Newton® Technology

JOURNAL

Volume II, Number 1 *February 1996*

Inside This Issue

Newton

Extra Extra: Extras Drawer Features in Newton 2.0

by Maurice Sharp, Apple Computer, Inc.

The Extras Drawer in Newton 2.0 has undergone significant changes. Some, like the ability to scroll, are user level changes. Others changes are for the developer. This article talks about the Extras Drawer API's that did not make it into the beta release of the Newton Programmer's Guide.

Filing Your Package

The Extras Drawer now has folders. Users can file icons in either built-in folders or in ones that they create. As a developer, there are some occasions when it makes sense to file a package in a particular folder. A good example is a help book which naturally belongs in the Help folder.

The Extras Drawer code associates a `labels` slot with each application icon. If the icon is unfiled, the `labels` slot is nil or not present. To file the icon you need to set the `labels` slot to the correct folder symbol.

This can be accomplished in two ways. Once the package is installed, you can use `SetExtrasInfo`. For more information on that way see `SetExtrasInfo` below.

The other way is to associate a `labels` slot with your package at compile time. Note that the Extras Drawer displays icons for parts and a package can contain multiple parts. We will explore this a little later.

For now, assume you have a single-part

continued on page 17

Gotcha! Common Issues Converting From 1.x to 2.0

by Maurice Sharp, Apple Computer, Inc.

Now that you have seen all the cool new features of 2.0, you must be anxious to update your 1.x applications. But beware, in that first rush of excitement you may try to convert too much too soon and end up in a debugging nightmare from which even the new debug tools can not extract you.

This article covers some of the common gotchas that you will encounter when converting from 1.x to 2.0. If you follow the suggestions you will save some time and aggravation. If not, read on.

First Steps

Before you even consider starting, you should read the 2.0 documentation from cover to cover. As you are reading, make notes about what parts of 2.0 will give you the biggest wins for your application. You can also make notes about those functions you wish you had in 1.x but did not.

Create a list of the 2.0 features you want to add in your application. Now order that list from most to least important. Generally you can tell which new features give you the largest gains in terms of performance or interface. Then start converting one area at a time. The worst thing you can do is try to change all parts at once. Code an area, debug it, test it, and when it is finished, move on to the next area.

continued on page 22

Newton® Technology JOURNAL

Volume II, Number 1 *February 1996*

Published by Apple Computer, Inc.

Lee DePalma Dorsey • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

Gabriel Acosta-Lopez • *Coordinating Editor, DTS and Training Content*

Philip Ivanier • *Manager, Newton Developer Relations*

Technical Peer Review Board
J. Christopher Bell, Bob Ebert, Jim Schram, Maurice Sharp, Bruce Thompson

Contributors
Todd Courtois, Lee Dorsey, Jennifer Dunvan, Rick Giles, David Glickman, Scott Gruby, Jim O'Grady, Maurice Sharp, Don Vollum

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

John Kawakami • *Editorial Assistant*

Judith Chaplin • *Art Director*

© 1996 Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014, 408-996-1010. All rights reserved.

Apple, the Apple logo, APDA, AppleDesign, AppleLink, AppleShare, Apple SuperDrive, AppleTalk, HyperCard, LaserWriter, Light Bulb Logo, Mac, MacApp, Macintosh, Macintosh Quadra, MPW, Newton, Newton Toolkit, NewtonScript, Performa, QuickTime, StyleWriter and WorldScript are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. AOCE, AppleScript, AppleSearch, ColorSync, develop, eWorld, Finder, OpenDoc, Power Macintosh, QuickDraw, SNA•ps, StarCore, and Sound Manager are trademarks, and ACOT is a service mark of Apple Computer, Inc. Motorola and Marco are registered trademarks of Motorola, Inc. NuBus is a trademark of Texas Instruments. PowerPC is a trademark of International Business Machines Corporation, used under license therefrom. Windows is a trademark of Microsoft Corporation and SoftWindows is a trademark used under license by Insignia from Microsoft Corporation. UNIX is a registered trademark of UNIX System Laboratories, Inc. CompuServe, Pocket Quicken by Intuit, CIS Retriever by BlackLabs, PowerForms by Sestra, Inc., ACT! by Symantec, Berlitz, and all other trademarks are the property of their respective owners.

Mention of products in this publication is for informational purposes only and constitutes neither an endorsement nor a recommendation. All product specifications and descriptions were supplied by the respective vendor or supplier. Apple assumes no responsibility with regard to the selection, performance, or use of the products listed in this publication. All understandings, agreements, or warranties take place directly between the vendors and prospective users. Limitation of liability: Apple makes no warranties with respect to the contents of products listed in this publication or of the completeness or accuracy of this publication. Apple specifically disclaims all warranties, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Editor's Note

Letter From the Editor

by Lee DePalma Dorsey, Apple Computer, Inc.

Newton 2.0 Hits the Road Running — Development Support and Tools Keep Pace!

Newton 2.0 made it's first public debut during the second week of November at Comdex in sunny Las Vegas. And a sunny week it was for everyone involved in the roll out of the Newton 2.0 operating system and all of the Newton platform enhancements. Not only was Newton 2.0 well received by customers eager to get their first peek at the next generation Newton platform, but it was also very well received by members of the press. The rollout was about a lot more than just the new version of the Newton operating system, however. It was about a comprehensive approach to the Newton platform as a whole, covering many issues critical to you, the solutions providers.

The excitement and enthusiasm around Newton 2.0 is a welcome change for members of Apple's Newton Systems team. We've seen favorable reviews in MacWeek, Computer Retail Week, InfoWorld, ComputerWorld, and other leading industry publications. Customers have been clamoring to upgrade their MessagePad 120's, and to buy new units with Newton 2.0 to replace MessagePad 110s and 100s. Developers have been flooding the Newton Systems Group with requests for information on the newest version of the OS and have waxed poetic about all kinds of new solutions to enhance the platform's capabilities. Developers with compatible solutions have received unprecedented interest. And two new licensees (Harris-Dracon Division and Schlumberger) have announced the adoption of the platform for their hardware devices. Perhaps the most notable acknowledgment of the team's work and improvements to Newton was winning Byte Magazine's "Best of COMDEX" award in the new OS category, in the very first week of its public life. Clearly, Newton 2.0 is off to a great start.

Getting a good start in life is important and often critical to a product's success. We think we've gotten that start and hit the road running. Newton 2.0 meets the market on Apple's MessagePad 120, and licensees expect to ship product based on 2.0 in 1996. However, Newton 2.0 is just the beginning of the platform roll out story and it's introduction is certainly just the beginning of the rest of the platform's life. Equally important as the new OS, is the developer story: the desktop integration tools, the development tools, the new platform philosophy, and an integrated marketing approach to the platform and its solutions.

We've talked about the new platform philosophy, the integration tools and the development tools in previous issues of the Newton Technology Journal, and I cannot emphasize enough how critical a role each of these pieces plays in Newton's long-term success. Apple recognizes that there are several keys to succeeding in the mobile business professional space as well as the vertical space. First is the delivery of an open platform with robust development tools. We think Newton

continued on page 5

Introduction to Newton Programming

by Jennifer Dunvan, Apple Computer, Inc.

