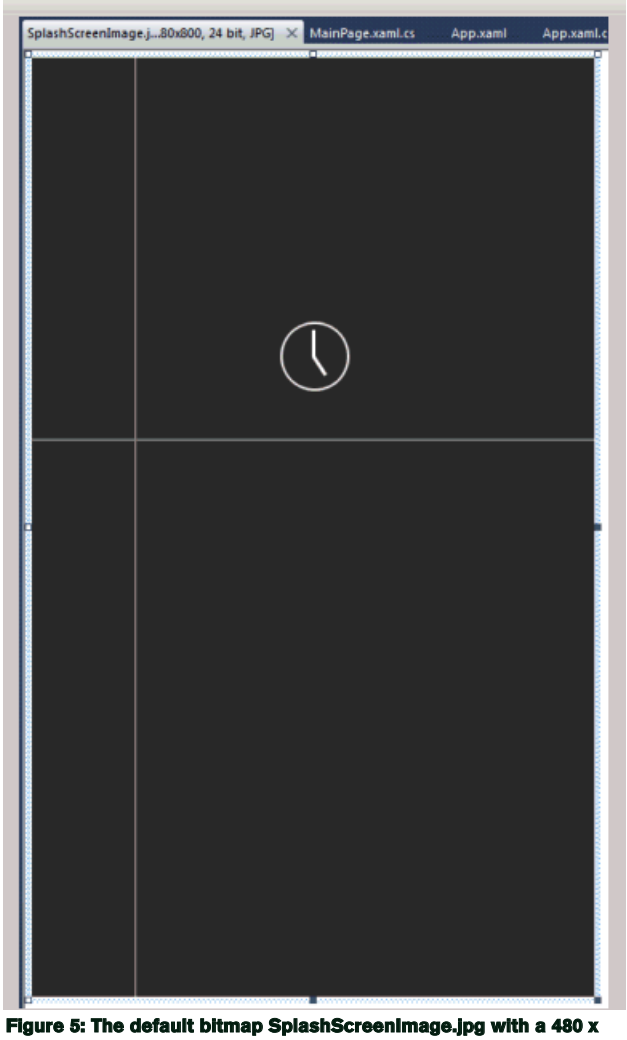


Figure 5: The default bitmap SplashScreenImage.jpg with a 480 x 800 pixel resolution.

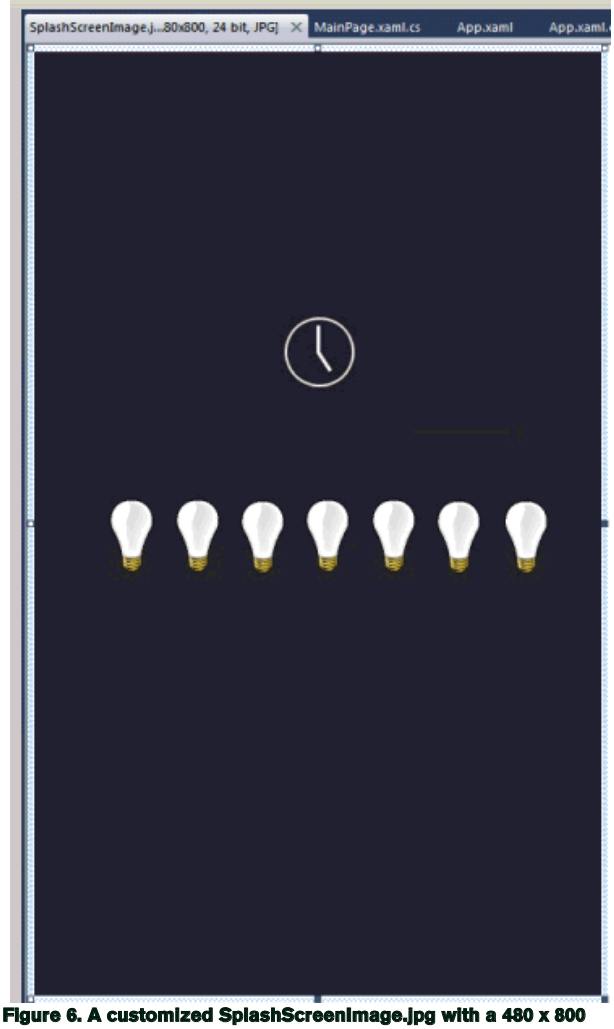


Figure 6. A customized SplashScreenImage.jpg with a 480 x 800 pixel resolution.

```
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="696"
    shell:SystemTray.IsVisible="True">

    <!—LayoutRoot contains the root grid where all other page
content is placed—>
    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <!—TitlePanel contains the name of the application and
page title—>
        <StackPanel x:Name="TitlePanel" Grid.Row="0"
Margin="24,24,0,12">
            <TextBlock x:Name="ApplicationTitle" Text="MY
APPLICATION" Style="{StaticResource PhoneTextNormalStyle}"/>
            <TextBlock x:Name="PageTitle" Text="page name"
Margin="-3,-8,0,0" Style="{StaticResource
PhoneTextTitle1Style}"/>
        </StackPanel>

        <!—ContentPanel - place additional content here—>
        <Grid x:Name="ContentGrid" Grid.Row="1">
        </Grid>
    </Grid>

    <!— Sample code showing usage of ApplicationBar—>
    <phone:PhoneApplicationPage.ApplicationBar>
        <shell:ApplicationBar IsVisible="True"
IsMenuEnabled="True">
            <shell:ApplicationBarIconButton
x:Name="appbar_button1" IconUri="/Images/appbar_button1.png"
Text="Button 1"></shell:ApplicationBarIconButton>
            <shell:ApplicationBarIconButton
x:Name="appbar_button2" IconUri="/Images/appbar_button2.png"
Text="Button 2"></shell:ApplicationBarIconButton>
            <shell:ApplicationBar.MenuItems>
                <shell:ApplicationBarMenuItem
x:Name="menuItem1" Text="MenuItem
1"></shell:ApplicationBarMenuItem>
                <shell:ApplicationBarMenuItem
x:Name="menuItem2" Text="MenuItem
2"></shell:ApplicationBarMenuItem>
            </shell:ApplicationBar.MenuItems>
        </shell:ApplicationBar>
    </phone:PhoneApplicationPage.ApplicationBar>
    <!— End of sample code —>
</phone:PhoneApplicationPage>
```

LayoutRoot is the root *Grid* within *PhoneApplicationPage*. All the page content is included in *LayoutRoot*. However, it is very important to notice that an *ApplicationBar*, with no specific name, is also part of *PhoneApplicationPage*, because the application bar is a special shell control. *TitlePanel* is a *StackPanel* with two *TextBlock* controls:

- *ApplicationTitle*. By default, its *Text* property is set to "MY APPLICATION". You can use it to display the desired title for your application.
- *PageTitle*. By default, its *Text* property is set to "page name," If your application has many pages, you can use this *TextBlock* to specify the actual page. However, if the application just needs one page with controls, this *TextBlock* can consume unnecessary space. If you delete this *TextBlock*, the *StackPanel*'s height will auto-adjust its value. Thus, you can remove *PageTitle* when you need more space to place controls.

ContentGrid is the *Grid* where you have to place the controls. Figure 8 shows a *TextBlock* and a *TextBox* within *ContentGrid*. Silverlight for Windows Phone supports theming, and therefore, each control is going to have a different look according to the theme selected by the user in his/her device.

By default, the Visual Studio 2010 Toolbox provides the most common Windows Phone controls. However, there are important controls that don't appear and you have to add them by selecting Choose items — in the context menu for the Toolbox. One example is the old InkPresenter control that is very useful to allow users to draw directly with their fingers.

You can use the 3D projections, introduced by Silverlight 3. However, if you don't want to write XAML code to add these projections, you will have to use Microsoft Expression Blend 4 for Windows Phone. In fact, if you have to create a complex UI, Expression Blend will simplify your work. Expression Blend also allows you to take advantage of the behaviors to simplify the creation of UI controls that respond to common multi-touch gestures.

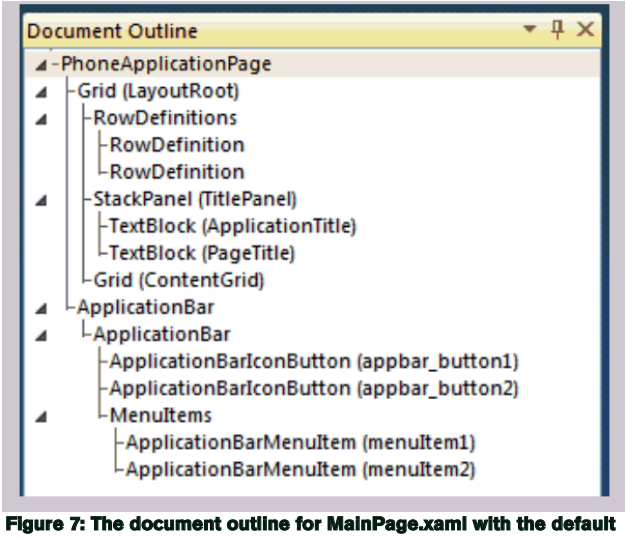


Figure 7: The document outline for MainPage.xaml with the default sample code that shows the usage of ApplicationBar.

Expression Blend 4 provides a more accurate design view when you have to work with the *ApplicationBar*. You can select the desired icon for each button from a list of predefined icons, as in Figure 9.

The *ApplicationBar* is composed of many *ApplicationBarIconButton* controls. These icon buttons display a small icon within a circle, as in Figure 10. The *ApplicationBar* can also include *ApplicationBarMenuItem* controls.

You can attach a *Click* event handler for each of the *ApplicationBarIconButton* and the *ApplicationBarMenuItem* controls. Remember that the *ApplicationBar* control is optional.

When you run the project in Visual Studio 2010 or Expression Blend 4 for Windows Phone, the results of the build process will be deployed in the Windows Phone 7 emulator. The first time you run the application, the Windows Phone 7 emulator will require time to load. However, you don't need to close the emulator to enable another debugging session. It is convenient to leave the emulator running, make the necessary changes to the project, and run it again. If you close the emulator, you will need more time to run the project again. Figure 11 shows the emulator running a very simple UI.

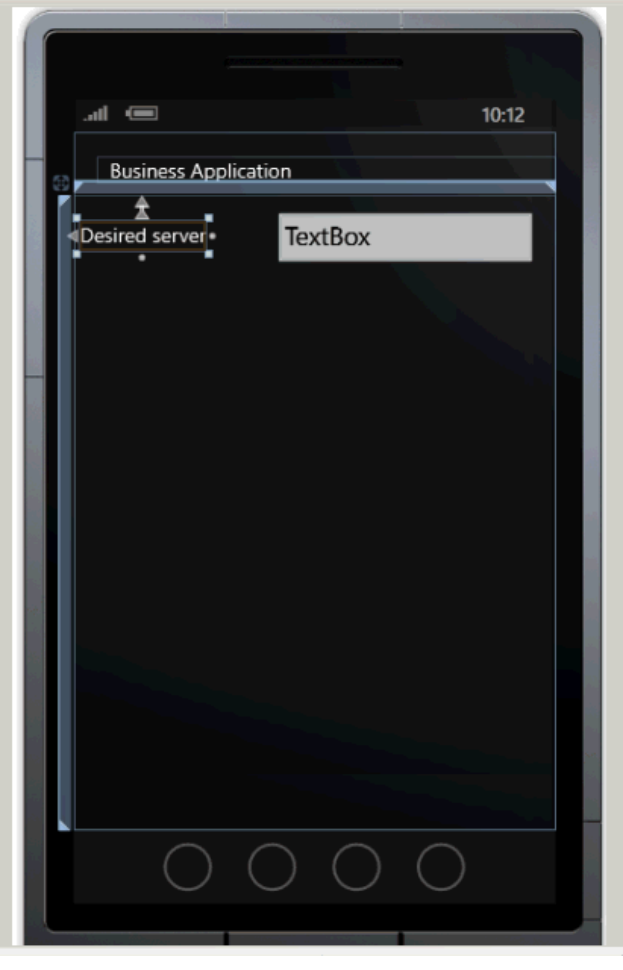


Figure 8. The design view applies the default theme for the controls that you add.

If you click on the Start menu on the emulator, you will see the icon for Internet Explorer. You can access the menu with the icon for the new application by clicking on the next icon button (the arrow). Figure 12 shows the customized icon for a sample application. Figure 13 shows the customized splash screen in the emulator.

By default the project defines support for the portrait orientation in the *PhoneApplicationPage*. The following line indicates the values for *SupportedOrientations* and *Orientation*:

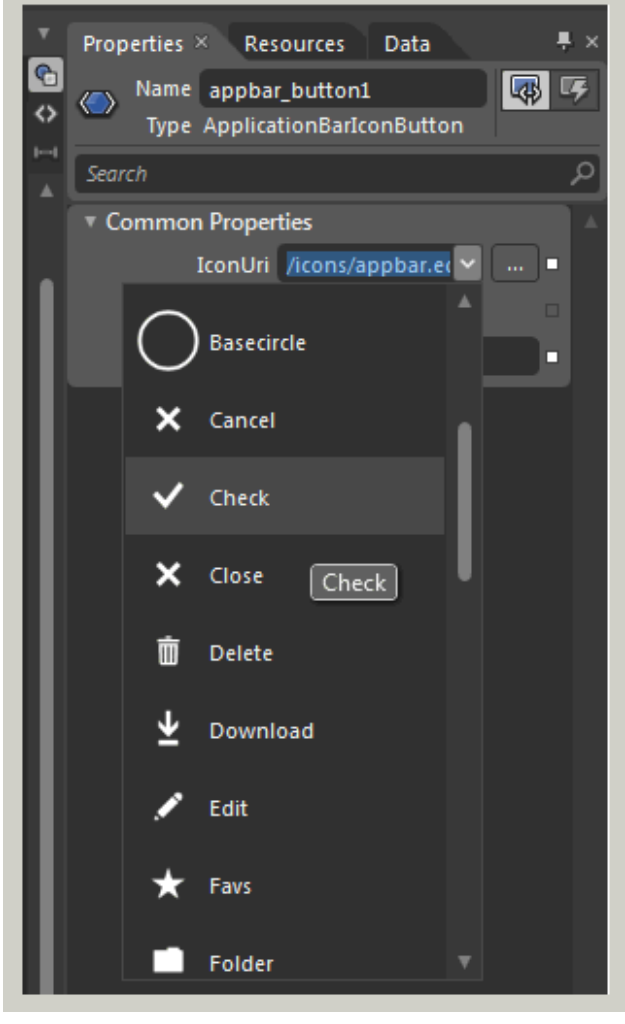


Figure 9: Expression Blend 4 showing a dropdown list with predefined icons for an ApplicationBarIconButton.

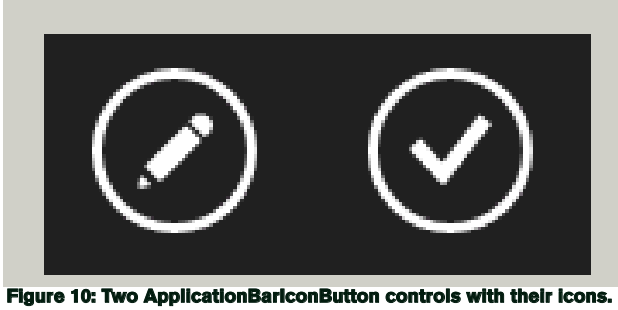


Figure 10: Two ApplicationBarIconButton controls with their icons.

```
SupportedOrientations="PortraitOrLandscape"
Orientation="Landscape"
```

The location and size for the controls will vary according to the device orientation and the value for the *SupportedOrientations* property for the *PhoneApplicationPage*. If you want your application to support both portrait and landscape orientations, you have to specify "*PortraitOrLandscape*" for *SupportedOrientations*. Remember to test the different orientations with the emulator to avoid unexpected locations or sizes for the controls when the user rotates the device. Figure 14 shows an example of an application with a new orientation in the emulator.

Conclusion

Silverlight for Windows Phone 7 Series allows you to take full

Figure 11: Windows Phone 7 emulator in action.

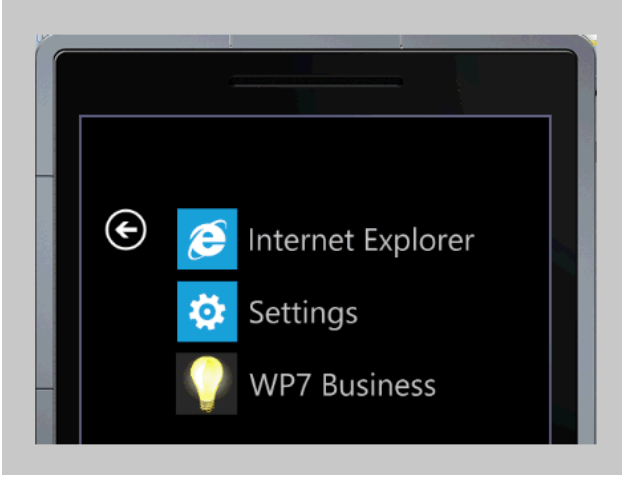


Figure 12: The menu item to access the application with its customized icon.

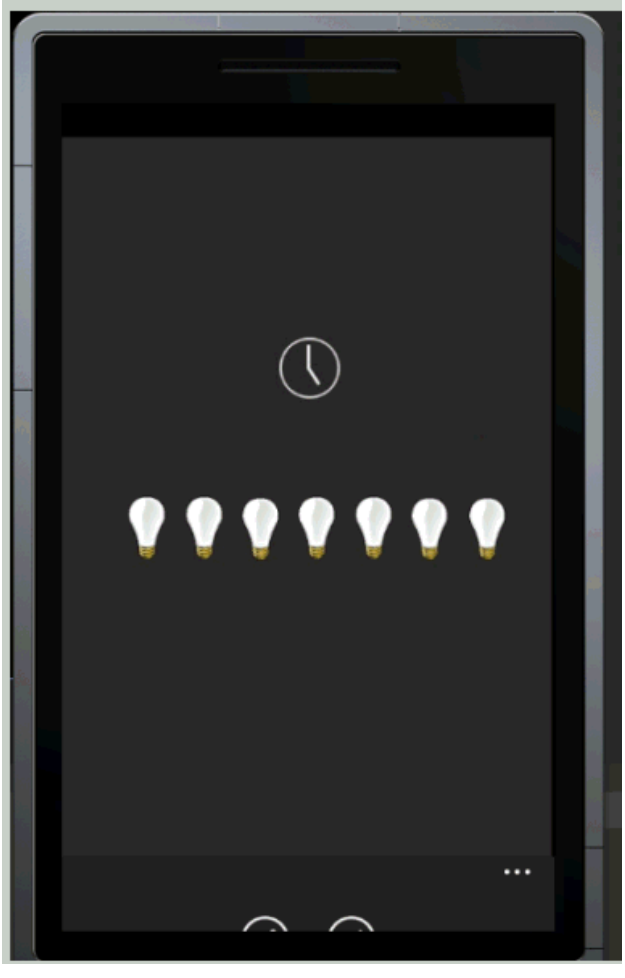


Figure 13: The customized splash screen that appears before the application launches in the middle of the animated transition.

Figure 14: Windows Phone 7 emulator displaying an application with the device in landscape orientation.