If you're considering taking the plunge to learn to write Newton applications, now is the time. If you have an object-oriented programming background, you've got a head start, but even if you've never programmed before, the Newton is a great place to begin. One of the best features of programming for the Newton 2.0 platform is the simplicity and power of the language, NewtonScript, combined with the fast and easy-to-use development environment, Newton Toolkit (NTK). This article will explain some basic concepts of Newton programming.

Before you get started with NewtonScript and NTK, you should understand some fundamentals about the architecture of the operating system. Given the inherent memory limitations of running on a small device, you may find the Newton memory model, view system, and inheritance model to be unique.

Memory

Before you begin writing Newton applications, you must understand how memory is arranged and used in the operating system. Imagine that Newton memory is divided into three parts: ROM, "protected" RAM, and RAM.

ROM is where operating system software components, or the built-in libraries, are stored. These include all of the pre-built buttons, boxes, menus, and other user-interface elements. Also included here are NewtApp objects and other pre-programmed "protos" for your use.

Protected RAM, or user storage, is where *soups* (also referred to as persistent data) and third-party applications, or *packages*, are stored. A PCMCIA card, or flash RAM, extends memory of this type, in addition to an already existing internal store. This area is deemed "protected" for two reasons: (1) packages are considered read-only, and (2) all data stored in this area will survive when you turn the Newton off or when you reset your device. (Note that this does not include what is called a "hard" reset, which occurs when you push the reset button while holding down the on-off button. Only data stored in ROM and system updates will survive a hard reset.)

The last portion is RAM, which is divided into two groups of memory, the NewtonScript heap and the System heap. The NS heap is what your application will use when it is opened by the user. You might expect your package to be loaded into RAM while it's running (just like apps your desktop machine), however, only a very small part of a Newton application uses RAM. Rather than making copies of objects and loading them at runtime, the Newton ingeniously makes use of objects already stored in ROM and protected RAM. This allows fairly complex applications to run in a relatively small space. This also has a profound effect on the way you go about programming for the Newton.

The System heap portion of RAM is used by the operating system for construction of objects and low-level communications purposes.

NewtonScript

NewtonScript is an object-oriented dynamic language. The fundamental data structures, or objects, of NewtonScript are called *frames*. You'll hear the terms *view*, *template*, and *proto*, which are typically all variations of frames. Their differences will be discussed later.

A Newton frame is reminiscent of a struct in C or a record in Pascal. Frames have fields or *slots* that contain data, functions, and/or more frames. The syntax for creating a frame is as follows:

```
myFrame := {
  age : 29,       //a slot in the frame
  name: "Jen",    //another slot...
  Weekend: nil,
  pet: func()     //here's a function definition
  begin
    if Weekend then return "kitten"
    else return "no pet";
  end,
}
```

You may notice several things from the above frame definition: a colon is used to initialize variables and functions in frame slots; you don't need to declare data types or allocate memory for frames or arrays (this is done for you dynamically); slot definitions are separated by commas; and you use the keyword `func` when you define functions. Not having to initialize pointers may seem too good to be true.

To access a slot in a frame, use dot syntax. For example,

```
myFrame.name;      //evaluates to "Jen"
```

To call a function within a frame, use colon syntax. For example,

```
myFrame.Weekend:=true;     //use ":=" to assign
myFrame:pet();     //evaluates to "kitten"
```

Let's try dynamic allocation and illustrate runtime typing:

```
myFrame.coolslot := "Wow, I got added on the fly!";
myFrame.age:="twenty-nine"; //change from int to string
```

Here's another nifty trick:

```
myFrame.myArray:=[1,"two",[3,4]]; //mixed data types
```

Fun, eh? And you probably guessed correctly that I didn't have to explicitly allocate any memory for that array.

If we look at the slots of `myFrame`, we now see

```
  age -> "twenty-nine"
  name -> "Jen"
  Weekend -> true,
  pet() -> function
  coolslot ->"Wow, I got added on the fly!"
  myArray ->[1,"two",[3,4]]
```

Views vs. Templates vs. Protos

As mentioned earlier, all things can reside in frames. There are some tricky concepts associated with this idea, and I'll do my best to explain them.

The basic objects in the Newton OS, the prototypes for all objects you will use and derive from, are called *system protos*. These protos reside in ROM and are built into the OS. They consist of the pre-programmed gadgets you use in your application, plus the basic framework your application will rest on. You can use system protos directly, or you can derive your own *user protos* and override functionality to your heart's content (be creative, but respect the UI guidelines). Keep in mind that system protos reside in ROM. Any protos that you create will usually reside in protected RAM.

In order to provide a user interface for your application, you'll need to create the prototype using NTK. This is where the visual aspect of your application comes to life. The frames you create in NTK are called *templates*. Newton uses the information contained in templates to construct the visual frames that the user sees and interacts with on the screen. These visual frames are called *views*, which are created in RAM by the Newton view system. Every item the user sees on the screen is a view, including buttons, icons, boxes, inputlines, and so on. Each of these views is instantiated at run time (and therefore resides in the NS heap) by the Newton view system using templates you create in NTK. Every view is defined by a template. And templates are typically defined by protos (system and/or user). For an excellent explanation of the Newton view system, read the article entitled "Tales From the View System" by Michael S. Engber, located on the latest Newton Developer CD.

Application Structure

A Newton application can be considered to be a tree of templates. Think of the topmost template as the fundamental or base template. When an application (or "package") is installed onto the Newton, a base view associated with this template is created and resides in RAM at all times, ready and waiting to receive an "open" tap from the user. When the application is opened, subviews for the application are created (these subviews are what the user sees and interacts with). When the application is closed, the subviews are destroyed, but the base view remains, ready to receive an "open" message again.

Inheritance

Newton inheritance differs from inheritance in the traditional programming sense in some fundamental ways. In the traditional idea of multiple inheritance, an object inherits functionality from two or more parent objects. In Newton programming, there are two flavors of inheritance: proto and parent inheritance.

Every view has two default slots. One is named `_proto` and contains a reference to the template it is based on, and the other is named `_parent` and contains a reference to its parent view. A typical template has one default `_proto` slot that references the system or user proto it is derived from.

Proto inheritance is pretty straightforward. For example, if I wanted to derive a new frame from `myFrame`, I'd do it like this:

```
newProtoFrame :={
  _proto: myFrame,      //proto inheritance syntax
  pet: func()           //override myFrame:pet()
  begin
    if Weekend
    then coolslot:="Yay!";      //creates and assigns locally
    return inherited:pet();   //calls myFrame:pet()
  end,
}
```

Currently, `newProtoFrame` contains two slots, a `_proto` slot referencing `myFrame` and a function slot containing `pet()`. You may find it interesting that calling `newFrame:pet()` creates a third slot called `coolslot` within `newProtoFrame`. Notice also the use of `inherited:pet()`, which automatically calls the proto frame's `pet()` function. The `inherited` keyword is reserved for proto inheritance only.