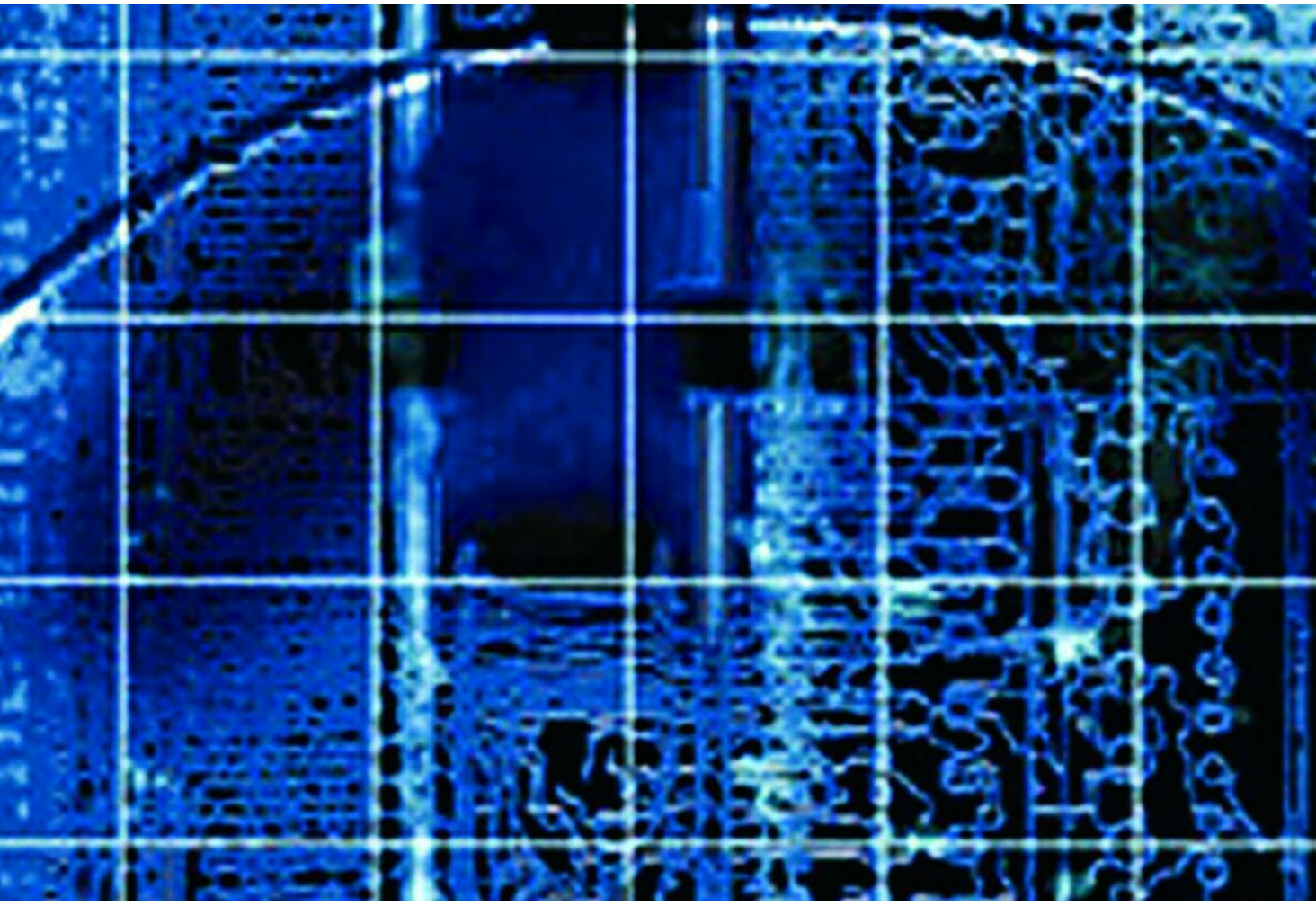
advantage of your Silverlight experience to create mobile applications. Visual Studio 2010 makes it simple to design the UI and debug new applications that target Windows Phone 7 Series and you can test and debug them with the help of the powerful emulator.

Expression Blend 4 for Windows Phone allows you to reduce your development times when you need to create complex UIs. If you want to learn as fast as possible, you can work with Expression Blend in combination with Visual Studio 2010.

There are many new namespaces and you have to learn many new classes to interact with the phone capabilities. However, you still have to work with a combination of XAML and C#. Silverlight for Windows Phone 7 Series provides an excellent opportunity to create mobile, rich, and interactive user interfaces that interact with services on the cloud.

— *Gaston Hillar is a frequent contributor to* Dr. Dobb's *and author of* Professional Parallel Programming With C#: Master Parallel Extensions With .Net 4.

Return to Table of Contents



Virtualization and Mobile Devices

Better multicore energy conservation with mobile virtualization

by Daniel Potts

Energy conservation is increasingly becoming a core requirement in the design of computer systems. On the desktop and servers, the main driver for conservation is the cost of powering and cooling computer systems, and the environmental impact of ubiquitous PCs and massive data centers. For the fast growing mobile device market, green requirements are driven by the need to extend battery life and enhance product performance.

Both multicore silicon and virtualization technology are enjoying increasing deployment in intelligent devices of all kinds. As in the enterprise and on the desktop, embedded virtualization meets the needs of a range of use cases: hardware consolidation, legacy application support and migration, IP isolation, trusted computing, and notably, energy management (EM).

In this article, I examine current assumptions about multicore, power management, and virtualization, especially in mobile devices. I focus on how the mission of virtualization becomes even more critical with multicore systems, expanding to include cross-core energy management. In particular, I compare power management mechanisms in embedded devices, and show how multicore processors require rethinking of power management to conserve energy.

Power Management Mechanisms

Energy conservation focuses both on building greener, leaner circuits from scratch and from using existing systems and silicon more efficiently. On the hardware side, conservation can entail using lower supply voltages and reducing leakage current, reducing power consumption. Under software control, power management technology has traditionally focused on mechanisms to support dynamic voltage and frequency scaling (DVFS) to meet changing load and use policies. Operating voltage and clock frequency, while independent concepts, are usually tied together by the realities of circuit design.

Voltage scaling involves lowering (and raising) processor core supply voltage (V_{cc}) from its nominal value (e.g., 1.3 VDC) downwards towards a minimum value. Since:

```
Power = Voltage x Current
```

voltage scaling, in theory, saves power and energy over time by reducing total power consumption.

However, operating at reduced voltage, processor circuitry inherently either runs slower or consumes more current at the same speed. Typically, scaling down core voltage predicates scaling CPU clock frequency, resulting in decreased overall processor performance.

Downward scaling of CPU clock frequency reduces power consumption, as switching the circuits between logic levels requires less energy to charge and discharge circuit capacitance more slowly.

Examining the two together, we derive dynamic power (P_{dyn}), which varies directly with frequency (f), and with the square of voltage:

$$P_{dyn} \propto f V^2$$

Due to the linear relationship between dynamic power and frequency, scaling frequency alone, while reducing power, does not actually reduce the dynamic energy use: a given number of CPU cycles still requires the same amount of dynamic power (just consumed over a longer time period). In fact, running at a lower frequency may result in increased total energy usage, because of constant static power consumption (from leakage current); static power is independent of frequency, and therefore static energy is proportional to time. At lower frequency, execution time increases, and so does static consumption.

Impact of RAM

Power consumed by RAM is independent of CPU core voltage. It also comprises a static component, and a dynamic component roughly proportional to the number of memory accesses (loads and stores) by the CPU, which depend on the CPU core clock frequency. However, memory accesses are slower than CPU operations, and the CPU frequently stalls waiting for data from memory. When the CPU runs slower, the number of stall cycles (which result in waste of dynamic CPU power) is reduced.

Thus, the relationship between power consumption and core frequency is a complex function of hardware characteristics and program behavior. While power use by CPU-bound programs (which rarely access memory) tends to be minimized at high clock

rates, for memory-bound programs, minimal power consumption occurs at low frequencies.

DVFS Approaches

DVFS is today a standard feature of most microprocessor families. However, due to the complexities described above, DVFS has yielded mixed results from software systems that attempt to use it. The OS kernel policies that implement any decision to adjust DVFS state need to consider:

- Relative CPU and memory power consumption .
- Importance of static vs. dynamic power use by CPU, memory, and other components.
- Degree of memory-boundedness of applications.
- Complex trade-offs between DVFS operating points and sleep modes.
- Dependencies and interactions among CPU cores, buses, and other subsystems.

Multicore

Multiple processors on a single piece of silicon were once a high-end capability implemented only on high-end server and desktop processors. Today they are fast becoming mainstream, with multicore enjoying adoption across the embedded computing landscape. Silicon suppliers routinely integrate specialized companion processors alongside general-purpose applications CPUs on a single substrate (asymmetric multiprocessing or AMP), and are also deploying 2x, 4x, and other parallel configurations of the same ARM, MIPS, Power or x86 architecture cores (symmetric multiprocessing or SMP). Driving this evolution are requirements for:

- Dedicated silicon to process multimedia, graphics, baseband, etc.
- Sustained growth of compute capability without power-hungry high-frequency clocks.
- Running multiple operating systems on a single device (e.g., Android plus a baseband OS).

Multicore systems present steep challenges to energy conservation paradigms optimized for single chip systems. In particular, multicore limits the scope and capability of DVFS:

- Most SoC subsystems share clocks and power supplies.
- Changing the operating voltage (V_{cc}) of one of several SoC subsystems (when even possible) can limit its ability to use local buses to communicate with other subsystems, and to access shared memory (including its own DRAM).
- Clock frequency scaling of a single SoC subsystem also presents interoperability challenges, especially for synchronous buses.
- Multicore systems usually share V_{cc}, clock, cache, and other resources, requiring DVFS to apply to all constituent cores and not to a useful subset.

Silicon supplier roadmaps point to further multiplying numbers of cores, today 2x on embedded CPUs, and soon 4x, 8x, and beyond. This surfeit of available silicon will encourage designers to dedicate one or more cores to particular subsystems or functionalities (CPU-function affinity). Some dedicated operations, like media processing, will use cores in a binary fashion — at full throttle or not at all. However, most other functions will impose varying loads, ranging from a share of a single core to saturating multiple cores. All-or-nothing use is fairly easy to manage, but dynamic loads on multiple cores present much greater power management challenges.

Power Management Across Operating Systems

Operating systems do not excel as resource managers. If OSes were more capable in this area, virtualization would not enjoy its mission-critical role in the data center. Embedded OSes aren't any better at managing resources than their server counterparts: They assume static provisioning and full resource availability, with fairly simplistic state models for resources under their purview.

Many intelligent devices also deploy multiple OSes: high-level OSes like Android, Linux, Symbian, Windows CE, or Windows Mobile to provide user services and to run end-user applications, and one or more RTOSes to handle low-level chores like wireless baseband and signal processing. These OSes and the programs they host may run on dedicated silicon, may occupy dedicated cores on a multicore system, or can also run in dedicated partitions of memory and cycles of a single shared CPU.

High-level applications OSes typically include their own power management schemes that leverage DFVS (e.g., Linux APM and DPM, and Windows/BIOS ACPI). Most RTOSes eschew any operations that curtail real-time responsiveness, leaving OEMs and integrators to roll their own or do without.

Whatever the inherent energy conservation capability of resident OSes, there remains the challenge of coordinating efforts in a multi-OS environment. Even if one of several deployed OSes is capable of managing power in its own domain, it will be unaware of the capabilities and state of its peer OSes elsewhere in the system, adding to development and integration headaches. Even if all coresident applications OSes and RTOSes have some power management capacity, how can systems developers and integrators coordinate and optimize operation and energy conservation policy across OS domains?

Managing Power in Mobile Handsets

Most of today's smartphones employ separate dedicated CPUs for applications processing, multimedia and graphics, and for real-time wireless baseband modem operations. As multicore CPUs (and virtualization) become more ubiquitous, designs are migrating these separate operations onto partitioned single and multicore processors. A good example is a mass-market smartphone such as the Motorola Evoke (see http://www.ok-labs.com/whitepapers/sample/motorola-evoke-teardown).

Lower cost mass-market smartphones consolidate diverse functions onto a single processor with one or two cores. Next-generation devices will build on silicon with even greater numbers of available cores, and will surely find ways for each subsystem to consume available compute capacity.

In these devices, one subsystem would be the baseband modem, whose real-time software stack comprises a load that fully occupies one or more cores during peak processing (e.g., for streaming or voice conferencing), but typically consumes perhaps a fifth of a single core's capacity when quiescent. Another subsystem would be an HD multimedia stack, requiring an additional core (or more) at full load, and zero when no media is displayed. A GUI stack might use a half core during heavy user interaction, and zero when quiescent. User applications would consume any remaining compute capacity when executing, and would occupy either zero cores when quiescent, or represent some other finite load with background processing.

DVFS Challenges

Clearly, each of the functional stacks and the OSes that host them presents unique challenges to managing power with DVFS. In combination on a multicore CPU, it becomes nearly impossible to determine useful DVFS operation points and policy for transition among them, both on a per-function basis and a coordinated one.

The above scenario clearly illustrates that localized DVFS schemes are inadequate to address the needs of next-generation multistack, multicore designs. Further analysis of the scenario also highlights the limitations of coarse-grained assignment of functions to available CPU cores:

- Peak loads for different subsystems can consume most or all of one or more cores' compute capacity.
- Gross assignment/dedication of functions to cores can waste available compute capacity and potentially starve functions at peak load.
- Real-world total load is unpredictable due to third-party applications (e.g., with Android Market), and additional demands placed on communications and multimedia stacks from those applications and the traffic they generate.
- Scalable loads dictate sharing of available CPU cores across functions.
- Most silicon cannot run stably at all frequencies and voltages. Real-world energy management paradigms build on discrete stable pairings of voltage and frequency (operating points).

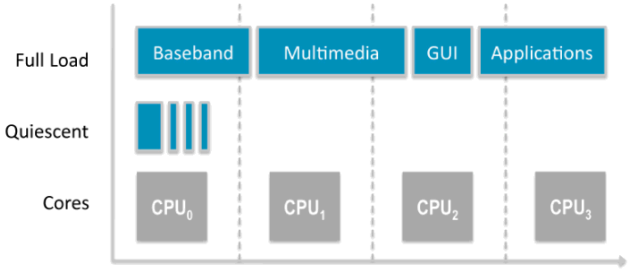


Figure 1: Full and quiescent loads across available cores.

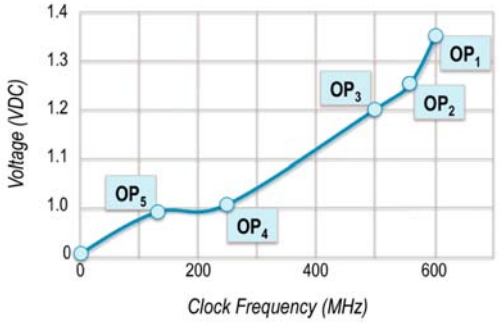


Figure 2: DVFS operating points between CPUs.

Energy Conservation and Virtualization

Rather than try to salvage legacy power management paradigms from each functional domain, let's employ the approach favored by data center IT managers — using virtualization for energy conservation.

DVFS wrings incremental gains in energy efficiency by reducing voltage and clock frequency. A given CPU offers developers and integrators a set of safe "operating points" with fixed voltages and frequencies. With varying load, EM middleware or EM-aware OSes transition from operating point to operating point (Figure 2).

Stop and Full Throttle for an ARM Cortex A8 CPU

A logical extension of applying DVFS is reduction of voltage to 0 VDC and completely stopping the CPU clock. That is, utilizing only two operating points — Full Stop and Full Throttle — but employing them across the range of available cores: OP1 uses one core, OP2 uses two, etc.

In multicore systems without virtualization, or with simple partitioning, wholesale shutdown of CPU cores presents nearly insurmountable challenges, because loads (OSes and applications threads) are tightly bound to one or more cores (complete CPU affinity, as in Figure 3):

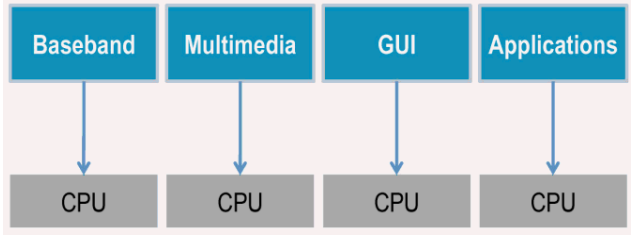


Figure 3: Complete affinity among functional subsystems and CPUs.

- Shutting down a CPU core requires a policy decision between a shallow sleep mode (with fast entry and exit, but significant remaining leakage power) and a deep sleep mode (with low leakage power but high overhead at both entry and exit).
- Migrating loads across CPUs is nearly impossible; only loads already running as SMP can shed CPUs, but not migrate across them.

Introducing virtualization neatly addresses the challenges of CPU shutdown and CPU core affinity. First, instead of binding loads to actual CPUs, the presence of a full-featured Type I Hypervisor associates subsystems with dedicated virtual CPUs. Based on real compute needs and on policy established at integration time, the hypervisor can bind virtual CPUs to one or more physical CPUs (Figure 4) and/or can share available physical CPUs among virtual CPUs as needed (as suggested in Figure 1).

To facilitate energy conservation, a hypervisor enables full stop of underutilized CPU cores by (re)mapping virtual CPUs (and their loads) onto fewer physical CPUs (Figure 5).

This neat trick is only possible via the construct of virtual CPUs, which facilitate arbitrary mapping of loads to physical silicon and migrating running loads transparently across CPU cores. The resulting consolidation means that, on average, more CPUs are in an off state and they remain there for longer, saving substantial energy and over time.

Also, quiescing whole cores leads to linear (and therefore highly predictable) performance-energy trade-offs, unlike DVFS, and is therefore easier to manage. Moreover, DVFS can still be employed on the active cores for fine-tuning energy-performance trade-offs. Since power management is now handled by the hypervisor, with full knowledge of performance requirements, hardware-imposed constraints such as common core voltage are readily incorporated.

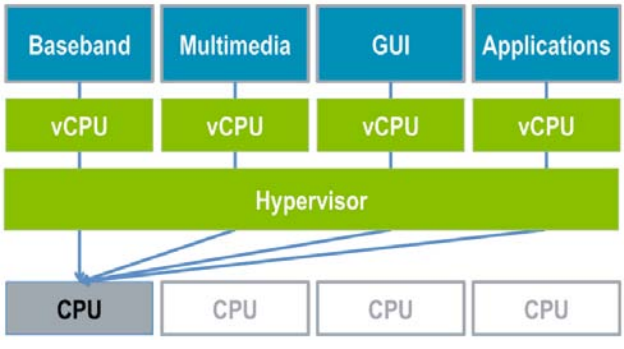


Figure 5: Shutting down underutilized CPUs and consolidating loads onto remaining core(s).

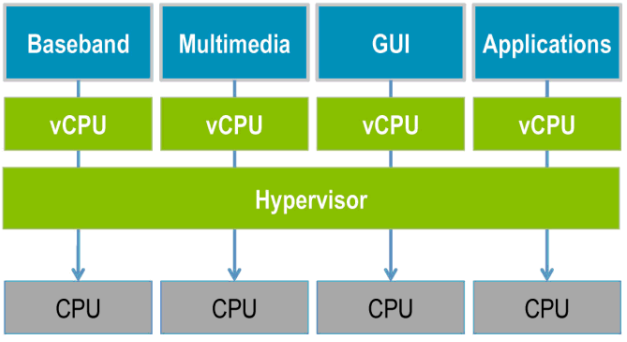


Figure 4: Affinity of loads with virtual, not physical, CPUs.

Virtualization: Ideal for Global Energy Conservation

As mentioned earlier, embedded OSes are notoriously poor at resource management. Those with native power management schemes have "been taught" to monitor their own loads and make energy policy transitions. They are not, however, equipped to manage power and CPU utilization outside their own limited domain, on other CPU cores running different OSes and loads. For multiple, diverse hosted OSes in a multicore system, effective power management must "step outside" of the local context of functional subsystems (baseband, GUI, etc.) to a scope that encompasses all subsystems together.

This article has offered a brief review of power management mechanisms and energy conservation, and challenges presented by modern multicore systems. Of currently available software-based energy conservation mechanisms, only virtualization is positioned (in the global architecture/stack) to manage power for all cores and all functional subsystems in concert. Since it is the hypervisor that actually dispatches threads to run on physical silicon, it is uniquely and ideally placed to comprehend real CPU loading (not calculated guest OS loads), and to scale power utilization by bringing available cores in and out of service.

—Daniel Potts is VP of engineering at Open Kernel Labs.

Return to Table of Contents

The Android Developer Experience

Is this the 'Droid' you were looking for?

By Mike Riley

With the success of the Apple iPhone, a new surge of development opportunities has arisen in the consumer mobile computing space. However, due to Apple's walled garden approach, some developers have been less compelled to spend a lot of time investigating its SDK. Compounding this closed environment is Apple's insistence to exclusively base the platform's programming syntax to be the same as Apple's desktop OS language of choice, that being Objective-C. While I've written a few simple applications in Objective-C, the language never really captured my interest, and its inability to reuse the code (especially the Cocoa GUI-based elements) on non-Apple platforms further reduced my desire to learn it more deeply. While Objective-C bindings can be wrapped around scripting languages like Perl, Python, and Ruby, the amount of work necessary to wire up cross-platform unfriendly GUIs isn't worth the effort. Besides, it doesn't appear that Apple will be allowing any other programming languages to reside on its iPhone platform anytime soon.

When Google announced its Linux-based Android platform that hosted applications written using Java syntax, my mobile development interests were quickly realigned to the open platform mantra that Google was chanting. And yet, even though Google released the Android SDK earlier in 2008, it wasn't until the physical hardware development units became available in December 2008 that I decided to take a more serious look at this alternative mobile OS.

Signing Up

To become a privileged member of the Android Developer Program, you must first visit the Android Market and create a Gmail account if you don't already have one. The Gmail account is required for Google Checkout, employed for the one-time $25 charge of being granted the permission to post Android applications to the Android Market after agreeing to the lengthy developer agreement. It is also through this marketplace that developers can purchase a single G1 developer device.

The Hardware

Unlike the G1 model currently sold at T-Mobile and other stores, the developer G1 is an unlocked model that permits SIM cards from other carriers besides T-Mobile to be used with the device. More importantly, this G1 version offers root user access, giving you full access to the Android filesystem as well as the ability to re-flash the device without constraints. Indeed, adventurous hackers have already created a minimal Debian-based Linux distribution that can be flashed onto the G1. This delightful experiment is nowhere near ready for primetime use, but the fact that it can even be done at all will unquestionably motivate other Linux mobile enthusiasts to charge ahead with even more innovative hacks. Needless to say, the developer unit isn't a device for the technically challenged. As a mark of special exclusivity, the developer unit G1's rubberized back plate has a custom Android polygon design

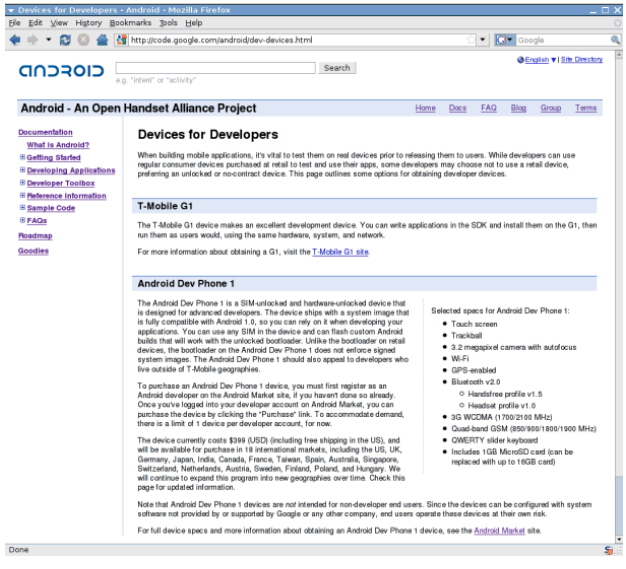


Figure 1: The Android Open Handset Alliance home page provides a starting point for obtaining the Android SDK and ordering a G1 developer device.

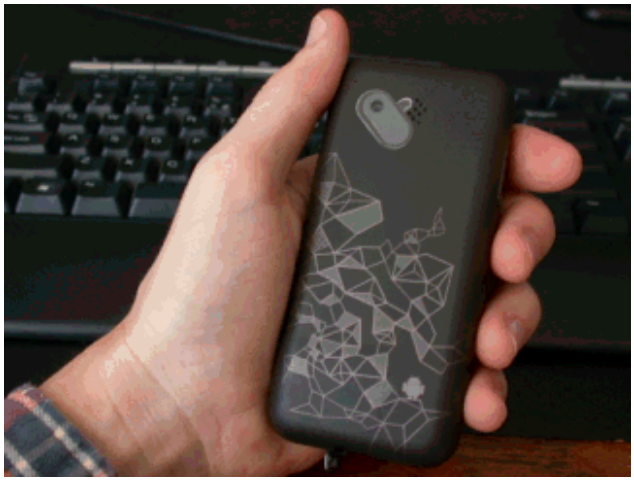


Figure 2: The back cover of the G1 dev unit sports a distinctive polygon art form advertising to other Android users of the device's elevated status.

imprinted across it for additional "Android Club" geek-cred points.

Based on the T-Mobile G1 developed by prolific Windows Mobile hardware designer HTC and modeled upon the HTC Touch, the developer version is everything initial reviews of the consumer G1 said it was: a good first attempt with adequate single touch screen resolution, feature laden with 2G/3G, Bluetooth, GPS, and WiFi radios, accelerometer, digital compass and…an abominable battery life. Since I have been using the G1, I have yet to make it through the day without charging it at least twice. However, barring the fact that this ambitious platform performs miserably in the power consumption category, it nevertheless provides a stable development reference platform to create and test Android applications in the real world.

There were other hardware issues that bothered me enough to have to correct with additional accessory purchases. First and foremost, the 1-GB microSD card included with the phone is simply too anemic by today's multimedia standards. While I debated upgrading to a 16-GB card, I opted for a much less expensive 8-GB capacity instead. Another truly bone-headed design limitation (or greedy aftermarket accessory approach, depending on the conspiracy factor) is the G1's proprietary HTC miniUSB-like adapter that is used for both the unattractive, bulky earbuds as well as the power recharge connection. Without additional adapters, G1 users cannot, for example, listen to music and charge their device at the same time. Fortunately, an inexpensive splitter from Amazon (search for "2-in-1 mini USB adapter") solved this annoying dilemma at the expense of additional bulk clipped to the base of the phone. Finally, what seems to be a standard omission for most mobile smartphone devices these days is a protective carrying case. I decided I had invested enough in phone extras and instead simply left the Mylar cover on that protected the screen during shipping.

Incidentally, in keeping with the burgeoning Android OS hacks, a few enterprising Linux netbook hobbyists have succeeded in compiling the Android source targeting an Asus eeePC subnotebook. While this certainly generated a buzz in the Android community, using such a device as a G1 replacement isn't practical due to the lack of hardware support such as the sensors and various drivers. Regardless, the feat further strengthens the reality of Android's open-source platform promise.

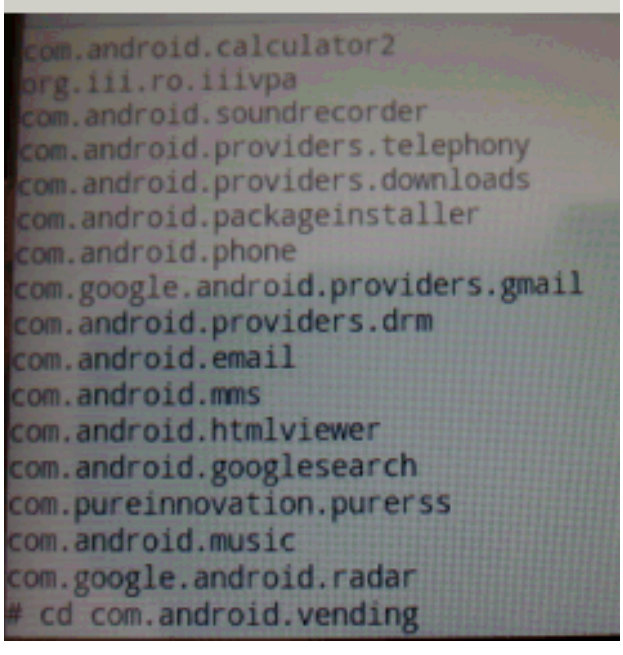


Figure 3: The Google Terminal Android application running on the G1 with root level access.

The Software

The OS version shipping on the developer G1 that I received was an unlocked Nov. 3 1.0 build running the Linux 2.6.25 kernel. While this first-generation version is a remarkable achievement for the Android team, there are unsurprisingly a few rough edges. I've read Android reviews stating how stable it is and how it's nearly crash-proof. That didn't turn out to be the case for me. I had the OS unexpectedly and forcibly reboot on me several times, most often when using Android's Music application and listening to MP3 file playback.

There is also the fact that the Android designers took the iPhone approach of masking the underlying filesystem such that files, even those being stored on the microSD card, cannot be easily accessed and manipulated from a standard file management/browser approach using the standard application set. Instead, a visit to the Android Market is necessary to more easily access the filesystem when untethered from the host development environment. The two I found most helpful are a third-party simple file browser called OI File Manager, and Google's own Terminal shell program. On consumer G1 models, the Terminal program isn't very helpful since root access is locked, but on the developer G1 model, root access is helpful, even a necessity at times. Since

Figure 4: Screen capture of the Terminal application running on the G1, grabbed using Android's ddms developer debugging tool.

the version of OI File Manager I used can only affect a single file at a time, I more frequently call upon the Terminal program to perform most of my file manipulation.

More importantly, the Terminal helped me alleviate one of the most painful design oversights of the Android Market and various Google applications like the Maps and Browser. This ravenous storage eater is the current version's inability to have these standard applications store their downloaded data caches on the microSD card and not the main onboard memory of the device. Googling for Android Market and cache together will yield a litany of angry complaint links from end users and developers alike of how their G1s are rapidly exhausted of memory as a result of these two applications lacking such an obvious feature.

As of this writing, the Android team is working on an update to the Android OS, codenamed "Cupcake," that is expected to fix this glaring oversight as well as add new features such as stereo Bluetooth (known as A2DP) support and a few other nice enhancements. In the meantime, hard-pressed developers

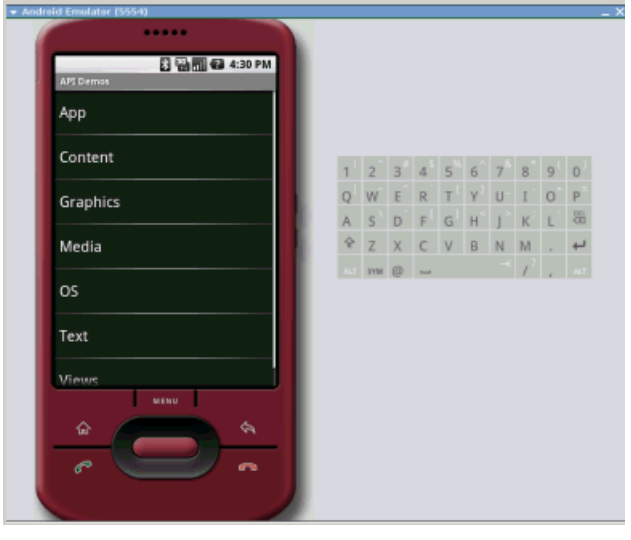


Figure 5: The Android emulator running the ApiDemos sample application.

equipped with the Terminal program have come up with a very easy UNIX-like solution to the cache problem. After downloading and installing the Terminal application from the Android Market, launch it, su to root and create a new directory called /sdcard/marketcache (this can be named anything, and even reside in a deeper subdirectory). Navigate to the /data/data/com.android.vending directory, delete the cache folder located there (rm -R cache), and create a symlink to a new cache folder location on the microSD card (in this example, execute ln -s /sdcard/marketcache cache as root). This caching issue is a problem not just with the Android Market app, but several others as well, including the Browser, Maps, and even some third-party Android programs that prefer to keep cached data on the device instead of providing an option to store locally cached data on removable storage. Because this is such a prevalent problem, community contributors at modmygphone.com have created a wiki entry outlining the steps needed to alleviate the cached storage constraints with various Android applications. Visit modmygphone.com/wiki/index.php/Caches_To_SD_Card for details. Note that because these cache relocations require root level access, they can only be performed easily on the developer G1 models. End consumers will either have to hold out for Cupcake, cold reset their devices to clear out the cache (and everything else they had installed), or embolden themselves with attempting to unlock the commercial G1 handset on their own and possibly breaking the phone in the process.

If you prefer not to mess with the filesystem and want to minimize the cache size, you do have another option. Developer Jay Freeman has created an Android Market front end at cyrket.com that anyone can visit with a web browser. This helpful site also allows non-Android owners to see what the Android Market has to offer. As long as Jay keeps cyrket.com alive, I will continue to visit it first before perusing the latest market updates via the Android Market application.

The Development Environment

Setting up the optimal configuration for Android development is a fairly easy process that takes less than 15 minutes, not including download time for the software. The Sun JDK 1.5 or higher is an obvious initial requirement to run both the Eclipse IDE and the Android Developer Toolkit (ADT) plug-in set. I chose to use the latest 1.6 release. Once the JDK version requirement is met, download and install the Eclipse IDE. The nice thing about Android development is that it can be performed on Windows, Mac, or Linux, the same platforms that Eclipse runs on. Any Eclipse version 3.2 or higher will do. I chose the latest 3.4.1 Ganymede release. While the Eclipse IDE isn't absolutely essential, the Android team has written the ADT exclusively for the Eclipse environment that makes launching and managing the Android emulator much easier. There are reports of some brave attempts to wire up these pieces within Sun's own NetBeans IDE, but unless there is some serious aversion to Eclipse, this effort appears to be more trouble than it's worth.

Next, download and install the Android SDK. This can be installed anywhere as long as the Eclipse plug-ins can consistently

locate the installation path. Finally, launch Eclipse and grab the ADT via the usual Eclipse plug-in install route. Select "Software Updates" from the Eclipse "Help" menu, click on the "Available Software" tab, then the "Add Site" button and enter the location of the ADT. Restart Eclipse and enter the installation path of the Android SDK in the Window —> Preferences —> Android dialog box.

Windows and Mac users should be ready to go, but Linux 64-bit users will need to ensure they have the 32-bit emulation libraries (ia32-libs) installed; otherwise the proprietary, closed source emulator compiled in 32-bit code for the Linux environment will fail to run on that 64-bit host.

Check to make sure everything is correctly configured by launching Eclipse, importing the APIDemos project from the Android SDK demo folder, and running it. Doing so should launch the emulator and, once confirmed running (initial instances take between 30 seconds and a minute depending on host computing resources), selec t Run. This will compile the project's source code and various resources into a single Android package called an "apk" file, download and instantiate APIDemo in the emulator. Fortunately, subsequent Android runs will occur much faster as long as the emulator is allowed to keep running in the background.

The emulator itself is for the most part an accurate reflection of a real Android device like the G1. Unfortunately, the emulator used during the writing of this article could not emulate the sensors or radios such as the accelerometer or GPS. However, the aforementioned folks at OpenIntents have created a helpful alternative to help simulate such inputs at the cost of modifying an Android project's code to get it to talk to OpenIntent's Sensor Simulator. However, I suspect that if the development community clamors loudly enough, the Android SDK team will respond with such capabilities built into future SDK releases.

Once acclimated to the emulated environment, it's time to test sending applications to the actual G1. Windows users first need to download and install the Android USB Windows driver. Linux users have more work to do. A rule has to be added to the host computer (example, for Ubuntu Hardy users, a new file needs to be created at /etc/udev/rules.d/50-android.rules containing the following: SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666". Once saved, chmod a+rx the file and the host computer should be able to identify and communicate with the Android device when plugged in via the USB port. Mac users fortunately do not have to mess with any drivers or settings, as the Android SDK tools and Eclipse environment will automatically recognize the device. As Google's SDK installation documentation boasts, "it just works." And they're right — it does.

The Android plug-ins will automatically assist Eclipse in targeting the attached device and downloading and running the desired apk accordingly. Simple. With both the emulator and device successfully receiving and running the API demo, you're ready to start writing your own Android applications.

Writing an Android Application

Creating an Android application is relatively painless, though due to the forced abstraction of resources including layout, strings for easier internationalization, and raw resources such as audio clips and images, a bit more forward thinking and manual wiring of the interface elements is required.

Assuming all the prerequisites have been met and the API demo successfully launched on either the emulator or the developer

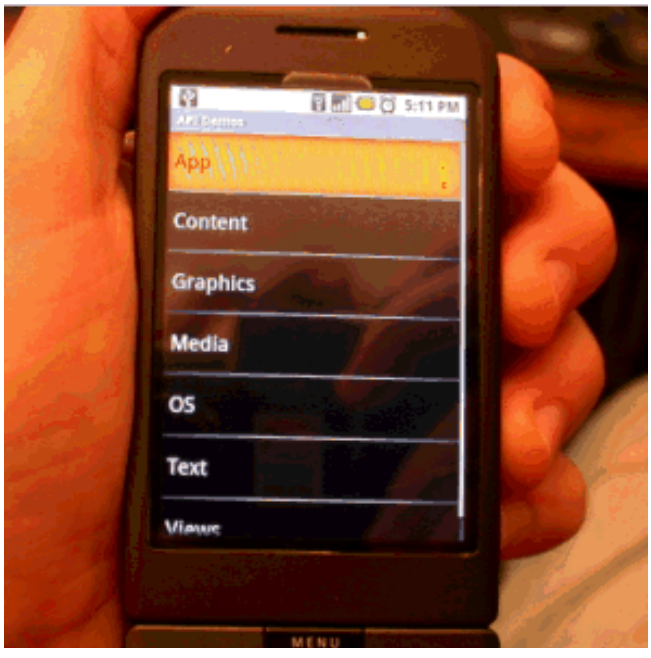


Figure 6: The G1 running the ApiDemos sample application, downloaded and instantiated from the ADK Eclipse plug-ins.

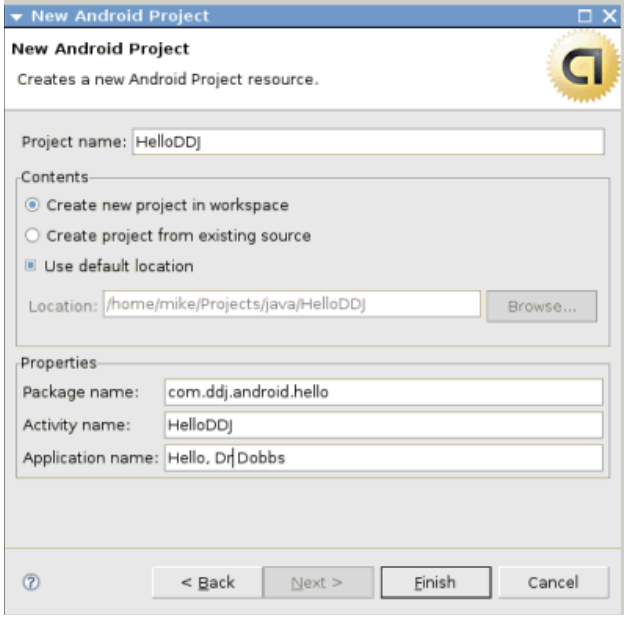


Figure 7: Creating a New Android Project with wizard assistance from the Android Developer Toolkit.

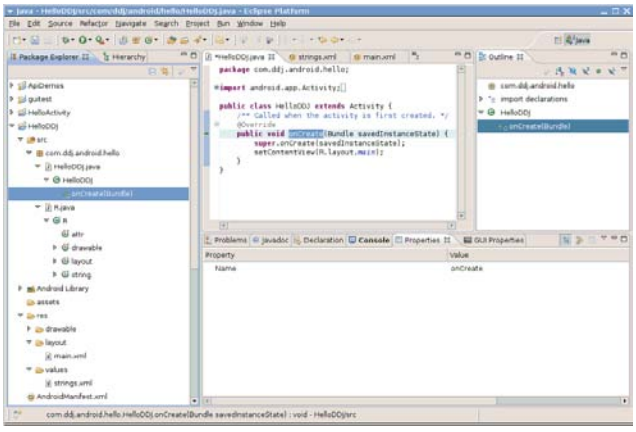


Figure 8: Examining the generated Android project files.

device, the Android Eclipse plug-in can be called upon to generate a skeleton Android application, complete with required class wrappers, files, and folder layouts. However, logic and interface code needs to be written and wired up before seeing anything meaningful on the Android display.

After launching Eclipse, select File —> New —> Project... and select Android Project. Name the project (for example, "HelloDDJ") along with a Package name (following the Java package namespace convention; for example, "com.ddj.android.hello"), Activity name (the name of the subclass from Android's Activity class; for example, HelloDDJ) and lastly, a human-readable Application Name (Hello, Dr Dobbs).

Clicking the Finish button will automatically generate the basic code and layout XML, enough to execute and display a simple window with the name of the application in the titlebar and a Hello World, (followed by the Activity name, in this example's case, HelloDDJ) text string in the body. However, before doing so, let's first explore what files the wizard generated. Expanding the HelloDDJ src tree reveals two auto-generated class files:

- The HelloDDJ.java containing the HelloDDJ class, extended from Android's Activity class.
- R.java final class.

This R class contains pointers to string, images, layouts, audio, video and other application assets. The R class is automatically managed and updated by the ADT whenever file assets are added or removed to the project's resfolder. I discovered that this R generator doesn't like files that begin with numbers, so name your image and audio files beginning with standard ASCII characters.

Examine the contents of the resfolder and note the three subfolders: drawable (containing images such as the generic program icon, as well as other bitmap graphics in PNG or JPEG formats), layout (containing the program's various layout definitions in XML format), and values (containing the program's strings structured in XML format). One folder that isn't auto-generated but required if adding file assets such as audio (AAC, AMR, MIDI, MP3, OGG, WAV, and WMA encoded files are valid Android audio playback formats) and/or video (H.263, H.264 and MP4 encoded video files are valid Android video playback formats) is the raw folder.

Because Android enforces strings to be referenced rather than embedded in the code, using a System.print.out() statement to display text in a window isn't valid. Instead, the strings are identified in the strings.xml file and referenced in the AndroidManifest.xml and vairous layout xml files (for example, main.xml). This keeps all the program strings in a single file, greatly simplifying localization and generally organizing all the strings displayed in the program. Of course, the Text properties of the various Android display elements like the TextView class can be programmatically modified, but the static strings like those found in about boxes, forms, and dialogs are contained in the strings.xml file.

Thus, to modify the HelloDDJ string, open the strings.xml file, select the hello string value and replace it with "Dr Dobbs says hello from Android!" Save the project and run the application. This will fire up the emulator (or send the application straight to the G1 if it is configured and hooked up to the host computer) and display the Hello, Dr Dobbs window containing the edited welcome message.