Parent inheritance works a bit differently. Suppose, for example, I want to use parent inheritance this time:

```
newParentFrame :={
  _parent: myFrame,      //parent inheritance syntax
  pet: func()            //override myFrame:pet()
  begin
    if Weekend
    then coolslot:="Yay!";   // re-assigns in parent frame!!!
    return myFrame:pet();   //calls myFrame:pet()
  end,
}
```

Calling `newParentFrame:pet()` actually changes the `coolslot` value in the parent frame (which in this case is `myFrame`) rather than creating a new local slot. This is an important difference between proto and parent inheritance. Normally, views use parent inheritance to communicate with each other and share data. Because they are located in RAM, it makes sense that a parent's child can alter the parent's data.

Proto inheritance, on the other hand, is typically used to inherit functionality from system or user protos, which are usually located in ROM or protected RAM. You can see that a frame would be unable to alter the data of the frame it `proto`'d from, even if NewtonScript allowed it.

Although a child frame has access to its parent's functionality, you normally would not override functionality via parent inheritance – although you may want to share data. Overriding functionality should typically be reserved for proto inheritance.

As you may have already gathered, another important aspect of Newton inheritance is that *an object stores only the differences from the object it inherits from.* This includes overridden data values and methods, as well as new data slots and method slots. The purpose of this design is to limit the amount of memory required at runtime, and to take advantage of the more abundant ROM. A typical view frame starts out small in RAM with only three slots: `_parent`, `_proto`, and a pointer to its `viewClass` (which you shouldn't worry about at this stage). Other slots are inherited or created dynamically as they get modified.

Finally, Newton inheritance provides no "security" in the C++ sense. All children derived from the the same parent can access their parent's and each other's slots, although you should remember that your templates and protos are usually safe because they reside in protected memory. In other words, you cannot usually alter data or functionality contained in a template or a proto. The only place data or functionality can be changed dynamically is in a view (because it resides in writable memory). This fact is probably the most challenging aspect of Newton programming, and it often takes some time to fully appreciate its implications.

Message-based Programming

If you have ever programmed for the Macintosh or Windows environment, you're familiar with event-driven programming. In this environment, when the user clicks or moves the mouse, an event-handling function in your application receives a message from the operating system, which it parses to determine what part of the screen got activated and which window to alert. While the Newton environment may seem similar to a desktop windowing environment in the sense that it receives user taps and gestures, it behaves differently in the way messages are handled.

Whenever you tap, write, or make a gesture on the surface of the screen, the Newton operating system sends a message directly to the *view that received the input* (rather than to a general message-handling function). For example, let's say the user taps on a button you created (and let's say you named it "myButton"). Upon receiving the tap, the operating system will send a `buttonClickScript()` message to the `myButton` view. It is in this function, the one that gets sent to your view by the OS, where you typically will want to add your code (like `self:SysBeep()` to make your button beep when it's tapped on, for example).

NTK

You'll be writing your application using Newton Toolkit (NTK), Newton's visual development environment that runs on a Macintosh (be on the lookout for WinNTK, the Windows version). You have to use NTK to write your application; there is no separate editor, compiler, or linker as you might expect with C/C++ or Pascal. Using your mouse in NTK, you visually design exactly what you want your Newton application to look like by adding and creating buttons, boxes, icons, and other gadgets. The best part about designing in NTK is that you get immediate satisfaction by seeing what your app will look like on the Newton device itself after mere seconds of building and downloading.

A feature I found to be very useful within NTK is the Inspector Window. Imagine an editor that interacts with you as you write and test code. Let's say you have an idea for a function you want to write in your Newton app. You can test it first in the Inspector (before you actually edit your Newton code). Any code you type in the Inspector Window is actually executed on the connected Newton device, and the result is returned immediately. For example, if I type

```
3 + 4 + 5
```

and then hit ENTER (not RETURN), I get

```
#IC 12
```

This means you can pre-test any algorithms and functionality you want before you even begin to write your application. The Inspector is the perfect place to play with NewtonScript and test out the new syntax and constructs.

What's Next

There are many other important topics to cover, including creating and accessing persistent data objects (called "soups"); using N

continued from page 2

Letter From the Editor

Toolkit 1.6 does this and the soon-to-be released Newton Toolkit for Windows will provide even more choice and flexibility. Second, providing libraries to enable desktop PCs and legacy systems integration is critical. The Desktop Integration Libraries (DILs) are the key to this functionality and developers have been working with them for several months now. Third, top notch training and support is also essential to ensure that the development community is delivering the most powerful solutions, in the quickest, most accurate way possible. Apple's Developer University provides the training and the DTS provides top notch support. All of these deliverables, combined with an award winning operating system and the promise of problem solving mobile solutions from third parties will continue to make the Newton the premier PDA platform.

This Newton roll out was about a great deal more than just Newton 2.0. It was about the evolution of an entire platform and all of its components. We're off to a great start and want to continue in the same direction by providing excellent service, support, and tools to some of our most important partners in this business – Newton

Newton 2.0 Developers Tell It Like It Is

It's In There!

by Todd Courtois, AllPen Software

The folks at Apple have asked me to give a candid glimpse at what it's like to develop an application under Newton 2.0. If you don't already know what it's like to develop an application under Newton 1.x, consider yourself lucky – but this glimpse probably isn't for you.

Newton 2.0 is heavier on the paradigm than Newton 1.x. The original Newton was a cute toy with a lot of potential: anyone could pick it up, plug it into NTK's Inspector, and see the potential it had. The new Newton requires the programmer to dive more deeply into the cranial fluid of a PDA – but once there, it's much easier (and more fun) to swim.

The first Newton application I wrote included animation, sound, and a completely interactive GUI – and I wrote it in less than a day. Then came the hard part: writing a *useful* application. As we thought of more and more killer applications for PDAs, it seemed to become more and more difficult to stretch the fabric of Newton to meet our goals. We screamed for better OS support for everything from communications to soups to desktop integration.

The result of our implementation anguish is Newton 2.0. We screamed for new features and speed improvements and, for the most part, Apple listened. Almost all of the obstacles to getting our job done right in the areas of communications, routing, data storage and retrieval, drawing, sound, and, yes, user input have been removed. Of course, we're already starting to discover new obstacles, but Newton 2.0's greatest achievement has been to obliterate the development problems we faced under 1.x.

Learning Newton 2.0 doesn't mean forgetting everything you already know about Newton development. Newton 2.0 is an extension of Newton 1.x; all of the things you liked about developing under 1.x have been enhanced, with the notable exception of the Inspector. Personally, I find it a bit more difficult to debug with the new NS Debug Tools than I did with the original Inspector (but then again, I became *really* good at debugging with the old Inspector). In time, I'll probably find all of these newfangled features – such as breakpoints and single-stepping – useful. And I suppose the return of Inspector over AppleTalk will make it easier to debug ADSP-based communications applications.