Once the rudimentary steps of creating a simple Android application are understood, go to the next level by writing a few conceptual tests that call upon frequently accessed features, such as setting an alarm, displaying alerts, displaying a web page, or capturing images from the built-in camera. The Android SDK documentation has a section dedicated to such tips (http://developer.android.com/guide/appendix/faq/commontasks.html).

One of the deflating aspects of the ADT is the lack of any kind of GUI builder in the current release. Instead, each layout needs to be described in manually constructed XML. Once well formed, this XML can be rendered in the layout view to check for accuracy and positional placement. Android offers several layout models depending on the application's requirements such as screen orientation, form resizing, absolute positioning, and others. Fortunately, Android developer Brendon Burns has created both an applet and standalone Java-based design tool called "DroidDraw" that helps UI designers create and export valid Android XML layout files. While I expect the Android development team to eventually incorporate roundtrip drag-and-drop GUI construction within Eclipse, DroidDraw supplies a more than adequate placeholder for Android UI generation.

As the Android developer community continues to expand, I expect more tools like DroidDraw to spring up, helping developers scratch the itches created by deficiencies in the SDK. And if Android's marketshare expands to the percentages projected by Google and other mobile industry analysts, there will no doubt be commercial add-ons, libraries, widgets, and code management tools created for an eager and demanding Android developer audience.

Books, Websites, Cool Apps, and Other Developer Resources

As enthusiasm for Android development continues to expand, more websites, books, and articles will add to the already adequate

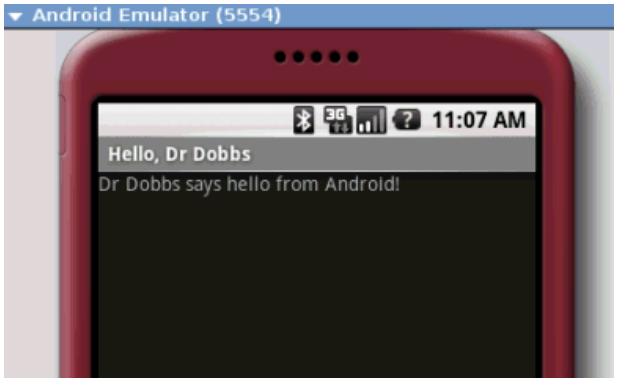


Figure 9: Dr Dobbs says 'hello' from the Android emulator.

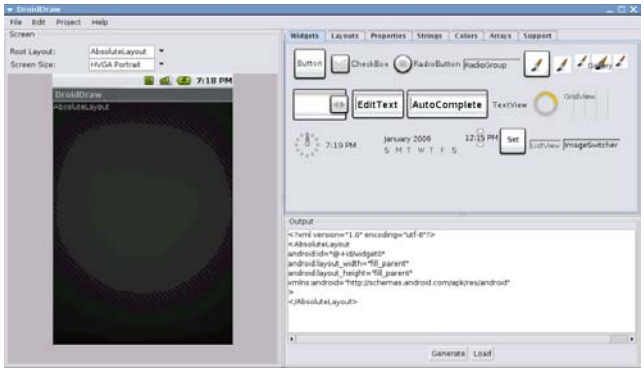


Figure 10: DroidDraw is a very useful donation-ware utility for quickly assembling Android UI layouts.

amount of information available today. Some of the more useful resources I discovered while working on my own Android development efforts include the following:

- Android Essentials by Chris Haseman, published by Apress (http://www.apress.com/book/view/1430210648). This was one of the first Android developer titles on the market, and while the content is still relevant, more recent titles have eclipsed its emulator-centric tutorials.
- Hello Android by Ed Burnette, published by Pragmatic Bookshelf (http://www.pragprog.com/titles/eband/hello-android). Just over 200 pages, this short book gets impatient developers like me up to speed quickly on programmatically controlling Android's most compelling features. Visit dobbscodetalk.com for my more comprehensive review of this title.
- Professional Android Application Development by Reto Meier, published by Wrox (http://www.wrox.com/WileyCDA/WroxTitle/Professional-Android-Application-Development.productCd-0470344717.html). One of the most comprehensive books on Android development available today.
- Android SDK Documentation (http://developer.android.com/guide/index.html). A first stop for anyone interested in writing Android applications. Do not pass go, do not start writing a line of code or post questions in the various Android website forums until this reference material has been read and digested.
- Android Development Community (http://www.anddev.org/). A community-driven Android developer website chock full of reviews, tutorials, coding problem threads and other discussion groups, each of which can be subscribed to via RSS.
- Android Developers Google Groups (http://groups.google.com/group/android-developers?pli=1). Google monitored developer discussion forum with over 14,000 members at the time of this article's publication.
- Google Maps (standard application on the G1; http://maps.google.com/maps). This application raised the roof at early Android demonstrations. While the compass mode in Streetview isn't the most efficient way to peruse street-level imagery, it undoubtedly continues to be the coolest.
- ShopSavvy by Big in Japan (http://www.biggu.com/applications/). One of the big winners in the Android Developer Challenge (code.google.com/android/adc.html), ShopSavvy uses the G1's built-in camera to capture a product barcode, identify the product, and present best price options online and local retail outlets based on the user's GPS coordinates.
- Twidroid by Ralph Zimmermann and Thomas Marban (http://www.twidroid.com/). Twitter client with built-in geotagged photo blogging support. This mobile microblogging client is also one of the first to support Laconi.ca's PHP-based open microblogging platform.

Conclusion

I have been developing applications on various mobile platforms since the days of the original Palm OS, spending most of that timespan within various incarnations of the Windows CE environment. When the first-generation iPhone shipped without an open SDK, my desire to do anything interesting on that device quickly wained. Besides, I already had everything I needed on my Windows Mobile gadgets anyway. Of course, the iPhone was a game-changing device that brought to the less technical crowd what I had already been doing for years.

Android represents an attempt to satisfy both the extreme-tech and the non-tech audiences. I suspect that for the time being, Android will serve code-hungry developers for a while but will struggle to gain significant marketshare in the segment that Apple's iPhone is serving so effectively. But as a result of the open conversations and open platform, faster innovation is more likely on Android compared to any other modern day mobile OS. How this will translate into commercial ventures is still murky, but from a pure development standpoint, I can't recall when I had this much fun (coupled with a few brief bouts of frustration) writing applications for a mobile device. Even with its first-generation flaws, the Android is the smartphone that currently goes in my pocket every morning, leaving my other mobile devices behind to collect dust on the retired gadget shelf back home.

—Mike Riley is a Dr. Dobb's *Contributing Editor.*

Return to Table of Contents

Porting JavaScript Applications to the iPhone

Porting a web-based application to the mobile device

By Tom Thompson

What a difference a few years can make. Case in point: The way you implement mobile solutions for enterprise applications has changed dramatically. What caused this shift? The iPhone did, but probably not for the reasons you expect.

In my article "The Android Mobile Platform" (http://www.ddj.com/mobile/210300551), I mentioned that the iPhone liberated the mobile phone UI, expanding its capabilities beyond that of a keypad and function buttons. However, the iPhone changed the rules for writing mobile solutions in another way—the iPhone's Mobile Safari browser is a fully featured web browser. Its ability to view many Internet web pages and execute their scripts opens the possibility that the iPhone can handle certain corporate web-based applications. In this article, I examine the iPhone's web capabilities by porting a web-based problem-reporting form to it.

I admit when the iPhone was first pitched as a web-application platform, I was among many who clamored for the ability to write native iPhone applications. The phone runs a stripped-down version of Mac OS X, after all. Fortunately, after a year Apple has allowed us to do just that. In the interim, however, I've had an attitude adjustment in that I believe the iPhone can also serve—within limits—as a decent web-application platform. To understand my change of heart, some background information is in order.

Reports From the Field

The company I work for produces microcontroller units (MCUs). Ideally, these MCUs—and the development tools that write the embedded programs that execute on them—work perfectly. Realistically, problems happen. Much effort is put into collecting problem reports on the silicon and software, then routing this information to the appropriate engineering team. The team is responsible for correcting the problem, or devising a workaround. Often our field engineers transmit problem reports straight from the customer's site while the details are still fresh, and to get a team working immediately on a solution.

Currently, these problem reports, known as "Service Requests" (SRs), are filed electronically using a HTML/JavaScript web page that executes in a laptop's web browser. This scheme avoids application and vendor lock-in, and the code is relatively easy to maintain. This solution works well—as long as you can access the customer's intranet or WiFi network. However, customers might deny our field engineer access to their network for reasons of security. Using an IT-department-certified smartphone to bypass this connectivity problem by sending the SR through a carrier's cell towers is the obvious next step.

Several years ago, I was asked if the company's SR web page could be migrated to smartphones. A key concern was to avoid vendor lock-in. At the time, the choice was simple: Use Java Mobile to write the application. It would run on any smartphone or Blackberry with Mobile Java support. Due to other higher priority projects, the project was shelved.

Time passed and I got my hands on an iPhone. I was impressed at how many web pages it could display and interact with. The capabilities of its browser got me to wondering: How difficult would it be to port that SR web page to the iPhone? I decided to do a proof-of-concept rewrite of its code and find out.

Mobile Safari Overview and Behavior

First, let's get the nomenclature straight. While Apple calls an appropriately designed web page for an iPhone an "iPhone application," I'm going to stick with the term "web page" because in the end, no matter how well the web page integrates into the iPhone's look-and-feel, that's what it is.

Mobile Safari brings lots of features to the table for developers. It uses the same WebKit rendering engine that the desktop version of Safari uses. It's also compliant with a number of the latest web standards: HTML 4.01, XHTML 1.0, CSS 2.1 and partial CSS 3, Document Object Model (DOM) 2, and ECMAScript 3, a JavaScript standard. Mobile Safari also supports the AJAX XMLHTTPRequest remote scripting object. These features let the browser render most web pages accurately and manage interactive script sessions.

You probably noticed the word "most" in that last sentence. It's there for a reason. Mobile Safari doesn't support Java applets, nor (at this time) does it handle Flash content. The browser is unaware of the phone's filesystem, so there's no downloading of plugins or other files. However, it does support cookies.

Another difference is that in Mobile Safari, certain browser events have changed or have disappeared in order to support Apple's gestures interface. For example, scrolling through a web page's content by use of a finger flick requires that the iPhone's gesture

interface capture and consume events that might normally be construed as mouse events. Most mouseover events, if they appear in the browser at all, are now mapped to mousedown events. For the same reasons, the hover style is gone. If your web page uses mouseover events to implement menu choices, you need to rework the JavaScript handlers to respond to mousedown events or to clickable elements instead. On the positive side, the form and document elements produce the usual onblur, onchange, and onfocus events.

Finally, Apple's human interface guidelines dictate that iPhone web pages should be small, and focussed on doing one thing very well. There are valid reasons for this:

- The iPhone has a small screen. Cluttering it up with multiple windows or controls makes the web page's functions difficult to intuit or access. Nor does Mobile Safari support multiple windows, other than the temporary display of alert or dialog boxes over the main page.
- The network interface can vary in throughput. As the iPhone user interacts with your web page, the connection speed to the host can vary as the device moves between WiFi, 3G, and 2.5G wireless networks. Therefore, you must assume the worst-case scenario and design the page for the slowest network. Smaller pages load faster on slow networks, but this also limits the pages as to what they can do.
- The iPhone has constrained RAM and processing throughput. Therefore, Mobile Safari can't execute web pages with complex scripts. To appreciate this problem, point Mobile Safari to a heavily scripted web-based e-mail service and watch it take the page several minutes to render and respond. There's a reason the iPhone comes with its own native e-mail application.

In short, the iPhone's small screen, varying network speeds, and limited processing throughput dictates that you keep your web pages small and simple. You can link to other, separate pages, but they should also follow these guidelines. Lengthy page loads and with sluggish responses are only going to frustrate users so that they don't use your web application.

Design Considerations

The original SR web page was an HTML form that presents an array of drop-down menu elements, along with several text fields where you can enter information, as in Figure 1. Other fields display explanations that help guide users. The form relies on CSS to provide some styling, and it uses the display property to reveal or hide form elements, depending upon the choices the field engineer makes. The page gathers the information entered (such as the MCU type and the team assigned to the problem) and massages the data into a text-formatted e-mail that the company's CRM system can read. All the engineer does to complete the report is inspect the e-mail for errors, then send it to our CRM system, where the SR is logged into a database and the selected team notified.

My plan was to overhaul the SR's UI because the layout of the web page's elements were made for a large laptop screen, and not for a mobile phone. The iPhone has a 320?480-pixel screen, which is large by mobile phone standards, but still much smaller than a laptop screen. Rather than try to pack everything onto the small screen, I would instead display the array of choices as a list, where each list element serves as a link to a separate page. I wanted the web page's iPhone-based front end to gather the SR information into the same JavaScript variables used by the original SR form's back-end scripts. I could then reuse the field-tested back-end functions to reformat the data for the CRM system. Along the way, I expected to contend with the changes in the browser's event behavior, plus any quirks Mobile Safari threw at me.

Some initial tests with simple HTML forms showed that Mobile Safari could present the required text fields and drop-down menus that the SR page used. However, the layout of these forms on the iPhone's screen looked like something you'd see on a desktop browser and not at all like an iPhone app. For the sake of consistency, Apple promotes the idea of having your web page resemble regular iPhone applications as much as possible. Oddly, the company doesn't offer any framework to help you do this.

Because I didn't have the time or expertise to write such a framework, I did the next best thing—I did a Google search for one. It turns out that there are several: the iPhone UI (iUI), originally written by Joe Hewitt and now maintained by Google code (code.google.com/p/iui), the iPhone UI Universal Kit written by Diego Lafuente (www.minid.net/iphone), and WebApp.Net, written by Chris Apers (webapp.net.free.fr). I chose iUI because it's

Figure 1: A portion of the original Service Request form as it appears on a laptop browser.

fairly simple, and its creator wrote Firebug, a FireFox add-on that lets you debug JavaScript code in the browser (www.ddj.com/architect/196802787).

Framework Features

The iUI framework consists of a mixture of CSS styles and JavaScript functions. These provide a set of style selectors that you apply to your page's HTML elements. The framework then uses JavaScript to manipulate the page's document tree to modify its look. The appearance and behavior of these modified elements mimics that of the iPhone UI. For example, an ordered list of hyperlinks becomes the familiar iPhone list with arrows used to jump to other pages of a program.

There are style selectors for setting up a navigation bar, lists, and hyperlinked lists of information. Other styles help construct panels that contain controls, display information, or provide buttons. Other iUI features that assist you with writing iPhone web pages are:

- Visual feedback. Some elements flash briefly with the same blue color as the iPhone UI, thus verifying that the element has responded to the user's tap.

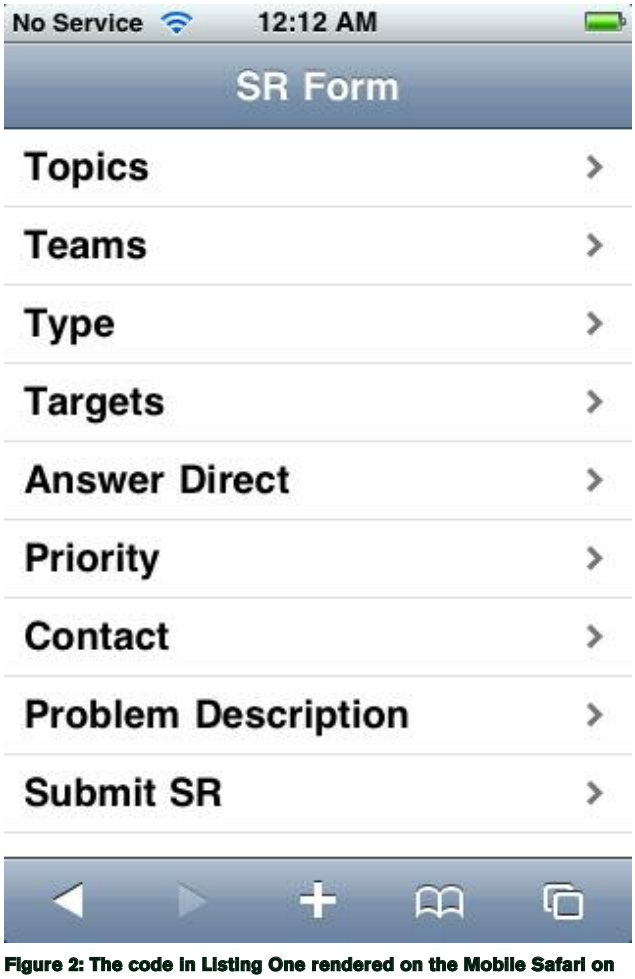


Figure 2: The code in Listing One rendered on the Mobile Safari on an iPhone.

- Conserve screen space. At the top of an iPhone web page is a bar that displays the page's URL. This URL text field bar consumes a precious 60 pixels of vertical screen real estate. You can temporarily hide the URL by invoking this code:

```
<body onload=
 "setTimeout(function() { window.scrollTo(0, 1) }, 100);"></body>
```

  This lets you recover those 60 pixels. iUI calls this code for you automatically.
- Handles changes in screen orientation. If the user flips the iPhone on its side, this action generates a change in orientation event that iUI responds to by re-rendering the web page with the new screen dimensions.

To use iUI, you reference its CSS styles file (iui.css), and JavaScript functions file (iui.js) inside your web page's header, bracketed by the appropriate *<javascript>* and *<style>* tags.

To recap, you write your iPhone page using HTML elements and tag them with iUI styles. After you put the interface together, you then write your own JavaScript functions that carry out the page's intent. When the page executes on the iPhone, it looks and behaves like an iPhone app, thanks to the iUI framework.

Writing and Testing Code

Because iPhone web pages render in a browser, you can use a standards-compliant desktop desktop browser such as FireFox 3.x, Safari 3.x, or Opera 9.x to write and test a page's code. The iUI framework makes the web page resemble an iPhone screen even on the PC, so you can also work the kinks out of your web page's layout before it is ever loaded onto an iPhone. Because IE 6 and 7 don't support many of the standard CSS selectors, iPhone web pages won't render properly in this browser. Simply put, don't use IE.

I started my code port using FireFox 3 along with Aptana's Firebug to help debug the JavaScript functions. While browsers are quite robust in rendering HTML with unbalanced or mangled tags, for a JavaScript error they simply abort and don't render the page, a behavior that can drive you mad. Firebug is really good at catching JavaScript errors, ranging from the simple typo to a reference to an object that doesn't exist, something that can easily happen when you're figuring out how to access objects on the document tree. You can step through the code and watch variable values change as the JavaScript executes, which is a good way of confirming that the code really works, rather than it just happens to work.

I was able to get a list display working quickly by using the toolbar style and writing a main list with the *<ul>* and *<li>* tags. Although this was only a proof-of-concept program, I then reworked the code so that JavaScript built the list dynamically, using a *for* loop for this purpose. I wanted to see if iUI was capable of handling a dynamically changing list, and the framework passed the test. Listing One(a) (available online at ftp://66.77.27.238/sourcecode/ddj/2008/) shows the HTML unordered list code that generates the main list, while Listing One(b) (also available online) shows the JavaScript that generates the main list dynamically. Figure 2 shows the rendered result from both listings.

Making a selection on this main list takes you to other screens with the relevant controls. For the processor, team, report type, and e-mail address where the engineer chooses an item from a list, I used the drop-down menus provided by the *<select>* tag. For the SR's priority level where there are several mutually exclusive choices, I opted to use radio buttons, which were handled by the *<input type="radio">* tag.

For those fields where text was entered, I used a form with tables for setting up the required fields. I could have used CSS to arrange these elements, but it was easier to reuse the table layout from the existing SR code. With some modifications, I got the subject and description fields, plus the character counter display to where it closely matched the original layout. See Listing Two (available online) for the code, and Figure 3 for the end result.

The submit screen was straightforward, as this was just some buttons within a form. I specified a bluebutton style for the Submit button, as the blue color indicates the preferred choice (much like an OK button on the Mac OS UI). I had to add some JavaScript code to handle the onsubmit event. This code invokes a verify function. For any missing information, the function displays an alert for the item in question and the submit process aborted. If all of the inputs were valid, the function instead packages the information for the e-mail's subject and body, launches the e-mail application, and drops the text into it.

Figure 3: The code in Listing Two rendered on the iPhone. While I made it resemble the original SR page (Figure 1), this isn't a requirement.

One minor problem I had with iUI was in accessing the content of various objects on the document tree. Although the iPhone web page appears as a main list screen with links that jump to other web pages with controls or text fields, the reality is that the application is just one big web page. The page anchors that comprise the main list point to other elements on the page, which might be more lists, controls, or forms. Because iUI modifies the document tree to make the designated elements appear as separate screens, accessing them through the document objects array was problematic. The solution was to assign the desired objects an ID, and then use the *getElementById()* method to access the object. For those elements that are part of a form, the usual techniques of accessing arrays on the document tree worked.

Acid Test with Mobile Safari

Once I got the web page working reliably on both the FireFox and Safari desktop browsers, it was time to try the web page where it really counted: Mobile Safari on the iPhone. I placed the program on a web site, downloaded it to the iPhone, and tapped away at the screen. When I tapped Submit, the verify function insisted I hadn't made any choices. Because the code's logic checked out on the desktop browsers, I knew the problem was rooted in changes to browser events for drop-down menus, since I relied on onchange events to transfer the choices into variables.

However, after some tests, I discovered that the drop-down menus weren't generating any events at all. I went to iPhoneDevWeb, a Google group where developers discuss issues with iPhone web page development. I spotted a report that explains where a bug in Safari prevents drop-down menus from producing events if they have a size attribute. It took only seconds to delete this attribute with the editor, and the drop-down menu choices began working.

The next issue I had was with the radio buttons. Rather than small circles, they were stretched across the screen. By adding table cells, I could get the buttons to appear less elongated, which hinted that something in iUI was the culprit. A query on iPhoneDevWeb got me a response in about a day. This was a known problem with the input selector in iUI's CSS file. A suggested workaround was to modify the CSS to position the button with margins. I decided to go in search of custom radio buttons instead, as the typical browser radio button doesn't hew closely to the iPhone interface.

Soon I found Checkbox and Radio Input Replacement (CRIR) written by Chris Erwin. This is more CSS and JavaScript code that generates custom buttons and checkboxes. I dropped the radio button code into the program and got it working in short order. The only changes I made were to replace the small, light-green button

image with the blue button image that appears in the Apple Mail application. I haven't had any problems with iUI and CRIR's styles and code creating conflicts. You can download CRIR at http://www.chriserwin.com/scripts/crir/.

Finally, when I attempted to enter text into a description field, the virtual keyboard kept disappearing after I typed a character. This I traced to the character count routine, which removed the focus from an input text field, updated the character count field, and reestablished the focus on the text field. This sequence of events interfered with the virtual keyboard's operation. The solution was to modify the counting function so that it didn't alter the focus.

After these hurdles were cleared, I was able to tap in various choices, hit submit, and the iPhone's Mail application appeared, with the message ready to go. The SR form's code could stand some improvements, but it demonstrates that the iPhone would be a suitable platform for making field reports. All that remains to do is add the back-end formatting code, but that will wait until the IT department approves the iPhone as secure enough for corporate work. For more tips on writing iPhone web pages, see the sidebar, "Steps to Writing an iPhone Web Page."

Better Browsing on Mobile

The partial port of the SR web page shows that it's not difficult to move a subset of a company's web-based applications onto the iPhone. I was able to locate and use an off-the-shelf framework and custom JavaScript software to impart an iPhone look and feel to the SR web page. Would using an iPhone for SR reporting risk vendor lock-in? Not really, which brings us back to the point I made at the start of this article. Apple's iPhone has raised the bar on what's acceptable for a smartphone browser. Mobile users will demand that their browsers be fully capable of viewing the Web—albeit painfully slow at times—and executing lightweight web applications. Over time, the performance of both the platform and JavaScript will improve; thereby expanding what a mobile phone is capable of doing for us. It should come as no surprise that Apple's added SquirrelFish, a high-speed JavaScript interpreter, to the WebKit rendering engine.

You can expect other smartphone vendors to improve the capabilities of their browsers as fast as they can. When that happens, then we'll run web apps on any phone that we choose. That's an outcome that will be good for all of us.

— Tom Thompson is the head of Proactive Support for embedded products at Freescale Semiconductor. He can be reached at tom_thompson@lycos.com. The views stated in this article are the author's, and don't necessarily represent Freescale's positions, strategies, or opinions.

Steps to Writing an iPhone Web Page

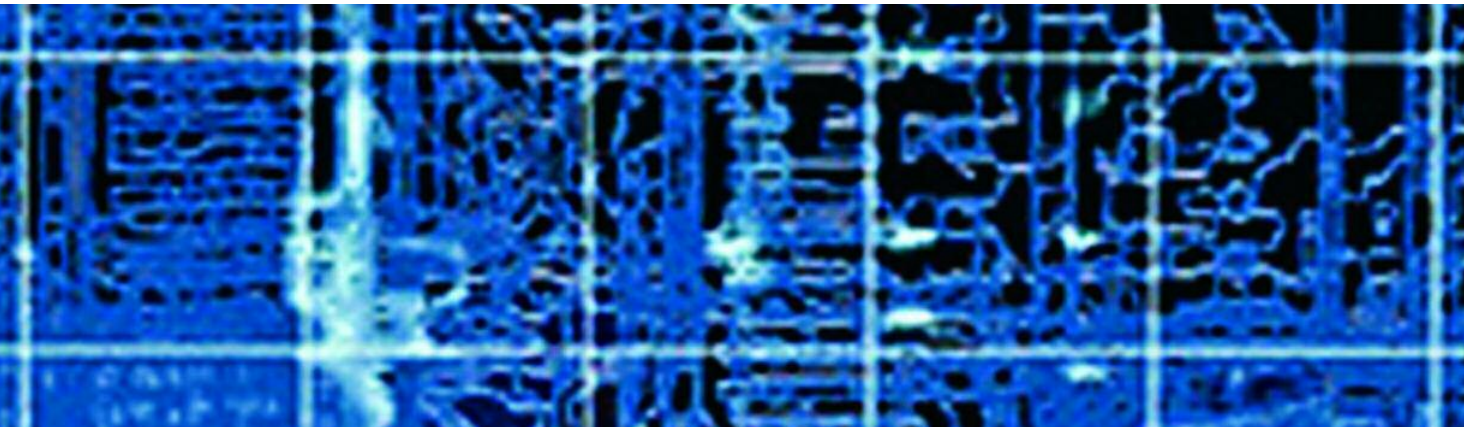
1. Prototype and write the page's code on a standards-compliant desktop browser, such as FireFox 3 or Opera 9.x I prefer FireFox because then I can apply Aptana's Firebug to debug the JavaScript. Note that for these two browsers that iUI's buttons don't render well, but it's good enough for code testing.

2. If you haven't already, modify the application's interfaces so that they don't use mouseover events.

3. Test and fix the page with the desktop version of Safari 3.x. The iUI buttons render fine on this browser. Clean up any quirks with events or side-effects to CSS selectors.

4. Do final test and revisions on the iPhone. Be prepared for another round of fixes for quirks and CSS side-effects. Also, you may have to tweak the interface for the small screen. Don't forget to reorient the phone and verify that you haven't hard-coded the screen positions of any of the application's elements or controls!

Return to Table of Contents



Mobile Widgets and the Internet Experience

Widgets come to mobile platforms in a big way

By Deirdre Blake

Craig Cumberland is director of technology and application marketing for software platforms at Nokia.

DDJ: Craig, is there a difference between "Web 2.0" and "Mobile Web 2.0"?

CC: There are lots of people out there trying to define "Web 2.0," so I won't add to that confusion. However, I do believe there is no distinction between the Internet and the "Mobile Internet," as it's often referred to. The Internet is the Internet, and whether you access it from a mobile device or a desktop or a gaming console, it is the same content. So, given that logic, Web 2.0 is conceptually the same and there would be no mobile version of Web 2.0. It's a long answer to a direct question, but it's an important distinction to make.

DDJ: What role do widgets play in this?

CC: I view widgets as an application bridge between two paradigms—the "Internet Access paradigm" and the "Internet Interaction paradigm."

That is to say that, prior to widgets, access to the Internet and its content has been about starting up a browser application and then accessing the desired content or service.

Widgets change that because the browser application is no longer the gateway, structuring access to content. Widgets allow for the creation, by developers and even direct consumers, of Continuous Content Experiences.

DDJ: Widgets have been around a long time—but on the desktop. So what's the big deal about mobile widgets?

CC: You're right, widgets have been around for quite awhile for the desktop, and you're seeing them emerge as additions to personal homepages and webspaces like Facebook, and so on.

But in a desktop environment, resources such as network bandwidth, processing power, and memory are not constrained in any way. So opening 5, 10, even 20 browser instances, tabs, windows...and slipping between them all to access and interact with a variety of content is the norm. And so widgets on desktops don't present a significant improvement in the content experience with the Internet.

However, widgets, as directed content experiences, have an immense benefit for mobile users because they allow people to personalize Internet content and interaction into lightweight web applications and stay current and active with the Internet content that matters to them.

DDJ: What's one of the coolest mobile widgets you've seen to date?

CC: Mobile widgets are coming to S60 devices shortly, so I can't really comment on any specific widget yet. But some very cool ones under development include travel-related widgets, such as flight trackers, currency converters, local weather services, and the like. You can imagine that in concept, anything you can design with standard web technologies, you can create a widget to enhance that interaction—taking into account the screen size of mobile devices. Examples can range from local traffic information while out on the road, to chat services, to following stock performance, or keeping tabs on your personal Internet auctions.

DDJ: Using Nokia's S60 as an example platform, how do you build widgets? What development environments? What tools?

CC: S60 widgets can be developed with a wide range of tools, from the S60 SDKs, which are available for download from www.forum.nokia.com, to web editors like Firefox's Firebug HTML editor tool and even simple text editors—depending on how bit-driven you really are. Any tools that can be used to write standard HTML, CSS, and JavaScript apps can be used to build widgets.

DDJ: Can mobile widgets participate in mash-ups?

CC: Absolutely, as mash-ups are primarily combinations of interactions with various Internet content and services, they would be great candidates for widgets. And as we are just starting to see mobile widgets emerge, I think mash-up widgets are just around the corner.

DDJ: Security is an obvious concern. What kind of safeguards can be put in place for widgets?

CC: Security is something we take very seriously and are determined to be active in the development of security controls and preventive measures. Widget support for S60 3rd Edition Feature Pack 2 devices does not create additional security risks as these widgets work in a very contained subenvironment. This is often referred to as a "sandbox" security model that provides a tightly controlled set of resources for foreign programs to run in.

DDJ: Do you expect to see a central clearinghouse for mobile widgets emerge?

CC: I believe distribution of mobile widgets will follow a number of different paths, much the same as we have seen on the desktop and in web-based environments. But in addition to those methods, I'd expect to see carriers distributing them as part of key services, other vendors in the mobile ecosystem offering them, even potentially phone-to-phone sharing of them.

Return to Table of Contents

The iPhone Isn't Easy

How to get started building an app

By Tom Thompson

With more than 20 years of software development under his belt, Ray Floyd was ready to tackle the next great platform, Apple's iPhone. He'd developed for the Macintosh, although not recently. He was familiar with Objective-C, and he had developed software for other mobile platforms. With a software development kit in hand and up to 75 million potential customers, Floyd couldn't wait to roll out his first iPhone app.

Two months later, he was still waiting, and his enthusiasm had largely changed to exasperation. Why? Because writing native applications for the iPhone platform can an intimidating process.

For one thing, as Floyd quickly discovered, there is a huge amount of information to digest before you can start coding. The iPhone/iPod Touch/iPad is a new platform in its own right. These are sophisticated general-purpose, handheld computers. And as a platform, it has its own operating systems, and a host of APIs (some new, some recycled) for programming. Not surprisingly, the documentation on operating systems and APIs spans hundreds of pages.

Even knowing the Mac isn't a free pass. The iPhone operating system is a variant of Mac OS X, but slimmed down to function on a mobile device. As such, a lot of familiar OS X APIs, such as those that manage a keyboard and mouse, are absent. This means that even experienced Mac developers, which Floyd didn't consider himself to be, may not have the features and facilities they're used to having.

The APIs are based on Cocoa, an object-oriented application framework used to write Mac OS X apps. Like the operating system, the Cocoa frameworks have been stripped to the bare essentials for the iPhone OS, and lightweight touch interface APIs added. This minimalist version of Cocoa, termed "Cocoa Touch," provides another hurdle.

To use the APIs, you need to learn Objective-C. Objective-C is an object oriented superset of C, and while it has many valuable features, its idiosyncratic syntax means it'll take a lot more than a cursory study to make sense of code. you won't make makes a cursory study of an example program's source code unfathomable. The language's syntax is quite different from C++, and until you have a grasp the syntax, example code will appear almost unreadable.

Like Floyd, I was initially lulled into thinking iPhone development was probably simple for experienced developers. How else could you explain the many thousands of iPhone apps available on Apple's App Store? In fact, many of these apps — and games written in C and C++, in particular — have been ported to the iPhone, where an Objective-C based wrapper interfaces to the Cocoa Touch frameworks, while the game engine remains written in C and C++. In other words, since the Apple Objective-C compiler accepts C and C++ code, you don't have to start entirely from scratch.

But as I began researching iPhone apps, I grimly realized even my background in writing INITs and FKEYs for the classic Mac OS and J2ME programming for various mobile phones was of little use. The iPhone platform, its OS and APIs, and the peculiarities of Objective-C were radically different. I decided to break the task up to reduce the steepness of the learning curve, hoping to find a magic bullet along the way that would streamline development.

First, what should my first app do? I wanted a specific project so I could focus on a subset of APIs, and not become overwhelmed by the big picture. I settled on porting the SpaceActivity app I had written for the Android platform some time ago (see "The Android Mobile Phone Platform"; http://www.drdobbs.com/mobility/210300551).

The app displayed a spaceship you steered with handset buttons. That iPhone APIs would be utterly different from Android's was a given. I'd have to learn how to draw the ship on the iPhone's screen, then figure out how to implement controls for it. I hoped that SpaceActivity's "core code" — the physics routine that calculated the ship's velocity in response to rocket thrusts, plotted its position, and kept the image on-screen — could be reused. This code was debugged and tested and had the virtue of using a minimum of API calls. It also had ready-to-use images of the spaceship with a transparent background, so I didn't have to design any new graphics. Finally, it would also be an interesting test of porting Android code. The complete source code for SpaceActivity is available at http://i.cmpnet.com/ddj/images/article/2010/code/iphone_space.zip.

Variations on C

The iPhone runs a stripped-down version of Mac OS X and uses a subset of the Cocoa application frameworks, along with some new ones, to support its touch interface. These revised frameworks are known as Cocoa Touch and comprise the iPhone platform's APIs. (For further information on the iPhone platform, see "Smartphone Operating Systems: A Developer's Perspective"; http://www.drdobbs.com/mobility/216300179). OS X programmers will have an advantage here in that they are familiar with the programming language and frameworks. There are some differences between the frameworks, however. For example, the primary Cocoa framework for OS X is AppKit, while on the iPhone the primary Cocoa Touch framework is UIKit. UIKit, as its name implies, consists of lightweight UI classes tailored for the resource constraints of a hand-held device. Also, the keyboard and mouse interfaces are absent. I'll point out more differences between the two platforms as I go.

Again, you write an iPhone with Objective-C. Objective-C consists of extensions to the C that implement many OOP techniques, such as object encapsulation, inheritance, and polymorphism. Because the language is a superset of C, Apple's Objective-C compiler accepts C and C++ source, which makes it ideal for reusing existing code written those languages. In fact, some games ported to the iPhone consist mostly of C code, and Objective-C code is used only to interface with the Cocoa Touch frameworks. For example, id Software's Wolfenstein 3D takes this route, since its core logic was written in C. John Carmack, the game's creator, was able to write Objective-C wrapper code that communicated with the iPhone's APIs, while the complex pieces of the game, such as the game's ray-casting logic and its rendering engine, were left intact. Carmack has made the source code of Wolfenstein 3D available for download from the id Software website (http://www.idsoftware.com/wolfenstein-3d-classic-platinum/).

Objective-C has its own language idiosyncrasies for defining objects and implementing other OOP-based design patterns. How you invoke methods differs, for example. In Objective-C, you don't "call" a method, rather you send a "message" to the object that owns the method, like so:

```
[shipView upDateShip:kNoThrust];
```

The content within the square brackets contains the gist of the message. To wit, this message is directed to the *shipView* object, and invokes its *upDateShip* method, while supplying it with an argument that consists of the constant value *kNoThrust*. Another important difference is that for certain complex UIKit classes you don't extend their behavior by subclassing them and writing override methods. Instead, you edit the supplied "delegate" class that implements the desired behavior. For example, you do not extend the *UIApplication* class that implements the core application functions. You modify and add overrides to a *UIApplicationDelegate* class. Messages unknown to the *UIApplication* object are routed to your *UIApplicationDelegate* object, where your custom methods can respond to them. You can get detailed information on Objective-C from the Objective-C 2.0 Programming Language manual, and more information on UIKit and message passing from the iPhone Application Programming Guide (http://developer.apple.com/programs/iphone/). Both of these documents are available on the Apple iPhone developer site. Given all of the changes implemented by Objective-C and UIKit, the structure of an iPhone app is an interesting mixture of the old and new. When the iPhone OS launches an app, it calls the standard C function, *main()* (Listing 1). However, all that *main()* does is set up a memory pool for your app and optionally parses any *argc* and *argv* arguments.

```
LISTING ONE
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

However, here the resemblance to C ends: *main()* immediately calls *UIApplicationMain()* function, which generates an instance of the *UIApplication* class. It also creates your delegate object, which is your modified *UIApplicationDelegate* class. *UIApplication* then starts receiving events from the iPhone OS and sends them to your delegate. For example, when your delegate receives an *applicationDidFinishLaunching* message, this signals that the app's runtime environment is established and running. You respond to this message by executing the delegate's *applicationDidFinishLaunching* method, which contains custom initialization code, and creates any required objects (Listing 2).

```
LISTING TWO
#import "SpaceAppDelegate.h"
#import "SpaceViewController.h"

@implementation SpaceAppDelegate
@synthesize window;
@synthesize spaceViewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application
{
          application.statusBarHidden = YES;

// Make instance of our controller
          SpaceViewController *aViewController =
[[SpaceViewController alloc]
                    initWithNibName:@"SpaceView" bundle:[NSBundle
mainBundle]];
          self.spaceViewController = aViewController;
          [aViewController release];
// Restore state if necessary
          [spaceViewController restoreState];
// Add view managed by controller to main window
          [window addSubview:spaceViewController.view];
    [window makeKeyAndVisible];
}
// Application will be terminated soon, save state data
- (void)applicationWillTerminate:(UIApplication *)application {
          [spaceViewController setMode:PAUSE];
          [spaceViewController saveState];
}
// Release all objects
- (void)dealloc {
          [spaceViewController release];
    [window release];
    [super dealloc];
}