I think the biggest difference between Newton 2.0 and Newton 1.x has been the change in attitude and motivation from Apple that we've witnessed over the past two years. The Newton platform has grown to be a wide-open platform with a tight feedback loop between developers and the Newton engineering team. The thing that encourages me most about Newton's future is that the changes we kicked and screamed for are now "in there"...and I believe this trend will continue.

Backward Compatible = New User Features

by Rick Giles, Holosoft, Inc.

We see Newton 2.0 as providing a good platform for building the next generation of PDA applications. In addition to providing many new protos and functions, the operating system has very good backward compatibility, making porting a large application fairly painless. This has allowed us to focus more on adding new content to our applications rather than fixing incompatibilities. Compatibility was a very important issue for us since we wanted to provide our installed base with an upgrade while at the same time adding features that would take advantage of the new operating system. One such feature is the ability to rotate the screen, which is particularly useful for spreadsheet users.

We have found that the improvements in the soup architecture and communications are a plus for our applications. With Equate, we have pushed the envelope of how much data one can load into the Newton, and now we have extended the envelope further while getting better overall performance. One way we have taken advantage of these features is by allowing users to import Excel workbooks directly to the Newton. The ability to import workbooks means that users can now import multiple worksheets of any size simultaneously, which places a bigger burden on both communications and the soup architecture.

The improvements in communications provides the groundwork for our "Desktop Direct" software, which allows users to synchronize their Excel data to Equate without the need for Newton Connection Kit. The use of DILs and the new Newton 2.0-specific communications features are crucial to our desktop integration efforts.

Newton 2.0 – Go Ahead, Do It!

by Don Vollum, PelicanWare

Developing for Newton 2.0 after developing for Newton 1.x is like reaching adulthood: there are so many things you can do which you couldn't even consider before. Extend the Notes application? Sure. Work with large binary objects? No problem. Copy bitmaps into `paragraphViews`? Be my guest. Everywhere you look, there are APIs to do things which either couldn't be done at all under 1.x, or required some very marginal hack.

Even better, there are new protos and functions that reduce the workload tremendously and let your application be more responsive to the user. For example, `protoOverview` generates a very slick overview, without much more work than the `protoTable` I used to use. `DoProgress()` generates a detailed status indicator with a single function call, and gives the user the ability to cancel the action in progress.

The stationery concept allows you to both extend built-in applications and create your own extensible applications. We extended the Notepad to include our spreadsheet application, QuickFigure Pro, so that users can create spreadsheets without ever leaving the Notepad application.

Newton 2.0: A Revolution for Programmers

by Scott Gruby, QUALCOMM, Inc.

Newton 2.0 is viewed by many as a big step forward for users of PDAs. While this is true, Newton 2.0 is more of a revolution for developers of Newton applications. Under Newton 1.x, many developers had to individually implement functions that would provide overviews, access the Names application from within their applications, and deal with the cumbersome and mysterious communications layer. With Newton 2.0, many of these dreaded tasks are documented protos that can easily be added to any application. This not only makes it easier for developers to quickly generate applications, it provides a more unified interface for the user.

For most developers, the switch to programming under Newton 2.0 will be an easy one. Porting applications to take advantage of the new features offered in Newton 2.0 should be quite simple (in most cases), as long as you have followed documented methods. The hardest task in porting will be finding and removing some functions you have created and replacing them with documented protos. Once you've made the switch and are accustomed to the new features, you won't want to support pre-2.0. Since Newton 2.0 has a very small market right now, you might not want to abandon users of current 1.x products; however, I think that the benefits of Newton 2.0 outweigh the continued support of Newton 1.x. In addition, once users see Newton 2.0, I feel that they'll want to upgrade and have access to all the new features.

To me, two of the most exciting Newton 2.0 frameworks are protoTransport and NewtApp. The protoTransport allows a developer to create different backends that can be used from other applications. For example, mail transports can be created not just for eWorld, but for POP/SMTP mail, AOL, CompuServe, and so on. This allows the user to have access to all these services via the Mail routing action and the universal In/Out box. The transport interface also allows developers to externalize the communications layer of an application so that the application and communications layer can be developed independently (and therefore speed up compile times). Transport layers do not all have to be for mail, but that may be the most common use for them. NewtApp was designed to be like MacApp in terms of providing different pieces that a developer can link to create an application without having to create each individual piece. Even though I've never used MacApp, I can see how the model is very useful on the Macintosh. NewtApp handles so many things for you—such as registering soups and handling various messages automatically – that a basic, yet potentially powerful, application can be created within a very short time. In addition, NewtApp will make applications have more of a standard interface, making it easier for users to switch among applications.

Developing for Newton 2.0 will be exciting for all developers. The improvements in the operating system will make more users interested in the Newton OS, thereby creating a larger market for your applications. Make the leap to Newton 2.0; Newton 1.x users will have to be content with what they have now, as there is no sense in developing for an operating system that has many limitations. The future is bright for Newton 2.0 and beyond.

Go out and start developing for Newton 2.0! NTJ

Converting an Existing Application to Stationery

by Don Vollum, PelicanWare

Stationery is one of the most exciting features of Newton 2.0. Using stationery you can extend the built-in applications, let other developers use your data in their applications, and create powerful stand-alone solutions. This article discusses the issues involved in converting an existing Newton application from a stand-alone application into a piece of stationery for the built-in Notes application.

What is Stationery?

A piece of Newton stationery has two parts: the `dataDef` and the `viewDef`. The concept is that you give a container application a definition for a specific type of data (the `dataDef`), and then provide an editor or viewer to manipulate the data (the `viewDef`). The container application looks at the type of data, and then searches for an appropriate editor for that data.

The container application also deals with all issues related to storing your data, displaying an overview, and routing (you do need to provide appropriate routing formats). A good example of this is the Notes application. Essentially, the Notes application is a container for different types of stationery. Each built-in data type (note, outline, and checklist) provides a data description and an editor. Other built-in applications which support stationery include Names, Calls, and I/O Box.

A piece of stationery may be used by several different container applications. For example, Notes and I/O Box both can display the note, outline, and checklist stationery. If you route an outline from Notes, you can open and edit it within the I/O Box application.

All of the built-in applications which support stationery are built on the NewtApp framework. NewtApp provides most of the structure necessary to support and create stationery, although your stationery item does not need to be based on NewtApp. Still, a good understanding of NewtApp is critical to building stationery.

Why Stationery?

The first issue to consider is whether your application is appropriate for use as a piece of stationery. Stationery applications have both advantages and drawbacks when compared to their stand-alone equivalents. The advantages include user convenience (if you are extending Notes, for instance, the user need only tap the "New" button to access your application), a smaller memory and system resources footprint, and extensibility – other developers can use your stationery to extend their applications.

Drawbacks include less usable screen area (because of the screen space used by the container application), no programmatic access to your application (you can't access your base view by calling `getRoot().|myApp:mySig|` any more), and a lack of flexibility if your application doesn't use a one frame per data item storage metaphor.

Application Design Issues

Several design issues must be resolved. First, your application does not have control of the status bar. If you need to provide status-bar style buttons, you can do this through the `newtFloatingBar` proto, but it uses up screen real estate (it floats above your editor or viewer's space.)

Second, you can no longer access your application from the root. This can be an issue for routing, and also if you provide an API for other developers. In the case of my application, QuickFigure Pro, this was significant, since we provide an open API which allows other developers to script our application.

Generally, you need to be aware that your code could end up being executed in a different environment than you intended, so it needs to be very self sufficient. For example, the same `viewDef` could be called by either the Notes application or the I/O Box application (to view data received via beaming or mail). Therefore, you must not make any assumptions about the application that will contain your stationery.

Unless your stationery is built into a stand-alone application, you should consider building it as an auto-part.

Data Storage

Finally, you need to consider how your application stores data. The stationery system assumes that your data will be entirely contained in a single frame, and will usually be stored in a soup. This frame may be added into a soup directly, or may be combined with other frames and added to a soup.

As a general rule, the further your design is from NewtApp (one frame per document), the more difficult data storage will be. If you can store everything in one frame (and fit that frame into a soup entry), you're in good shape. If not... well, read on.

If you store your data in multiple soup entries (or in an entire soup, as was my case), there are two strategies you can pursue. The first is to write your data into a Virtual Binary Object (VBO) when you need to save it, and then attach that object to your soup entry.

VBOs are binary objects which are stored persistently in a specified store and attached to a soup entry. A VBO's size is limited only to the space available in the store on which it resides, making it perfect for working with large binary objects. When the soup entry to which the VBO is attached is deleted or moved, the VBO goes with it.

This makes VBOs an excellent solution for stationery items needing to manipulate lots of data. It minimizes the burden of keeping track of different pieces of data: when the container application deletes or

moves your soup entry, the VBO goes with it.

If your data doesn't lend itself to being manipulated in binary form, then VBOs have a significant drawback: performance. You can use the NewtonScript `translate` function to convert between frames and binary objects, but it's not particularly fast. If you have to write your data from a frame, into a VBO, you can have some serious performance problems. Use of native code can help a lot with these performance issues.

Implementing Stationery

DataDefs

If you are starting with an existing application, the primary piece of new code you will have to create is the `dataDef`. This should be a template based on the `newtStationery` proto. Most of the slots and methods related to `newtStationery` are self-explanatory; however, a few slots are especially important:

`symbol` – The contents of this slot link the `dataDef` to the `viewDef`. When the container application finds a piece of data with this symbol, it looks for a `viewDef` with a matching symbol (which is set when the `viewDef` is registered). This symbol must be unique, so use your registered signature.

`SuperSymbol` – This slot contains the symbol for the intended container application. However, your `dataDef` could end up in another container; for example, a piece of data intended for the paper roll could end up being viewed in the I/O Box application.

ViewDefs

Your `viewDef` is the editor for the data defined in the `dataDef`. If you are converting an existing application, you can probably use your existing main view. Keep in mind that the status bar, as well as anything at the top of the display should be deleted.

To convert your main view into a `viewDef`, you must add the following slots: `version`, `type`, `symbol`, and `name`. `Symbol` must be the same symbol as the one in the `dataDef`'s `symbol` slot.

Install and Remove Scripts

In your `InstallScript`, you need to call `RegisterViewDef` to register your `viewDef`, and `RegDataDef` to install the `dataDef`. If your application is an auto-part, you can use `GetLayout` in your `installScript` to get access to your main view (`viewDef`) and `dataDef`. For example:

```
partData := {};
partData.qfDataDef := GetLayout("QF DataDef");
partData.qfViewDef := GetLayout("QF Stationery 3.0 Main");

InstallScript := func(partFrame,removeFrame)
begin
  // add dataDef and dataView so it'll show up in paperroll new picker
  RegDataDef(EnsureInternal(kQFStationerySym),
partFrame.partData.qfDataDef);
  RegisterViewDef(partFrame.partData.qfViewDef,
kQFStationerySym);
end;
```

Note that if your application is not an auto-part, you must be careful about using `GetLayout` in your install script.

RemoveScript

In your `removeScript`, you need to unregister the `viewDef` and `dataDef`, along with any routing formats you have installed. For example:

```
RemoveScript := func(removeFrame)
begin

  //unreg the viewDef:
  UnRegisterViewDef('default,kQFStationerySym);

  //unreg the dataDef:
  UnRegDataDef(kQFStationerySym);

end;
```

When removing your stationery item, you should keep in mind that your `viewDef` could be open in another application while it is being removed – this is a major problem with roll-based applications, such as Notes. In this case, you need to notify all applications that your `viewDef` has been removed, so they can handle this gracefully.

Newton Systems Group is working on a solution for this problem, but it was not available at press-time.

Routing

Routing support for stationery items is handled no differently than for other Newton 2.0 applications. All you need to do is create your route formats, and register them appropriately. The container application will automatically provide duplicate and delete, and check for any route formats you have registered to handle your class of data.

For frame routing (beam and mail), you should ensure that your data requires no special processing when it is put away. Your stationery item has no control over the container's `PutAway` method.

NTJ

To send comments or to make requests for articles in Newton Technology Journal, send mail via the Internet to: NEWTONDEV@applelink.apple.com

PC Integration and Newton 2.0: First Class Connectivity for Mac OS and Windows Users

by David Glickman, Apple Computer, Inc.

Organize. Communicate. Integrate. Most likely, when hearing about Newton 2.0, you have come across this "mantra" that we live by in the Newton Systems Group. Newton 2.0 is not just a new operating system, but rather, a full suite of applications, developer tools, and third party solutions that make up the most powerful and flexible PDA platform today.

A key component of Newton 2.0 is the capability that the platform provides to let customers easily integrate and exchange information between Newton PDAs and personal computers – both Windows and Mac OS based computers. After listening to customers and developers alike, the Newton group set out to provide some key solutions in the area of integration:

- provide a simple, cost-effective solution to store the data from a Newton PDA on a PC.
- enable true synchronization of information between Newton PDAs and PCs.
- provide an easy method for gathering and publishing reference information stored on a PC to distribute and view on a Newton PDA.
- continue to offer innovative solutions to use Newton PDAs in conjunction with PCs – from remote access capabilities to the ability to use a PC keyboard with a Newton PDA.

The advantage of Newton 2.0 through PC Integration

The Newton 2.0 platform has been designed to provide the basis for a complete suite of solutions for integrating PDAs, PCs, and enterprise environments without changing the way people use their PCs. Built-in capabilities, along with add-on utilities and developer technologies from Apple and third parties, provide the functionality users require in Newton 2.0 – functionality for making their Newton PDA a powerful extension of their PC and enterprise environment, with the following advantages:

- Windows and Mac OS connectivity. Newton 2.0 provides full cross-platform connectivity, so that Newton PDA devices can be easily integrated into both Windows and Mac OS environments – including Windows 95. This capability is a key requirement of users and developers alike.
- Connectivity to an enterprise. Newton 2.0 has a sophisticated communications architecture that gives users easier access to their corporate computing environment, whether they're at a desk in their office building, or miles from headquarters. This architecture includes connectivity to a corporate network, as well as individual PCs or workstations.
- Data security. The Newton 2.0 platform, by providing seamless connectivity to PCs and a variety of backup and restore features, helps ensure the safekeeping of user information. This is especially useful for users working with information from remote locations.
- Productivity tools. The Newton 2.0 platform offers products and technologies that allow users to extend what they can do with information – for example, synchronizing it between applications, publishing electronic books, and performing PC connectivity tasks from remote locations.