@end
```

Another difference from C is that the iPhone APIs don't provide a method that allows you to terminate an application. Instead, your

app receives an *applicationWillTerminate* message, which signals that the iPhone OS is preparing to shut the app down. In response to this event, your delegate executes an optional *applicationWillTerminate* method that saves the app's data context (if required) and tears down any storage that it set up in *applicationDidFinishLaunching*. This routine should execute quickly, as the iPhone OS usually terminates the app within a few seconds.

Now that I had a grasp of Objective-C and an idea how the iPhone OS interacted with an app, my next step was to figure out how to draw something onto the device's screen. That meant delving further into the frameworks.

A Window...With Views

An iPhone app displays UI elements and content on the device's screen, and the user responds by tapping the screen or making gestures across it. A singleton window object and multiple views are the visible components that manage the iPhone's UI and any content display. The window object encompasses the entire screen and lacks a border and a title bar with controls. The window and views are implemented by instances of the *UIWindow* and various *UIView* classes, respectively.

The *UIWindow* class is actually a subclass of *UIView*. However, the window instance serves as the root container for all of the other views, and it is always the size of the screen. It serves as a backdrop on which you add any number of views and subviews. The *UIApplication* object sends events to the window object, which in turn delivers them to the appropriate view objects.

An iPhone app typically generates an instance of the window object immediately, using information stored in a NextStep Interface Builder (nib) file. Nib files describe the window's dimensions and other information necessary to make an instance of the window. I'll have more to say about nib files later.

Alternatively, you can generate the window instance from code that you write in the delegate. Once the window is set up, you then add the views that make up the app's content and controls as subviews to this window. The *UIView* class has methods that allow you to add, remove, or change the order of how views are presented in the window. After you add the app's views, often you are done with the window and deal only with events delivered to the views.

You can see some of how this setup is accomplished in Listing 2. The generation of the window is done for you automatically if you use a nib file, as is the case here. All that you need to do is provide a pointer, termed window in the listing, which references the window instance. Apple's XCode development tools automatically generate the window nib file, MainWindow.xib, along with a *UIApplicationDelegate* header file and method file for you.

Views, like the window instance, can be generated either from a nib file or through programming. The choice of which option to use depends upon how dynamic you intend the view's UI elements to be. For example, a help display with information presented in a static layout is best handled with a nib file. If you have controls that change in response to the user's interaction, you may want to use code to generate a portion of the views and then manage their position and appearance. Likewise, a view with the spaceship needs to move about the screen, so it should be generated by code, and not by a nib.

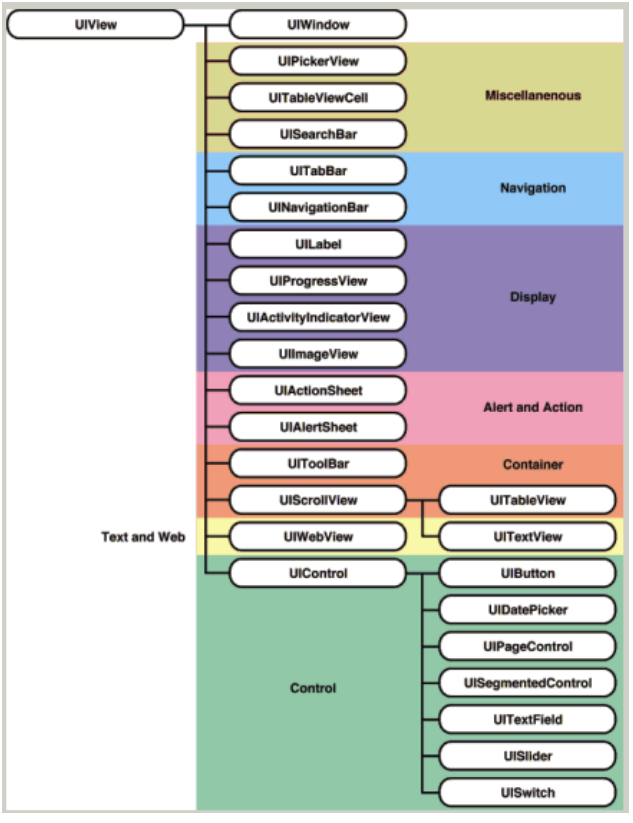


Figure 1

Besides the *UIWindow* class, *UIView* has a number of child classes that provide useful capabilities (Figure 1). A control group of views implements visual elements such as buttons, sliders, progress indicators, switches, and text labels. Another group serves as containers for dynamic content such as pages or tables. A navigation group provides tab bars and navigation bars that facilitate moving swiftly from one screen of content to another. Other subclasses support scrolling of content, and a toolbar for grouping items.

Views also display content, which might be text, vector graphics, rendered HTML material, or images. For games, you can use cross-platform OpenGL ES (http://www.drdobbs.com/cpp/187203532) APIs to draw and render 3D content, but the OpenGL drawing region must be presented within a view. My program's goals were modest: I only wanted to paint pixels on the screen and move some of them about. The basic *UIView* class would be suitable for this purpose.

The origin of a view's coordinate system starts at the upper left corner and the axes extend to the right and downward; see Figure 2.

Interestingly, for OS X, the situation is reversed: The origin is at the lower left corner and the axes extend to the right and upwards. In fact, the vector graphics APIs, Quartz (http://lists.apple.com/archives/QuickTime-API/2007/Jul/msg00129.html), still use this orientation in the iPhone, but UIKit automatically corrects the orientation for you before rendering the graphics calls. Coordinates are

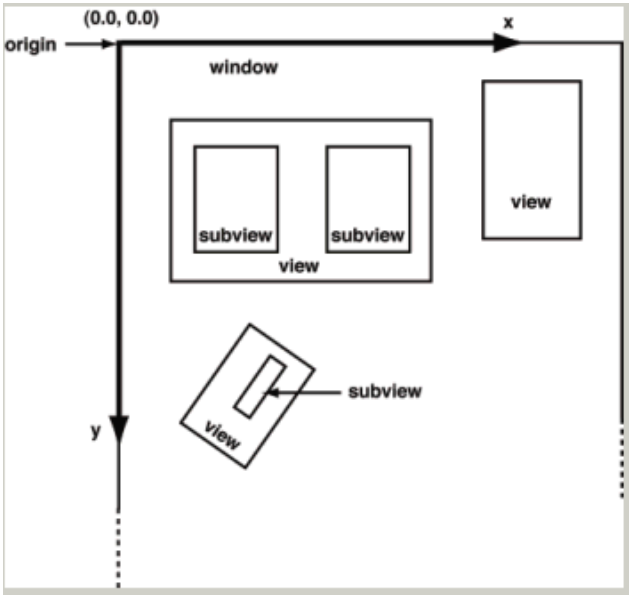


Figure 2

specified in floating-point numbers for device-resolution independence. A frame describes the view's rectangular area, and its midpoint is defined by an instance of point (a structure with *x* and *y* values) termed center. The center point information is handy to have when you move or rotate a view — particularly when the view might contain a spaceship image.

Views can be nested in one another, to form complex visual elements, as Figure 2 shows. In addition, views have characteristics known as properties, and the center point is one. Another useful property is alpha, which manages the view's alpha channel. This property can be modified to apply transparency effects to the view's contents. There is also a visibility property, hidden, that can be used to hide or reveal views.

Rotating an image is handled through the view's transform property. This property contains a transformation matrix that can be applied to the view. For me, a transformation that did a simple rotation would be sufficient. However, more complex transformations can be applied to a view's contents, and UIKit provides a number of predefined matrices that can be used to generate special effects with the view's content, or when switching to another view.

At this point I understood the UIKit framework adequately to know what classes I needed to build the basic core of an iPhone app. This core (an instance of *UIApplicationDelegate*) would start by making the required window object, to which I would add a view for a background, and another view that contained the spaceship image. A transformation matrix would rotate the spaceship view. I was a bit fuzzy on the details as how to control the spaceship, but I figured that another view could display a virtual joystick or control pad. That view would then react to touches on it. So far, so good.

It's All About Controllers

As I brought the details of implementing UI controls into sharp focus, I discovered that while I'd be still have the app add the views to a window, there was an intermediary involved: a *UIViewController* object. This object, known as a view controller, supports the OOP-based model-view-controller (MVC) scheme used by iPhone apps. In this program design, a model object manages the app's data and implements its custom logic, while the view displays the model's data. The controller acts as an intermediary that manages user interactions detected by the view, and has the model respond to them, either by editing or updating its data content. In addition, the controller then has the view refresh its content with the model's revised data. MVC keeps the app's unique data and behaviors within the model object and separate from the view assigned to display its data. Can you say encapsulation?

My planned app had only one screen where the spaceship would appear, plus a help screen. This arrangement could be neatly managed by a single instance of the generic *UIViewController* class, which would implement the physics model and jump to the help screen and back. While it's possible to implement all of the app's logic within a view object or in other ways, I found it better to follow the MVC design pattern that the Cocoa Touch frameworks support.

Unlike the *UIApplication* class that requires you edit a delegate object, the *UIView* and *UIViewController* classes can be subclassed so that you can extend their behavior with custom code. The custom code in the view controller would receive touch events from the view representing a virtual control pad, and convert them to commands that would affect the physics of the spaceship. The controller would then update the position and appearance of the subview displaying the spaceship. To recap the program design, both the control pad and spaceship would be subviews within the background view. This view in turn would be attached to the mother-of-all views, the window object. The program's design was coming together.

Apple's XCode development tools, as mentioned previously, provide ready-to-use iPhone application templates to help get you started. I launched XCode, and in its New Project dialog, chose the Cocoa Touch frameworks, and then chose an app template that was window-based. This I named *SpaceAppDelegate*. While there were templates that generated a view-based app template, I wanted to add the view controller and view classes as I went.

Next, I added a view controller class. This was just a matter of selecting New File in XCode, followed by choosing the *UIViewController* class from the dialog. I named my view controller *SpaceViewController*. XCode generates the basic header file and method file for the class. The latter has stub routines for some of the class methods, allowing you the opportunity to add custom code.

The first thing the view controller would do is generate a view that covered the app's window and display an image of a star field

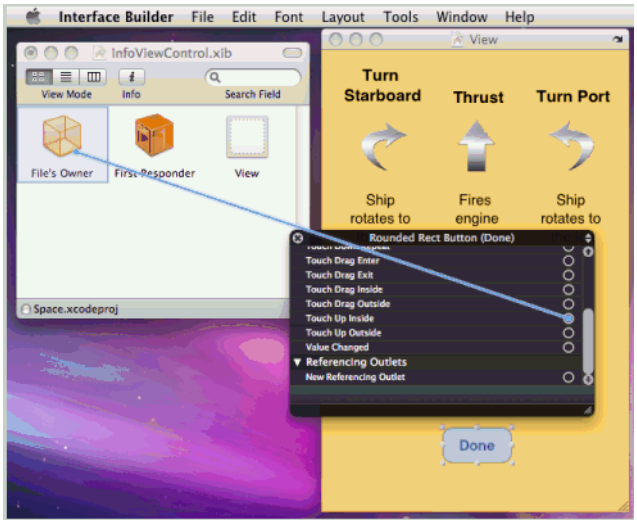


Figure 3

as a background. Since the background was a static image, I decided that storing the view and its image in a nib file would be a good idea. It was, but getting a properly made nib file that didn't crash the app (the phone itself wasn't affected) was a major problem until I sorted things out.

Interface Building

To make nib files for your app's visual interface, you use the appropriately named Interface Builder program, often just called "IB." It provides a number of preconfigured objects such as UIImageViews (that display images), scrolling views, buttons, sliders, and other visual controls. You drag and drop these objects from IB's library window onto a mockup of the view to assemble the app's UI. IB thus allows you to design the visual elements of the app's UI without worrying about its underlying code. This fits nicely into the MVC design pattern, as IB keeps most of the UI elements uncoupled from the app code.

However, IB does much more than assemble an app's UI. The visual elements that you place into a view's layout become full-blown objects when they are loaded. (Apple characterizes the information stored in the nib file as "freeze-dried objects".) Put another way, the XML generated by IB that describes the object and its properties can be reconstituted at run time into actual UIKit objects. For example, that button you place in a view becomes a *UIButton* object that can detect a touch event on it. Furthermore, it responds by highlighting itself — no action on the part of the programmer is required. More important, the button object can send one of many predefined messages to the application. In short, when you use IB to put together the app's UI, you are not only constructing its visual interface, but also its underlying event and messaging apparatus that communicates with the program.

IB lets you route the messages generated by these UI elements to specific destinations, which in my case are methods in *SpaceViewController*. This is accomplished by clicking and dragging a link representing a message generated by the UI object to the name of a method in the receiving object. For example, for the view representing a button, you Control-click on it, and a menu of possible messages that can be generated in response to the action appear. You next click on the desired message link and drag it to the File's Owner icon, which represents the view controller receiving the messages; see Figure 3. A pop-up menu of methods appears, and when you select one, it appears in the Connections browser window. The visual metaphor used to match UI messages to methods is a powerful one, and makes coupling the UI's actions to specific program responses easy to do. Having said that, IB's assignments aren't foolproof or bulletproof. Mistakes can be made, and I made a good one at the outset.

I started my foray into nib files by launching IB, and in the New File dialog selected a view, and named it *SpaceView*. I made it the size of the iPhone screen (320 x 480 pixels). From the library window, I dragged a *UIImageView* onto the view, and made it the same size. From the object inspector, I assigned *UIImageView*'s content to be an image of the Horsehead Nebula that I had trimmed to the screen dimensions and saved in a file named Backdrop.png. With the view thus set up, the existing stub code in the view controller should be sufficient to load the nib and display the nebula image in the iPhone simulator. Displaying the image would make for a good initial test.

No such luck. My trial app died a horrible death, usually with an uncaught exception.

It took me a while to track down the culprit. In IB, if you botch specifying the object that a view sends its messages to, bad things happen. That's because Objective-C's message-forwarding and delegation mechanism builds the links between objects and methods at run time. Therefore, if you fail to specify say, a view controller as the message recipient, the result is that the iPhone OS loads a view with broken links. When the view attempts to pass messages to the view controller, they go spinning into oblivion, carrying your app with them.

The gotcha is in how IB sets the destination object for the view's messages. IB uses a placeholder object, called the File's Owner, to represent the destination object. You've met this object already, and can see it in the IB document window for SpaceView in Figure 3. Put another way, the File's Owner information designates the object that "owns" the interface, which means that it is the message recipient. Stated another way, the owner is the instance of the class that loads the nib. For my program, that would be *SpaceViewController*. Granting ownership isn't accomplished by pointing and clicking in IB as was the case with the other assignments. This is done by typing the class name into IB's Identity Inspector. I had overlooked doing this. Once I entered the class name and rebuilt the project, the background image of the nebula appeared in the iPhone simulator. I was one step closer to my goal. For more information on IB's pitfalls and how to have problem free interface construction work, see the sidebar, "IB Tricks and Traps".

Views Within Views

Unlike the static scene in SpaceView where I used a nib file, I decided that the view displaying the spaceship should be generated by code. The reason is that it is a dynamic object, moving about in response to data from the physics engine. The view's image would also change in appearance, depending on whether the spaceship engine was firing. I created a simple view, *ShipView*, to manage the rocket's appearance (Listing 3). Two images, rocket_firing .png and rocket.png, would show the ship with its engine either firing or not. The method *upDateShip* changes the view's appearance by loading one or the other of these images. Besides drawing itself, *ShipView* doesn't do anything else: *SpaceViewController* is responsible for rotating the *ShipView*, and plotting its position. Once these updates are done, *SpaceViewController* directs *ShipView* to redraw itself, so that appears in the new position, orientation, and with the proper image.

```
LISTING THREE
#import "ShipView.h"

@implementation ShipView

@synthesize shipImage;
@synthesize shipThrustingImage;
@synthesize shipNotThrustingImage;

- (id)init {
          // Retrieve the images for the view and determine the
size of one
          shipNotThrustingImage = [UIImage
imageNamed:@"rocket.png"];
          shipThrustingImage = [UIImage
imageNamed:@"rocket_firing.png"];

          CGRect shipFrame = CGRectMake(0, 0,
shipNotThrustingImage.size.width,
shipNotThrustingImage.size.height);
          // Set self's frame to encompass the image
          if (self = [self initWithFrame:shipFrame]) {
                    self.opaque = NO;
                    shipImage = shipNotThrustingImage;
          }
          return self;
}

- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame]) {
        // Initialization code
    }
    return self;
}

- (void)upDateShip:(int)imageSelection {

          if (imageSelection == kNoThrust) {
                    shipImage = shipNotThrustingImage;
          }
          if (imageSelection == kThrust) {
                    shipImage = shipThrustingImage;
          }
}

- (void)drawRect:(CGRect)rect {

          // Drawing code
          [shipImage drawAtPoint:(CGPointMake(0.0, 0.0))];
}

- (void)dealloc {
          [shipImage release];
          [shipThrustingImage release];
          [shipNotThrustingImage release];
    [super dealloc];
}

@end
```

Creating the virtual control pad was a bit tougher. One of the best features about the iPhone is that its touch screen allows you to design the UI to suit the needs of your app. One of the worst features of the iPhone is that you have to design the UI. You don't want come up with an unwieldy interface that could frustrate a user. However, the first order of business was to figure out to respond to the touch events received by the control pad's view. The intuitive interface could come later. Although the control pad's position was fixed, I designed the *ControlPadView* class, like *ShipView*'s, to be generated through programming. This allowed its appearance to change in response to a touch, and to potentially support a user's preference to change its position. *ControlPadView*'s code was much simpler than *ShipView*'s, as it only had to load one image. For a control pad image, I started with three buttons: a left one to turn the ship starboard, one in the middle to fire the rocket, and the third to command a turn to port. A view responds only to touch events that fall within its frame, and these events correspond to several types of messages. There's *touchesBegan*, which is sent when you place your finger on the screen, and *touchesEnded*, which is sent when you lift your finger from it. A touchesMoved message is sent when the iPhone OS detects finger motion across the screen. If you touch the screen and release it quickly, the iPhone OS interprets the two closely spaced events as a tap and advances a counter. This counter can be read to determine how many times the screen was tapped. With this set of events, I could implement the control pad and other actions, as in Listing 4. For a *touchesBegan* event, I first determined if it was on the *ControlPadView* or a view with an information button. If the touch occurred in *ControlPadView*, I next obtained the touch's coordinates within the view to determine what button had been touched. I had designed the buttons to fill the entire view vertically, so all the code had to do was compare the horizontal displacement with specific constants to determine which button was "pushed." For the two steering buttons, a *shipTurning* variable is set to signal a turn and its direction. A touch on the thrust button sets a variable named thrust. When *ShipView* redraws itself, it would display the image of the ship firing its engine. A touch on the information button loads the help view.

```
LISTING FOUR
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{

          UITouch *touch = [touches anyObject];
          if ([touch view] == controlPad) {

                    location = [touch locationInView:controlPad];

                    int disp;
                    disp = location.x;
                    if (disp <= kPortPad) {
                              shipTurning = kPortTurn;
                    } else if (disp > kPortPad && disp <=
kFireEnginePad) {
                              thrust = kThrust;
                    } else if (disp >= kStarboardPad) {
                              shipTurning = kStarBoardTurn;
                    }
                    controlPad.alpha = 0.75;
                    return;
          }
          if ([touch tapCount] == 2) {
```

```
                    [self doReset];
                    return;
          }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{

          thrust = kNoThrust;
          shipTurning = 0.0;

          [shipView upDateShip:kNoThrust];
          controlPad.alpha = 0.5;
          infoIcon.alpha = 0.5;
}
```

To provide visual feedback when the control pad was touched, I cranked up *ControlPadView*'s alpha channel property up so that the buttons would be less translucent and appear to light up. Finally, just in case things went horribly wrong with the program, I had *touchesBegan* watch the tap counter for two taps anywhere on the screen, which would reset the physics engine and the app's operating mode.

When the user lifts their figure, the touchesEnded method cleans up by clearing the shipTurning and thrust variables, and having *ShipView* update its image. I also reset the alpha property of the *ControlPadView* back its default value, which made the buttons partially translucent again. The ability to manipulate the properties of views, particularly their alpha channel and visibility, allows you to craft a slick interface with little effort and code.

Now it was time to put things together. The *UIApplicationDelegate* loads, and it in turn loads the main window nib and *SpaceViewController*. *SpaceViewController* in turn loads *SpaceView*'s nib and receives a message, *viewDidLoad*. This message signals *SpaceViewController* that *SpaceView* loaded without trouble. I used the *viewDidLoad* method to add the views of the spaceship, the control pad, and info button into *SpaceView*. The order in which the views are drawn is important, because you can layer views over one another. By drawing the ship first, it would be under the control pad and info button, but this wasn't a problem because their alpha property made them translucent. The *UIView* class also provides methods for changing the order of views on-the-fly if your design requires it. The last thing *viewDidLoad* does is make an instance of *NSTimer*. It implements a periodic timer that calls the game loop method every 0.075 seconds.

The Loop That Isn't and Modes

The master method that drives the app and periodically updates its interfaces is — appropriately enough — called *gameLoop*; see Listing 5. Note that the method is a loop in name only: it calls housekeeping methods that update the physics model and the screen, and exits. The *gameLoop* method keeps executing because of the timer.

```
LISTING FIVE
-(void) gameLoop {
          if (programMode == RUNNING) {
                    [self updatePhysics];

                    [controlPad setNeedsDisplay];
                    [infoIcon setNeedsDisplay];
                    CGAffineTransform shipRotate =
CGAffineTransformMakeRotation(shipHeading / 180.0 * M_PI);
                    [shipView setTransform:shipRotate];
                    [shipView upDateShip:thrust];
                    [shipView setNeedsDisplay];

                    [self.view setNeedsDisplay];
                    return;
          }
          if (programMode == PAUSE) {
                    return;
          }

} // end gameLoop
```

The logic in *gameLoop* is starkly simple. The code calls *doPhysics* to update the physics model. *doPhysics* calculates the spaceship's latest position, velocity, and heading, based on the changes in the variables for rotation and thrust received from the UI. Next, *gameLoop* has the control pad and info button views redraw themselves. This is done so that the controls can appear with the proper translucent value, depending upon whether the user is touching them or not. Updating the spaceship view is slightly more complicated, but not much. The *ShipView*'s transform property is updated with the ship's latest heading, and then the transformation matrix in this property is applied to the spaceship view, which rotates it. *ShipView*'s *upDateShip* method is called to load the corresponding image into the view, which depends upon whether the thrust variable is set. Finally, the code has *ShipView* redraw itself. The code of doPhysics I dropped in straight from the Android program. I modified its API calls that obtain the absolute time and trigonometric values from Android's to those for Coca Touch (Listing 6). The one hiccup I had was that I couldn't just jam the values for the ship's *x* and *y* coordinates into *ShipView*'s center property. This property is a structure of type point and Objective-C syntax doesn't allow you to modify individual elements in a structure if it is an *l*-value. The easiest solution was to place the coordinates into a point with the *CGPointMake()* function and write this point into the center property.