To satisfy these needs, Apple and third parties are providing a range of solutions designed for Newton 2.0.

Desktop Integration Libraries (DILs)

Integrating data from the PC and Newton PDAs

Newton 2.0 incorporates a rich set of tools for developers, enabling you to create direct links between applications on PDAs and PCs so that users can synchronize PDA data directly with the PC applications they use every day. The key technology for this PDA-to-PC integration is Apple's Desktop Integration Libraries (DILs). DILs make the Newton PDA an extension of the PC platform. Simply put, DILs are application programming interfaces (APIs) that you can integrate into your PC applications, to let users access Newton PDA data from their PC. DILs help match the functionality of data used on a Newton PDA with the corresponding data in a specific application running on a PC. This is especially helpful when using personal information management (PIM) software, such as appointment-scheduling applications, because it allows users to keep the data in PDA PIMs current with data in a PC PIM. DILs also enable import and export of names and dates information.

The flexibility and simplicity that DILs contribute to applications make Newton 2.0 an even more attractive platform in an enterprise environment. To learn more about DILs and receive information about how to obtain this technology, contact the Newton Developer Relations Group at NEWTONDEV@applelink.apple.com.

Newton Press: Electronic publishing on Newton PDAs

Many developers are probably familiar with Bookmaker, a tool for creating Newton Books. With Newton Press, we have delivered this functionality to the end user and we therefore see the proliferation of Newton Books growing exponentially. This will increase the opportunities developers have to create commercial quality electronic books, since customers will have a greater appreciation of the value of

Newton Books.

Newton 2.0 software gives users an easy way to combine different types of information into an electronic document and distribute it on PDAs. Through Newton Press software, mobile professionals and corporate customers can use the combined power of PCs, enterprise information, and Newton PDAs to publish and distribute electronic books. Newton Press lets users "drag and drop" text blocks, word processing files, e-mail messages, and graphics onto the Newton Press icon on their PC screen to compose an electronic book for personal use – or to publish it for an entire group of Newton PDA users. On an enterprise-wide level, this is especially useful, since documents can be easily and quickly distributed to whole teams – documents such as manual updates, price lists, approval forms, and the like.

Newton Press is a flexible tool. It can integrate any imported word processing, text, or graphic document supported by Claris XTND technology (for the Mac OS). It supports all of the most popular word processing and graphics applications on the Windows platform, including the following:

- Microsoft Word v1.x, 2.0, and 6.0
- Interleaf Publisher v1.1, 5.2, and 6.0
- PC Paintbrush
- Windows Bitmap
- WPGI (Word Perfect vector)
- ASCII

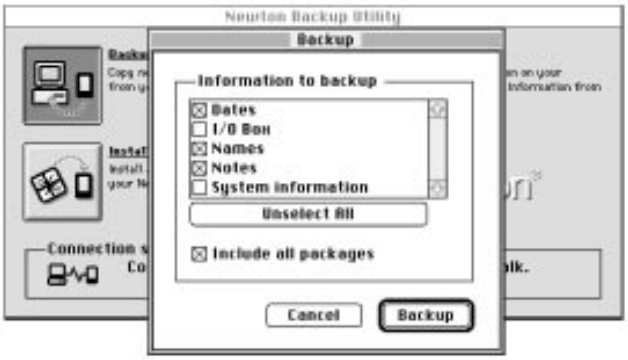
Newton Press running on Windows 95

Data backup and restoration with Newton Backup Utility

Research shows that data security is a key concern for PDA users. The Newton 2.0 platform offers a simple solution for backing up Newton PDA data. It's easy to back up information residing on a Newton PDA onto a PC running either the Mac OS or Windows software – and later restore it to the Newton PDA.

The Newton Backup Utility (NBU) lets users connect a Newton PDA to a PC – and use the PC as a place to store PDA information. Users with a Mac OS-based computer connect their Newton PDA via the serial port or LocalTalk port; Windows software-based systems can be connected using the serial port.

By providing NBU free of charge to users (it's included with the MessagePad 120 and available online for other Newton PDA users), we are able to ensure that all customers will have a method to maintain the integrity of their data. In addition to all the built-in software in Newton 2.0, NBU will back up and restore all third party information as well.



Newton Backup Utility running on Mac OS

Imagine a Newton PDA user who is in a rush to catch a plane, and wants to back up important names and dates data before leaving. Using NBU, users simply tap the "Backup" button on the Newton PDA screen, and choose where on the PC they want to store the information. To retrieve data that's been backed up, users simply click the "Restore" button on the Newton PDA screen. NBU also allows users to selectively restore any part of that information from a backup file – saving significant time. Or, they can install software packages from their PC.

The Newton Backup Utility allows users to download packages into a Newton PDA.

After clicking the "Install Package" button on the Newton PDA screen, users are presented with a standard open dialog box, and can select the packages they wish to download. The data is stored in the Extras drawer.

Looking Toward the Future

We believe that providing you with tools and solutions such as the DILs and Newton Backup Utility is a great start to establishing Newton 2.0 as the PDA platform for PC integration. However, we have not stopped here – we are continually working to bring developers and end users more integration solutions.

In the future, as the Newton platform matures, you will see solu

Newton Platform Market Segmentation and Positioning – A Useful Tool for Solutions Marketing

by Lee Dorsey, Apple Computer, Inc.

It is a common event in the high-tech industry to find companies creating innovative new technologies, throwing them out on the market, and just waiting to see who will bite and exactly how they will be successful. Sometimes they win, and sometimes they don't. Most often, they don't. However, as the computer industry becomes more and more mature, successful players are finding that they need to employ more of a text-book approach to defining marketing strategies and methodologies in order to understand their markets and customers and to compete more effectively. In the Newton Systems Group, we've gone through market analysis exercises to define the PDA market, proactively identify and target our market opportunities, and define the segments where the Newton platform offers real competitive advantages.

The marketing team in the Newton Systems Group has segmented the market and identified two market opportunities where the Newton platform best meets the needs of the target customers. In Marketing 101, this approach is known as "needs-based" marketing. The segment(s) are identified, the product is positioned with respect to the unique needs of the target customer in the segment and the benefits it offers to that customer, and the product is marketed in a way that demonstrates its ability to satisfy the needs of the identified customer.

As the platform marketing group at Apple works to make the Newton Platform successful for two key target markets (mobile business professionals on the horizontal side and corporate and institutional customers on the vertical side), third-party solutions developers should target a subset of the same customers and appeal to the same needs in order to sell solutions. It will be a critical exercise for solutions providers to go through the steps of segmenting their markets within those that the Newton Systems Group has identified, qualifying the target customer and his/her needs, and then marketing the product based on its ability to satisfy those needs. Understanding the customers and positioning that Apple is going after from a platform perspective will be essential. We've provided you with the Platform Positioning Matrix shown below to help in your strategic market planning. Use the information to help you position and talk about your product with respect to the Newton Platform and its target

	Horizontal	**Vertical**
Platform Target Customers	Mobile Professionals (Business people who spend considerable time away from their desks and have a need to organize and communicate information, and integrate information on their Newton devices with information on their Mac OS or Windows personal computers.)	Corp./Institutional Customers (Business, Government and Education - Customers who need specialized applications)
Audiences	Mobile Professionals Channel ISVs Media Key industry influencers	Corp./Institutional Customers Channel ISVs, SIs, VARs Media Industry/financial analysts Apple Internal
Platform Category	PDA	PDA
PDA Definition	A new generation of hand held devices based on advanced technology and designed from the ground up to meet the needs of business professionals and organizations. Newton PDAs provide a full range of personal computer	A new generation of hand held devices based on advanced technology and designed from the ground up to meet the needs of business organizations and education institutions. Newton PDAs provide a full range of mobile solutions for key markets such as health care, sales force, field service and education customers

	integration, organization, and communications solutions.	**Vertical**
Platform Definition		The Newton platform comprises the Newton operating system, development environment, connectivity solutions, applications and peripherals from Apple and third-parties, and devices from a variety of manufacturers in the telecommunications, consumer electronics, and computing industries.
Overarching Platform Positioning		Newton leads the industry with the most advanced, open, easy-to-use PDA platform with the best mobile solutions on the market today. Newton leads the industry with the most advanced, open, easy-to-use PDA platform with the best mobile solutions on the market today.
Key Platform Messages		• Powerful platform for software development The Newton platform provides cross-platform, high-productivity tools for rapid software development. • Complete developer support infrastructure Development on the Newton Platform is supported through developer programs, co-marketing opportunities, training, and technical support. • Seamless integration into enterprise computing environments A range of products have been developed for the Newton Platform to give customers access to desktop applications, networks, Internet, and corporate databases
Platform Positioning		The Newton platform is for business, government, and education customers requiring a powerful communications-enabled platform to deploy custom hand held vertical market solutions for individuals, departments, and enterprises.

NTJ

New Logo Licensing for Newton Developers

by Lee Dorsey, Apple Computer, Inc.

Those of you who attended the Newton Platform Development Conference in September got a sneak preview of some of Apple's plans to change Newton branding strategies for third-party solutions providers. The changes are now final and official, and if you haven't already heard from Apple's Software Licensing Group, you'll want to contact them at SW.LICENSE@applelink.apple.com or (512) 919-2645 to learn more. The basic changes are outlined here for your convenience.

So what's changed? First, and most important, the circular "Newton Compatible" logo (Figure 1) that many developers licensed for use on packaging along with the NTK license agreement has been terminated. That means that you should no longer be using the mark on packaging, documentation, or media.

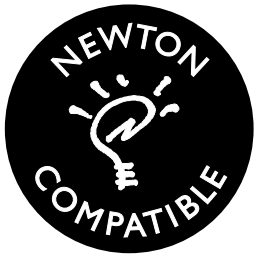


Figure 1.

Why did we terminate this mark? We found that the logo didn't allow for future growth and changes in the platform operating system without causing customer confusion or modification of the mark. Customers might experience a great deal of confusion upon finding a "Newton Compatible" application on the shelf, only to find that it really isn't compatible with ALL versions of the Newton operating system. The mark would need to be updated for each version of the Newton operating system released, causing developers to have to re-print packaging or cover the old marks with new ones. We wanted to move third-parties to a more universal mark that customers could easily identify and will work in the long-term without changes each time we rev the OS.

In place of the "Newton Compatible" logo, we have moved to the use of the original vertical Newton Signature (Figure 2) by third-parties, which comprises the familiar light bulb logo and the word Newton.

Figure 2.

For the first time, third-parties will be able to license the Newton Signature for use on their packaging and product promotions. The Newton Signature is a highly recognizable mark that carries with it a great deal of brand equity in the market place. We'd like to extend the use of the mark to third-parties to identify their applications and solutions as platform products, and to leverage the equity that Apple has built around the logo. The Newton Signature logo should be used in combination with system requirements on the solution packaging. Customers will then know to look for the familiar Newton logo on both hardware and software platform products for Newton PDA devices. They will then be able to identify its operating system compatibility by the system requirements on the packaging. For a no-fee license agreement and guidelines for use of the mark on packaging, contact Software Licensing at SW.LICENSE@applelink.apple.com.

We have also developed a logo strategy to help you communicate to customers about your adoption of new technologies in your solutions. The first of these, seen below (Figure 3), will allow you to communicate to customers that your solution works on Newton 2.0.

Figure 3.

Whether you've got a 1.x application that is compatible with Newton 2.0 or have created a new application from scratch for Newton 2.0, use of this mark will clearly identify it for customers as a product to use on Newton 2.0 PDA devices. This was important, given the number of changes that occurred in the operating system and the number of applications that used undocumented calls under 1.x, thereby making them incompatible with Newton 2.0. Use of this mark will make it easy for developers and customers to communicate which products they should buy and run on new Newton 2.0 PDAs, whether it's an original 1.x application or a new 2.0 application.

Like the Newton Signature logo, this "Works with Newton 2.0" logo is easy to obtain and use. You may request a no-fee license agreement from Software Licensing. Part of the agreement is a technical criteria survey, against which you must test your application. Once your product meets the technical criteria, it qualifies for use of the mark. We encourage all developers to test their application against the survey criteria and to use the mark whenever possible. Additionally, by licensing the mark, your product will automatically be placed on a list of applications for Newton 2.0, which will be made widely available to Newton customers via on-line postings, fax-back services, and other distribution channels. Use of the logo is a great way to educate your customers that your product is one that they should use on a Newton 2.0 device and to gain exposure for your product in Apple's marketing channels.

Questions on any of these logo programs can be directed to Software Licensing or to the Newton Developer Relations Group at NEWTONDEV@applelink.apple.com. NTJ

To request information on
or an application for
Apple's Newton developer programs,
contact Apple's Developer Support Center at
408-974-4897
or Applelink: DEVSUPPORT
or Internet:
DEVSUPPORT@applelink.apple.com

Apple Drops 1% Royalty Requirement on Newton Software Sales

by Staff, Apple Computer, Inc.

In a continuing effort to make the Newton platform the most attactive PDA platform for software development, Apple announced at the recent Newton Platform Developer's Conference that it was dropping the 1% royalty requirement on all Newton software sales. Previously, all developers creating commercially available software for the Newton platform needed to pay Apple Computer a 1% royalty on their total product revenues.