```
LISTING SIX
-(void) updatePhysics {
   CFTimeInterval now = CFAbsoluteTimeGetCurrent();

// Calculate and respond to situation where ship tries
// to go off screen.
   CGRect rect = [[UIScreen mainScreen] applicationFrame];

   if (shipX <= kEdgeCondition){
      shipX = kEdgeCondition;
      mDX = -mDX;
   } else if (shipX>>= (rect.size.width - kEdgeCondition)){
      shipX = (rect.size.width - kEdgeCondition);
      mDX = -mDX;
   } // end else
   // Handle top and bottom
   if (shipY <= 10){
      shipY = 10;
      mDY = -mDY;
   } else if (shipY >= (rect.size.height - kEdgeCondition)){
      shipY = (rect.size.height - kEdgeCondition);
      mDY = -mDY;
   } // end else

   double elapsed = (now - lastTime) * 10.0;

   // Ship is rotating — update its heading
   if (shipTurning != 0.0) {
      shipHeading += shipTurning * (SLEW_SEC * elapsed);

      // Bring value back into the range 0..360
      if (shipHeading < 0.0) shipHeading += 360.0;
      else if (shipHeading >= 360.0) shipHeading -= 360.0;
```

IB Tricks and Traps

Interface Builder (IB) is valuable for visually constructing an iPhone's UI and choosing the messages that it generates. However, like any software tool, mistakes in construction can result in the app crashing. The UI objects, their attributes, and message assignments are buried in either in .xib file (a text file consisting of dense XML that's generated for development purposes) or binary format (a .nib file that's generated for the release version of the app). These objects are loaded where you can't study or trace them, which makes trouble-shooting difficult. Generally, if the debugger informs you that it is terminating the application due to an uncaught exception, the problem is rooted in a view not loading properly, or due to a broken link in the nib file.

However, there are a number of ways you can safeguard your efforts, which I have fashioned into rules. These rules have been derived from my own experience.

Rule 1: When you make a view in IB, the first thing to do before all else is configure the File's Owner object. That is, you specify the object that is the recipient of the view's messages. Click on the File's Owner icon in IB's document window, and choose the Identity Inspector. In the Class field, enter the class's name (typically a view controller). Type-ahead in this field is supported, but be careful: if you have a lot of classes with similar names, a wrong keystroke could select the wrong one. Save the nib file with an appropriate name.

Rule 2: For view objects, be sure to assign an outlet. This is the inverse of the purpose described rule 1: you're assigning a variable in the File's Owner that points to the view. Control-click on the File's Owner icon in IB's document window, then drag a link to the view icon in this window. You should get a pop-up menu specifying a view instance. Choose it, and save the file.

Rule 3: Build your nib files from scratch. While you can learn valuable interface techniques by studying the nib files of example programs, reusing them is not a good idea. The reason is that as you change things to suit your app, you break the pre-existing links in the example program's nib. Unless you are very thorough, finding all of the broken links is difficult and tedious. You're better off spending the effort making your own nib files from the ground up. You'll learn more in the process as well.

Rule 4: If you change the name of a class in your source code, be sure to update the changes in IB as well as any source files. In particular, update the File's Owner object. See rule 1.

Rule 5: When loading a nib file in code, be sure that the nib file name you present the initWithNibName method matches the actual file name, or is the correct one. Where this burns you is when you copy and paste some code, and forget to modify the nib file name. While I find it useful to have the nib file describe its class (such as SpaceView for a view object), there's no requirement that the nib file name has to match anything.

Rule 6: Have XCode, along with your project, up and running as you build the nib file, as suggested by Apple. These two programs intercommunicate changes, which is handy as you modify or add methods that will be targets of messages sent by the UI. Sometimes one of these programs can miss a beat and the nib file falls out of sync with the app code. This can be avoided by performing a clean build once and a while.

Rule 7: When IB gives you a hint, take it. Occasionally as you re-arrange the UI's elements, you might notice a warning icon at the bottom of IB's document window. IB's telling you something isn't right. Click on the icon to get a nearly plain-English explanation of the warning, which should help you correct the problem.

Apple has an excellent tutorial on building a nib file from scratch (see "Your First iPhone Application") that covers Rules 1 and 2. However, it doesn't stress sufficiently that failing to observe these procedures is going to cause you grief.

```
    } // end if mRotating != 0

    // Base accelerations
    double ddx = 0.0;
    double ddy = 0.0;

    if (thrust) {
        // Taking 0 as up, 90 as to the right
        // cos(deg) is ddy component, sin(deg) is ddx component
        double elapsedFiring = elapsed;

        // Have this much acceleration from the engine
        double accel = (FIRE_ACCEL_SEC * elapsedFiring) / 10.0;
        double radians = shipHeading / 180.0 * M_PI;
        ddx = sinf(radians) * accel;
        ddy = -(cosf(radians) * accel);
    } // end if

    double dxOld = mDX;
    double dyOld = mDY;

    // figure speeds for the end of the period
    mDX += ddx;
    mDY += ddy;

    // figure position based on average speed during the period
    shipX += elapsed * (mDX + dxOld)/2;
    shipY += elapsed * (mDY + dyOld)/2;

    CGPoint shipLocation = CGPointMake(shipX, shipY);
    shipView.center = shipLocation;

    lastTime = now;
}
```

Finally, because of the absolute time values returned by the iPhone API were different from those returned by Android, I had the tinker with some constants to get the spaceship's motions working smoothly. Overall, the changes to the physics code were minor and it worked almost "as is" in the app, due to the fact that Java's arithmetic operations are nearly identical to C's. In fact, porting the physics algorithm was probably the easiest part of the project, because I was working with proven code. Porting more sophisticated code from Android or other mobile platforms would require extra work, depending upon how much the algorithms rely on the other platform's APIs.

Touching the information button loads the *InfoViewController* class, which is a controller view that manages a view displaying help information. There's some nifty code that performs an animation that rotates the help screen over *SpaceView* as it is added as a subview. I had originally considered making the help screen just another view, but I added a view controller, *InfoViewController*, since I might add other interactive elements to it in the future. Such elements might be the ability to let you choose where the control pad appears.

Once I got the basic help screen working, I realized I had a problem: the spaceship would keep drifting merrily about while the help screen concealed it. I therefore made a *setmode* method that modifies a variable, *programMode*. If *programMode* was set to the constant value PAUSED, then the *gameLoop* immediately exits, rather the calling *doPhysics* and updating the views. With this change, when you switch to the help screen, *SpaceView*'s actions stop until you dismiss it. The little bit of code present in *InfoViewController* performs the rotating animation back the *SpaceView* screen, and has the app resume its execution.

Saving State

Being able to halt the app's physics calculations also served another valuable purpose. Suppose a phone call interrupts the app? In this case, the calculations and updates must be stopped while the user decides whether to take the call. If she answers the call, the iPhone OS warns the app that it is going to be shut down. The app receives an *applicationWillTerminate* message, and in response it must preserve its critical state variables. If this is done properly, when the app is relaunched it restores its state and can resume where it left off.

There are a number of ways to store an app's state. Cocoa Touch offers archiving features that allow you to save data and objects. The most comprehensive archiving mechanism is *NSCoding*, which lets you save the complete object graph of a group of objects. *NSCoding* preserves objects into nib files, and when they are later reloaded, the complete state of the archived objects is restored. You got a glimpse of this capability when working with views in IB. The UIKit class documents specify upfront whether each class implements *NSCoding*. The problem is that you must write *NSCoding* methods to preserve and restore your custom-made class. That seemed a lot of work, particularly when my app can be easily reconstituted by preserving just a few variables.

Cocoa Touch also offers a serialization mechanism that lets you store certain data objects. The serialized data is stored as a property list, which could be part of the application's property list (the Info.plist file) or stored as user preferences, using *NSUserDefaults*. I chose *NSUserDefaults* because it was simple to set up and met my needs.

With *NSUserDefaults*, you stash the data into a shared data object, using a key (a string) to uniquely identify it. You also use key names to both identify and access the variables you store within the object. The methods I wrote to store and retrieve the variables are shown in Listing 7.

```
LISTING SEVEN
- (void)saveState {
   NSUserDefaults *savedData = [NSUserDefaults
standardUserDefaults];
   [savedData setDouble:shipHeading forKey:@"shipHeading"];
   [savedData setDouble:shipX forKey:@"shipX"];
   [savedData setDouble:shipY forKey:@"shipY"];
   [savedData setDouble:mDX forKey:@"mDX"];
   [savedData setDouble:mDY forKey:@"mDY"];
   [savedData synchronize];
}

- (void)restoreState {
            NSUserDefaults *savedData = [NSUserDefaults
standardUserDefaults];
            int programStateSaved = [savedData
integerForKey:@"programStateSaved"]; // Get saved state flag

   if (programStateSaved) {
      shipHeading = [savedData doubleForKey:@"shipHeading"]; //
Get heading
      shipX = [savedData doubleForKey:@"shipX"]; // Get X coord
      shipY = [savedData doubleForKey:@"shipY"]; // Get Y coord
      mDX = [savedData doubleForKey:@"mDX"]; // Get X velocity
      mDY = [savedData doubleForKey:@"mDY"]; // Get Y velocity
   } else {
                  // First run
      [savedData setInteger:1 forKey:@"programStateSaved"];//
Create save flag
      shipX = self.view.center.x;
      shipY = self.view.center.y;
      mDX = 0.0;
      mDY = 0.0;
      shipHeading = 0.0;
   }
}
```

Saving data in the user preferences object is straightforward. You retrieve a pointer to access the object, and save the critical velocity, position, and bearing information tagged with descriptive key strings. This code is invoked when the *UIApplicationDelegate* object receives the *applicationWillTerminate* message.

Retrieving the stored data is easy, and its method is called from the delegate's *applicationDidFinishLaunching* method. The code first checks for a flag, *programStateSaved*, to see if the app's stored data exists. If not, it makes such a flag and stores some default values into the object. If the flag exists, then the code uses the same keys used to store the data to retrieve it. The code places these values into the appropriate variables. Now all that's left to do is have *gameLoop* resume execution, and the physics engine picks up where it left off.

Finally, because of the iPhone's OS X underpinnings, you have a third option in which to save the app's context. You can use UNIX BSD-style file system calls to store and read the app's state, rather than Cocoa Touch. A good example of this technique appears in Wolfenstein 3D, where its *SaveTheGame* method uses a passel of *fwrite()* calls to save the game's critical variables, and its LoadTheGame method uses *fread()* calls to restore the game's state.

Final Polish

Getting a working app isn't enough, however. There are other design details that need to be dealt with. The first was checking the usability of the interface that I'd cobbled together. For that, I sought the feedback of a beta-tester, which was my son, John. While waiting in line for Avatar, I handed him the iPhone and had him fire up the app and fool with it. His feedback was swift and sobering: The control buttons were too small, and they needed to better indicate their function. I had drawn some crude arrows for the control pad and obviously they were lacking.

While I have some graphic arts skills, the app's UI would fare far better if I found some professionally designed arrow images on the Web. Ideally, they should be in PNG format (the iPhone's preferred image format, although it can handle JPEG, TIFF, GIF, Windows BMP, and XWindows bitmap files), and with a transparent background. I wasn't asking for much, was I?

After a little scouting, I found the Icons web page, which is part of the MySiteMyWay site (http://icons.mysitemyway.com/). It

Figure 4

provides a vast array of attractive icons in different colors and designs. There are icons for all sorts of images and concepts, and they are 512 x 512 pixels in size with transparent backgrounds. You can use any pixel-editing program to scale them down for the iPhone, although the original size might be handy for iPad app interfaces. Best of all, they're free!

I downloaded the glossy silver icons archive, and used Lemke Software's Graphic Converter X to edit the images. I selected several arrow images to represent the various control pad actions, scaled them down, and then merged them together into one image. This became the control pad displayed on the screen (Figure 4).

To complete the app, two other images had to be made. The first was a 320- by 480-pixel PNG image that's stored in a file named Default.png. When the iPhone app launches, this image is loaded automatically as a background scene until your app initializes and adds its views to the window. Otherwise, the user becomes alarmed when they stare at a blank screen for more than several seconds. A properly designed default image shows the user that the application is starting up and doing things. You can see this behavior in Apple's weather app, in that it first displays the blank days of the week graphic while it fetches weather information. For simple apps, this default image could be either a splash screen, or present a message that states the app is loading. More complex apps with longer load times should display a progress bar.

I had used Blender3D, an open source 3D modeling program, to generate the spacecraft image for Android. I had the original model file handy, so I used Blender3D render a larger image for use as the splash screen. Next, I had Blender3D render the ship into 57 x 57 pixel image file named Icon.png. This image serves as the application icon. I added these two files to the XCode Space project and after rebuilding the app, the icon and splash screen appeared with no further coding effort on my part. My tester, John, was much happier with the UI.

Lessons Learned

Looking back, there are a few things I would have done differently. For example, for my info button I use a graphic, but late in the design I learned that the there are two button types, *UIButtonTypeInfoLight* and *UIButtonTypeInfoDark*, that display the info icon for you. In addition, I'd group the initialization of the control pad and info button views into one method, and probably do the same for updating them. Contrary to how things are done on a desktop application, where one updates the UI as things happen, on a mobile platform it's better to group the graphics redraws together into bursts that allow the processor to fall idle and converse battery life. The reason for the separate code for these views is that my app grew incrementally as I learned things. Finally, the simple update code just hammers out redraw messages in response to touches, particularly for the ship's engine firing. It might be worth investigating if a little logic could implement smarter screen updates that occur only when something really changes.

Looking ahead, improvements that I'd like to make to Space are its ability to handle a change in screen orientation. Also, I'd like to make the screen controls configurable. I'm left-handed and put the control pad in the iPhone's left corner, but I realize my handedness is in the minority and other uses might want the control pad on the right. I'd also like to add guard logic so that the app could react if critical image files were missing, or if the saved variables were corrupt.

However, as such my iPhone app shows that I gotten off to a good start on my journey. I now understood Cocoa Touch's View classes sufficiently to draw to the screen, display controls, handle finger taps, and update the display in response to them. In addition, the app saves and restores its state when shut down, or when I take a call. The program accomplishes all of this using little code, which speaks well for the capabilities of Cocoa Touch and the iPhone mobile platform.

—— *Tom Thompson is the head of Proactive Support for embedded products at Freescale Semiconductor. He can be reached at tom_thompson@lycos.com. The views stated in this article are the author's, and don't necessarily represent Freescale's positions, strategies, or opinions.*

Return to Table of Contents

Contextual Applications and the Flash Platform

Creating mobile apps that look-and-feel the same as on the desktop and browser

By Ryan Stewart

In an increasingly fragmented world of devices and screens there has been a rise in the demand for "contextual applications" — applications that provide a similar and unique experience across all of the devices it's deployed on. It's not the same application running everywhere, but rather a similar application that makes full use of the features of individual devices. The user experience and overall look-and-feel are customized based on the constraints of those devices. A web version of the application might be optimized for the browser with short, quick access from anywhere. The mobile version of the application might be set up to use the GPS to provide local information based on where the user is.

With so many different screens and contexts it becomes a challenge to create a unified experience and manage deployment on those devices. With the Flash Platform, Adobe offers cross-platform runtimes for both browser content and deploying applications on desktops and devices. With the Open Screen Project (http://www.openscreenproject.org/) Adobe is working with content providers, handset manufacturers, carriers, and even silicon vendors to provide users a consistent experience across devices. The next generation of Flex (codenamed "Hero") will further improve developer productivity when creating contextual applications on the Flash Platform.

Originally, Flex was created as a way to create robust, complex applications inside the browser. With the release of Adobe AIR it evolved to support cross-platform desktop applications as well. With the rise of mobile applications Flex and Adobe AIR are taking that same message of cross-platform to small screens. The next generation of AIR and Flex "Hero" will let developers create mobile applications using the same tools and workflows they have been using to build Flex/Flash applications in the browser and on the desktop. Just like their desktop/browser counterparts, these applications will run on any device that supports AIR so that the developer doesn't need to write code for multiple platforms.

But with the momentum of mobile applications, there are new challenges for developers. The smaller screen real estate means that developers have to be more careful about managing navigation and content. Nearly all mobile devices are touch-screen enabled so controls have to support touch inputs. Mobile devices also have extra features like GPS support and accelerometers that users expect to be able to use in their mobile applications.

One of the benefits of Flex "Hero" is that existing Flex developers don't have to change a lot of the way they do things. The components are the same, the language is the same, and the same tools work to build mobile applications.

In fact, one of the core goals of Flex "Hero" is to provide a framework that can easily transition between screens and devices. In fact, a lot of the core components, Buttons, List, etc, are getting mobile support baked in. That means developers won't have to use a different kind of Button component for mobile devices. Instead, there is a mobile theme that provides certain optimizations for using mobile applications. But from the developer's point of view, a list component they create for the desktop application can be used in generally the same way on a mobile device. Flex "Hero" changes the look, feel, and behavior of the list component for whatever screen/theme the developer sets.

That doesn't mean that a desktop application will translate perfectly into a mobile application so that one application will run everywhere. But it will let developers reuse some code while still making sure the experience is customized for whichever device it's being deployed on.

This application will use the *MobileApplication* tag as the root. The *MobileApplication* tag allows use of the paradigms I'll talk about below. By default, it also uses the mobile theme.

```
<s:MobileApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
            firstView="views.DemoProjectHome"
           chromeColor="#007c3a"
splashScreenImage="@Embed('assets/olympics.jpg')">
```

One of the things in that application code is a property called *splashScreenImage*. Flex "Hero" provides developers with a way to show an initial image on startup. This helps the application feels more responsive and provides immediate feedback to the user when the application loads. To set a *splashScreenImage* simply use that property on the main application and the user will see that image when the application first starts up. In this case, the image is a slightly modified version of what users see when the application finishes loading.

Views

Because of the small screen size, most mobile applications are divided up into more pages or screens than desktop applications. It becomes critical to be able to manage and create these screens for each part of the application. "Hero" deals with this by making each view a separate file and giving developers a *View* component that can be extended or worked with as a base. A typical view might have a list of items and another view might be the detail page, which is the case in this application.

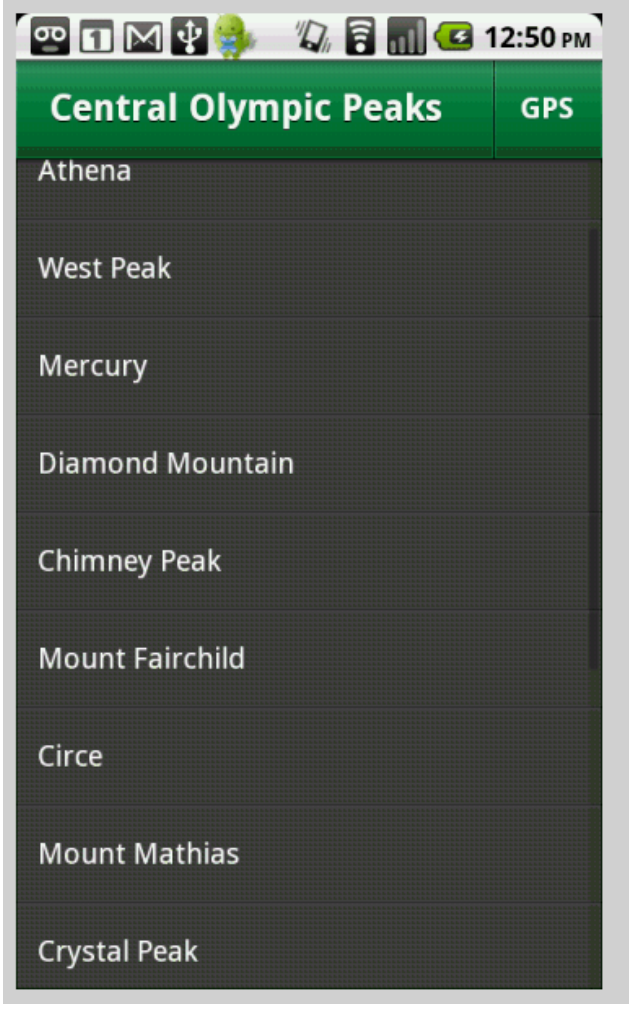


Figure 1: Main view with list component

The view component in "Hero" extends the basic Flex container component, a Group, so any Flex developer familiar with how containers work can add views. Here's a basic view, which includes a *list* component:

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
                  xmlns:s="library://ns.adobe.com/flex/spark"
                    title="Central Olympic Peaks">

        <!— This is just an array of all the peaks with data —>
        <fx:Script source="../PeakList.as" />

        <s:List id="list" dataProvider="{arrPeaks}" width="100%"
             height="100%"
                    labelField="name"
change="list_changeHandler(event)" />
</s:View>
```

Flex "Hero" mobile applications provide the *ViewNavigator* class to add and remove views from the application. The main application tag includes a *firstView* property that sets the default view. In this application the *firstView* is set to the view above. When an individual *list* item is selected the application will transition to a detail view using the *navigator* property, which is global to the entire application. The *navigator.pushView()* creates the view and then activates it. It also plays a default transition that is optimized for mobile devices so it will look smooth and provide a sense of depth to the application.

By delaying the creation until the view is needed, Flex optimizes the amount of memory used. When the view is no longer needed, the developer calls *navigator.popView()* and Flex automatically removes it and transitions back to the original view. But often after the view is destroyed, the user will want to go back to it. The view data architecture in Flex "Hero" takes this into account. Even when the view itself is destroyed the framework saves all of the user's data from that view. Therefore, if the application is closed or if the user goes back to the view, the data will still exist without the overhead of keeping the view rendered off the screen. See Figure 1.

```
<fx:Script>
      <![CDATA[
            import spark.components.ViewNavigator;
            import spark.events.IndexChangeEvent;

            protected function
list_changeHandler(event:IndexChangeEvent):void
            {
                  // When a list item is selected, the pushView
method moves to the detail
                  // view and also sends the data from the
selected item so it can be used
                  // in the detail
view.navigator.pushView(DemoProjetDetail,list.selectedItem);
            }
      ]]>
</fx:Script>
```

ActionBar

One of the mobile design paradigms that has become popular is the idea of an action bar. Because you can't easily fit a lot of items on

a page the way you would with a desktop application, using the action bar can be a helpful way to highlight the most important or often-used tasks. Flex "Hero" has an *ActionBar* component that can be global or customized based on the current view.

By default, the *ActionBar* will just show the title of the current view. By using the *<s:actionContent>* tags however, the global action bar can be modified to show whatever the developer wants. In this case the application is going to have a global button that goes to a view that shows current latitude, and longitude coordinates. That button will show up in every view of the application. The *actionBar* content looks like this:

```
<fx:Script>
      <![CDATA[
            import mx.events.FlexEvent;
            import spark.components.ViewNavigator;
            import views.DemoProjectGPS;
                  protected function
                      btnGPS_clickHandler(event:MouseEvent):void
                 {
                  // When this is selected move to the GPS view
                       navigator.pushView(DemoProjectGPS);
                  }
    ]]>
</fx:Script>
<s:actionContent>
      <s:Button id="btnGPS" right="5" label="GPS"
             click="btnGPS_clickHandler(event)"/>
</s:actionContent>
```

Each view can also have an *actionContent* tag that will overwrite the global *actionContent*. That makes it easy to add something like a back button in a detail view.

```
<s:actionContent>
       <s:Button id="btnBack" label="Back"
              click="btnBack_clickHandler(event)" />
             </s:actionContent>
```

Enabling Touch

One of the other major issues when dealing with mobile devices is the input. The irony of building mobile applications is that the screen size is smaller but the components have to be larger to fit the less accurate finger instead of a precise mouse cursor. By default, the skin in Flex "Hero" was made with this in mind. Buttons are bigger, text is bigger, and in general every mobile-optimized component will be bigger to support the navigation using fingers.

But another critical part of inputs on devices is gestures and touch-enabled components. The list is the perfect example. In a desktop application or in a browser application, a list usually has a scrollbar that is used for moving up and down. But with mobile applications, the default list scrolling gesture has become a swipe with the finger up and down.

Flex "Hero" tries to make the input modes as transparent for the users as possible. Every component now has an *interactionMode* property that can be either touch or mouse. By default, when using the mobile theme, the components will use the touch *interactionMode*. All of the other themes will use the mouse *interactionMode*.

The touch *interactionMode* can also be used on other devices or screens. Even if the developer is not using the mobile theme, the components can still be touch-enabled by detecting whether the device the application is running on supports gestures and then toggling the *interactionMode*.

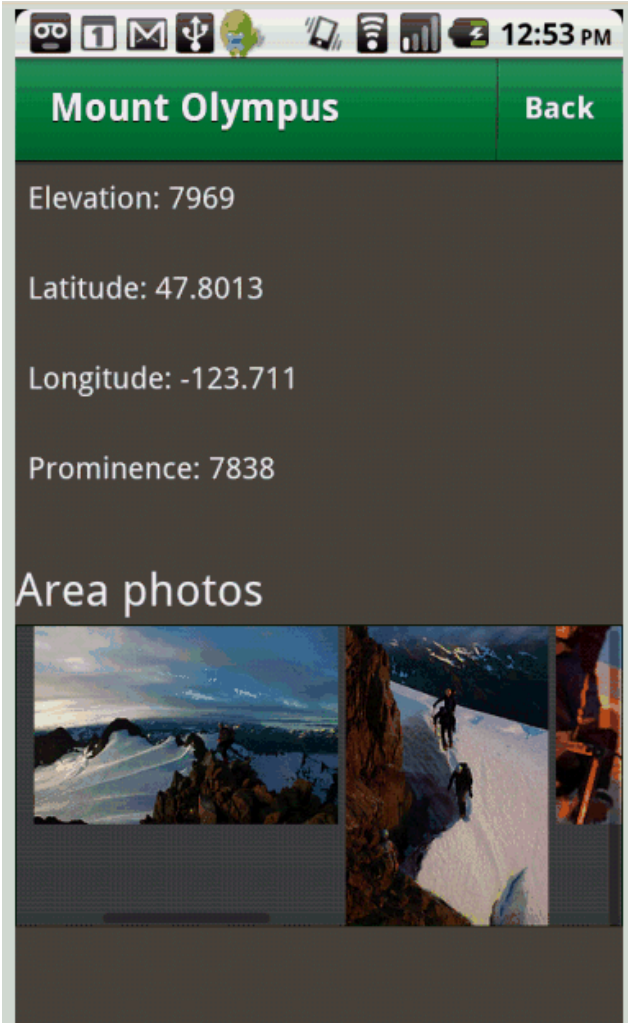


Figure 2: Detail view

For instance, in a desktop application that is using Flex "Hero," most of the interaction will be done with a mouse. But the developer can program logic that detects if the application is running on a touch screen monitor. If it is, the *interactionMode* of the application can be set to touch and the user can interact with the components using gestures. See Figure 2.

Device Integration

One of the most exciting things about mobile devices is that they include so many sensors. Most smartphones now come with GPS receivers, accelerometers, and cameras. Users expect applications to be able to take advantage of those sensors.

With Adobe AIR and Flex "Hero," those sensors can be used by developers in ways that will be completely cross platform. So instead of having to create code to talk to the GPS or accelerometer

of specific devices, AIR and Flex provide APIs that will work across screens and devices. In an increasingly fragmented world, that means a lot less extra development work and more time refining the user interface and design.

In this example, the application is going to use the GPS sensor to figure out the location of the user and then sort the list of mountains by distance. When the user selects the GPS view, the application will provide information about latitude/longitude and the accuracy of the GPS fix.

The GPS sensors in AIR are very easy to use. Once the developer checks to be sure that GPS is available, he or she can specify how often to request location updates, and then create event handlers for when those updates are received. The application will then sort the list accordingly.

```
<fx:Script>
    <![CDATA[
          import flash.sensors.Geolocation;
          import mx.events.FlexEvent;
          import spark.components.ViewNavigator;
          import views.DemoProjectGPS;

          public var gpsEvent:GeolocationEvent;
                  protected function
                      btnGPS_clickHandler(event:MouseEvent):void
          {
                 // When this is selected move to the GPS view
                      navigator.pushView(DemoProjectGPS);
          }
         protected function
             mobileapplication1_applicationCompleteHandler
               (event:FlexEvent):void
         {
  // Check to make sure that Geolocation is supported on the device
                  // and activate it
                if(Geolocation.isSupported)
                {
                       var geo:Geolocation = new Geolocation();
           // This checks for new GPS data every 10 seconds
                      geo.setRequestedUpdateInterval(10000);
  // Add an event listener for when the GPS updates its status

geo.addEventListener(GeolocationEvent.UPDATE,onUpdate);
              }
               }
               protected function
                 onUpdate(event:GeolocationEvent):void
              {
  // When we get an update from the GPS, set it to the gpsEvent
  // variable which will be accessed by the GPS view
                   gpsEvent = event;
               }
     ]]>
</fx:Script>
```

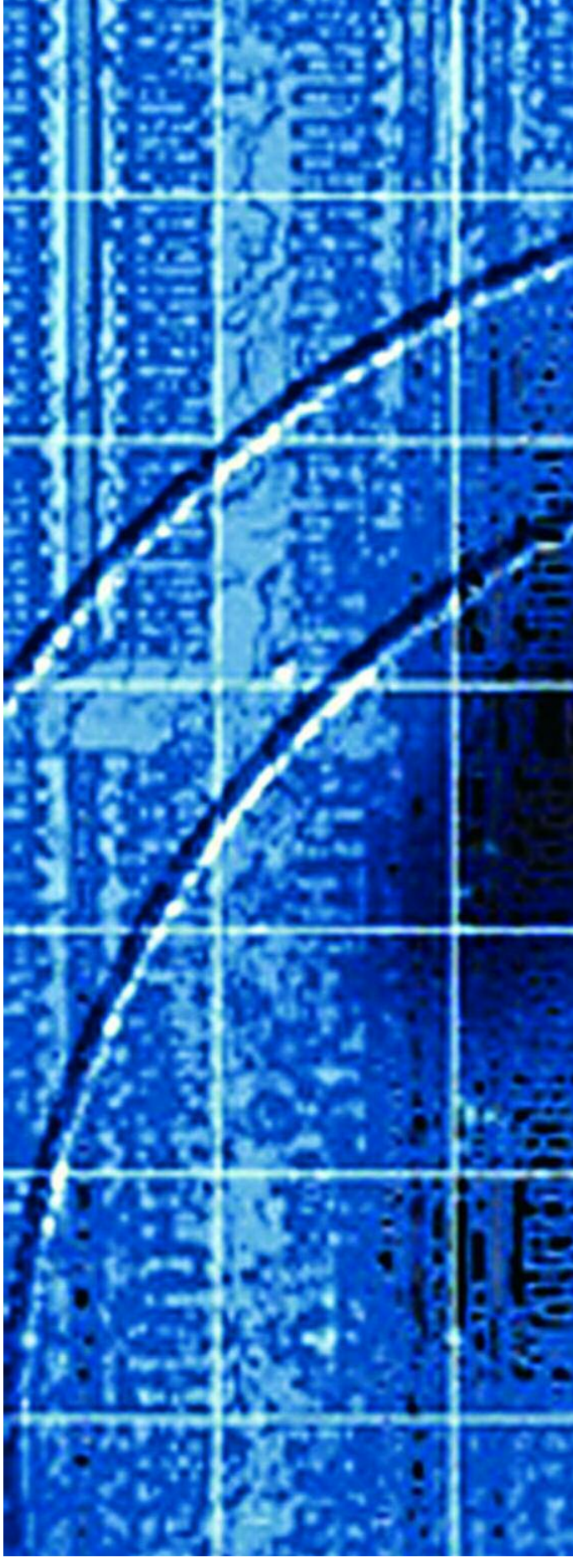
Conclusion

Flex "Hero" is designed from the ground up to let developers move easily between screens so that Flex applications can be deployed quickly for the desktop or for mobile devices using the same components and same workflow.

The applications created with Flex "Hero" are completely cross platform thanks to AIR for mobile devices which means developers can take advantage of the specific features of devices without needing to know the APIs for each individual mobile platform.

— Ryan is a platform evangelist at Adobe.

Return to Table of Contents



MeeGo Application Development

A platform designed for squeezing a positive user experience out of limited hardware

by Eric J. Bruno

The MeeGo platform is a Linux-based OS built especially for Intel Atom-based netbooks. Not only is MeeGo an excellent OS to use, it's a well-supported Linux kernel-based application platform that you can write to. Intel and Nokia, as well as the entire Linux community, provide plenty of tools to make your dream application a reality. In this article I examine the MeeGo architecture, platform, and SDK, to help you get started on your netbook application today.

MeeGo Architecture

MeeGo (http://meego.com/developers) is designed for the best user experience on limited hardware, and contains a Linux kernel, a custom desktop based on the X Window System (X11), a message bus for efficient inter-application communication, and all of the communication services, graphics APIs, and media services that you'd expect in a modern day OS (see Figure 1).

At the base of the OS is a hardware adaptation layer that hardware vendors provide to support MeeGo on top. This includes device drivers, configuration data, and any special software required for MeeGo to interface with the low-level hardware. Directly on top of this is the Linux kernel itself, which includes its own set of kernel drivers to work with the hardware components, as presented by the hardware adaptation layer. Since the adaptation software abstracts the specifics of the hardware, no changes are expected in the kernel from device to device.

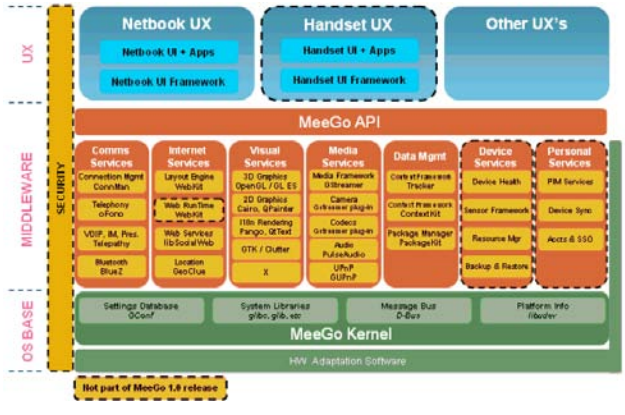


Figure 1: The MeeGo OS architecture (courtesy of MeeGo)

On top of the kernel, you'll find X11, the system bootloader, and the MeeGo Core, which consists of a set of services that applications can use:

- Communications Services: Bluetooth, telephony/VOIP, IP, and so on
- Internet Services: WebKit, GeoClue (location services), social networking web services, and so on.
- Visual Services: X11, 2D/3D graphics, OpenGL/ES, QPainter, QtText, GTK, and so on
- Media Services: GStreamer (animation and video), codecs, audio, and so on
- Data Management: package manager, and so on
- Device Services: sensor framework, resource manager, backup/restore
- Personal Services: PIM, device sync, and so on

A unified MeeGo API makes these services uniformly available for application development, which includes support for both the GTK and Qt UI frameworks. On top of it all are the individual user experience packages (such as the Netbook UX, Handset UX, Tablet UX, and so on).

The MeeGo API

The heart of the MeeGo API (http://meego.com/developers/meego-api) is Qt (pronounced "cute") version 4.7, which is a cross-platform development framework for GUI applications (also known as a "widget toolkit") originally developed and distributed by Trolltech. Nokia acquired Trolltech in 2008, mainly to gain control of Qt and integrate it into its own embedded products. Qt (http://apidocs.meego.com/qt4.7/) has also been used in some very popular software packages, such as VirtualBox, Opera, KDE, and even Google Earth.

The power of Qt is that it allows you to develop an application with one set of sourcecode that will run across platforms, including within browsers, across desktops, mobile devices, and even resource-constrained embedded systems seamlessly. Qt Creator is

the IDE (see Figure 2) that Nokia provides for application development and debugging, but other tools are often required as well. These include Qt Designer (see Figure 3), which aids in GUI layout and development; Qt Linguist, which helps you internationalize your applications for use in other regions of the world; Qt Assistant, which helps you write and present online documentation and help pages for your applications; and other tools such as Qt Virtual Frame Buffer, which helps you develop embedded Linux applications on your desktop using X11.

The latest version of Qt Creator allows you to use the other tools embedded within, but they can still be run standalone.

Also essential to developing MeeGo applications is the MeeGo Touch Framework, which provides everything you need to support gesture and touch-enabled devices. This API ensures that all MeeGo applications look, navigate, and generally behave the same way when it comes to touch interfaces. The Touch API builds on Qt, and exposes a higher level programming paradigm with a look-and-feel optimized for embedded devices. In particular, MeeGo Touch builds on top of the following Qt frameworks:

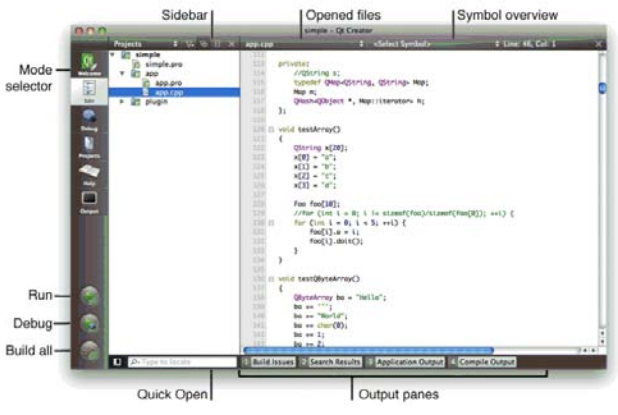


Figure 2: The Qt Creator application in use (courtesy of Nokia).

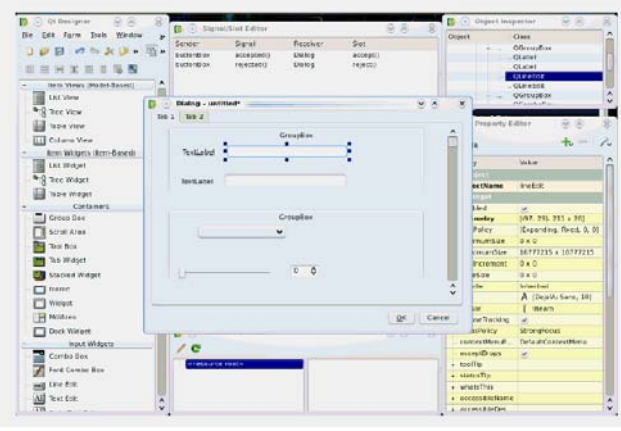


Figure 3: The Qt Designer application in use (courtesy of Wikipedia)

- Low-level 3D Framework, an OpenGL-based layer 3D graphical rendering.
- Graphics View Framework, an API to manage large numbers of 2D graphical items, with an event framework that tracks mouse and keyboard input and movement, based on the model-scene-view paradigm.
- Animation Framework, part of the Kinetic project, this framework allows you to create smooth, timeline-based, animated GUIs, with UI effects we've come to expect in embedded devices.
- Qt State Machine, this framework allows you to create and enact state graphs with statecharts. This allows you to model your UI's behavior to system and user events, and then program the reactions to those events. When the events occur (i.e. the user clicks a button) in the system hosting your application, the framework does the actual work to transition from one state to another according to your specifications.

Under development (but available in beta form) by Nokia, Intel, and partners is the MeeGo Web Runtime (WRT), which allows you to build web-based Ajax applications (HTML, CSS, and JavaScript) but still access the device's lower level features, and Qt Mobility. This API allows you to target embedded telecommunication devices running MeeGo, and access features such as contacts and the low-level communication system.

The Development Environment

In terms of a development environment, you have three choices when it comes to developing MeeGo applications (all of which are Linux only at the current time):

1. You can develop on MeeGo itself, installed on a machine with higher capabilities than the typical netbook or handset. This requires you to download the MeeGo image onto a workstation class desktop or laptop computer and run it as your base OS. This can be awkward to some developers, and the MeeGo UX is aimed more towards consumers than it is to power users or developers. However, it does make it more convenient to code, test, and debug, since you never need to leave the confines of your host MeeGo system.
2. For more flexibility, you can install the MeeGo SDK with the QEMU virtual machine onto your favorite Linux development desktop or laptop computer. Within it, you can run the MeeGo image (either the netbook or handset UX) within a window on your host system, where the graphics support is pass-through to your hardware for acceleration. You can even debug your application, running within the QEMU window, from Qt Creator running in your host environment (outside of QEMU, but all on the same hardware). You have the flexibility to run and test with either the x86 or ARM MeeGo images regardless of your host system architecture.
3. For the best performance, you can install the MeeGo SDK with Xephyr, which is supports a virtualized filesystem within which you unpack the MeeGo image.

This virtual MeeGo system supports everything the actual MeeGo filesystem does, including the installation of other packages and applications, which you can then run within a GUI window on your host system. With this setup, although you run the MeeGo UI and your applications within a window on your host system, you must launch Qt Creator within this environment to develop and debug with. The end result is excellent startup and execution performance, with a slight loss of flexibility.

When you're ready to release your application, you simply create a standard Linux RPM formatted package to distribute in of the available netbook application stores. One strong candidate is the Intel AppUp Center and Atom Developer Program (http://appdeveloper.intel.com/en-us/). Here, you'll find tools and resources to improve you applications' marketability, reach consumers interested in purchasing applications for Atom-based netbooks, and generally increase the market for your applications.

The Future

MeeGo 1.0 was released in May of 2010 with APIs and SDKs that support only Linux host environments, and the MeeGo Netbook UX only (http://meego.com/developers/meego-ux-design-principles). Since then, there have been weekly development builds that have improved on the 1.0 release with previews of upcoming features such as the Touch Framework and the Handset UX, but the more significant MeeGo 1.1 release is due in October of 2010. This release will officially support the Netbook and Handset UX (and possibly even an early-access version of the Tablet UX), along with the MeeGo WRT, and support for more host development environments such as Windows and Mac OS X.

— Eric J. Bruno is a contributing editor at Dr. Dobb's Journal. *He can be contacted at www.ericbruno.com.*

Return to Table of Contents