"Software developers and compelling third-party software solutions are key factors that have established Newton as the premier PDA platform," said Rick Fleischman, Product Line Manager for Newton Tools at Apple. "Removing the royalty requirement on third-party software sales sends a clear message that we value our developers and the work they do in supporting the Newton platform."

Removing the 1% royalty requirement is only one example of the many ways that Apple has been working to increase the opportunity and decrease the costs for Newton software developers. Other recent changes include: dramatic decreases in the pricing for both Newton development support and development tools; opening up the previously confidential byte code and package formats for third-party use; the addition of a compiler and profiler to Newton Toolkit, enabling developers to acheive higher performance in their Newton applications; and the release of the Newton 2.0 platform, providing a much richer platform for software development. In the future, Newton developers can look forward to development tools hosted on the Windows platform as well as the Mac OS platform, and tools to enable developers to add routines written in C or C++ to

NTJ

Newton 2.0 Programming Courses

from Newton Systems Group Developer Training

Newton 2.0 is a major revision of the operating system and the result of extensive customer feedback and user testing. The Newton Developer Training group has been busy developing new courseware for developers who want to take advantage of the Newton 2.0 operating system. We are proud to present two new courses complementing the Newton 2.0 platform: *Newton Programming: Essentials 2.0* and *Newton Programming: OS Enhancements.* Soon we'll be releasing self-paced training modules for Newton 2.0 Communications and Extended Topics, and possibly other areas, depending on developer interest.

What does Newton Programming Courseware provide that books and source code don't? Well, have you ever found that reading source code and documentation provided you with the content you needed but not necessarily the context? Or, as you were learning about a new concept, have you had sophisticated or contextual questions that the written material couldn't answer? Imagine spending an entire week steeping yourself in the technology you are committed to mastering with virtually no interruptions. Our courses are designed for developers by developers, with the goal of providing not only the technical information you need, but also background information and an understanding of how the technology is meant to work. Our instructors are experts in their fields, and if you manage to stump them they will do everything they can to find you the answers. Our courses are designed to cut your learning and development time while providing you with the expertise you need to write useful and powerful applications.

Students who attend Newton Programming courses should expect to spend a productive and demanding week in class, since they often are learning a new technology, a new operating system, and a new language over the course of five days. Students spend much of their time doing hands-on development and debugging, and have continual access to a highly qualified instructor. Participants will also spend a session with the Newton Developer Support team and have the opportunity to ask technical questions.

Students who attend *Newton Programming: Essentials 2.0* will learn how to write, test, and debug fully functional Newton 2.0 applications using Newton Toolkit, NewtonScript, and NewtApp. Any programmer with object-oriented development experience is welcome to attend the *Essentials 2.0* class.

For those developers already experienced in Newton 1.x programming, *Newton Programming: OS Enhancements* provides migration support to the Newton 2.0 operating system. This class is designed to bring 1.x programmers quickly up to speed in Newton 2.0 technology and assist them in converting their 1.x applications. Students will learn the differences between 1.x and 2.0, how to write applications that take full advantage of Newton 2.0, and how to move existing 1.x applications to the 2.0 platform. Students are encouraged to bring their existing 1.x applications to class and begin converting their code to Newton 2.0.

Call **(408) 974-4897** or e-mail DEVUNIV@applelink.apple.com to register!

NEWTON PROGRAMMING: ESSENTIALS 2.0

Course Description

Learn how to develop applications for Newton using NewtonScript and NewtApp, and how to develop and optimize your Newton 2.0 applications using Newton Toolkit 1.6.

Schedule for the Week:

Monday
- Newton 2.0 System Overview
- NewtonScript
- Structure of an Application
- Newton Toolkit

Tuesday
- Understanding Inheritance
- Protos
- Programming with NewtApp
- Lab Time

Wednesday
- The View System and Justification
- Lab Time
- Lunch with DTS
- Lab Time

Thursday
- Debugging and Tools
- Soups and Stores
- Rich Strings
- Lab Time

Friday
- Stationery
- Routing
- Lab Time

This course will provide you with all the knowledge and hands-on

experience you need to begin building powerful Newton 2.0 applications. You will learn NewtonScript, the programming language of the Newton OS, in addition to Newton Toolkit, which provides you with a graphical interface and easy access to powerful, reusable components. Using NTK you will interactively develop your applications without having to execute sequential edit, compile, and link cycles. You will also learn to program with NewtApp, a new application framework designed to help you build complete, high-quality, and full-featured Newton applications quickly and easily. A brief overview of Newton communications is also provided.
Length of class: 5 days from 8:30 am to 5:30 pm
Prerequisites: You must have developed a complete application in an object-oriented programming language and have basic familiarity with the use of a Macintosh computer and a Newton MessagePad.
Dates: January 15–19, February 12–16, March 11–15
Tuition: $1500

NEWTON PROGRAMMING: OS ENHANCEMENTS

Course Description

Here's your opportunity to learn about the many changes in all areas of Newton System Software for version 2.0, how to write a 2.0 app, and how to convert and make your 1.x Newton apps 2.0 savvy.

Schedule for the Week:

Monday
- Newton 2.0 System Overview
- NTK 1.6
- Compatibility
- Programming with NewtApp

Tuesday
- Data Storage and Retrieval
- Find and Filing
- Stationery
- Lab Time

Wednesday
- Routing
- Extending your Application
- Lunch with DTS
- Lab Time

Thursday
- Miscellaneous Topics
- Rich Strings
- Lab Time

Friday
- Desktop Integration Libraries
- Communications
- NTK 1.6 Advanced Features

The *Newton Programming: OS Enhancements* course provides a comprehensive overview of what is new and what has changed in system 2.0, focusing on those programming interfaces that you'll be most interested in as a developer. During this course you will learn how to take advantage of Newton 2.0 features including NewtApp (a new, easy-to-use application framework), Stationery, new Protos, new functions, Routing, and improvements in Text Input, Recognition, and Soups. This course focuses on compatibility and taking full advantage of 2.0 features. Be sure to bring your 1.x applications and take this opportunity to begin converting your code to 2.0.
Length of class: 5 days from 9:00 am to 5:00 pm
Prerequisites: You must have developed a Newton 1.x application, completed the course *Newton Programming: Essentials,* or read *Programming for the Newton* by Julie McKeehan and Neil Rhodes.
Dates: January 29–February 2, February 26–March 1
Tuition: $1500

Newton Programming Self-Paced Training

Soon to be released are Newton Programming Self-Paced training courses, which will provide you with convenient and inexpensive access to Newton Developer training. Included with this training are technical articles, step-by-step coding labs, illustrations, demos, sample code, and online assistance. NTJ

continued from page 1

Extra Extra: Extras Drawer Features in Newton 2.0

help book that you want to file into the Help folder. In Newton Toolkit you can use the `SetPartFrameSlot` compile time function to add a slot to the part frame that is generated at build time. The symbol for the Help folder in the Extras Drawer is `'_help`. So, to set the default folder for the help book to the Help folder, you would add this line to a text file (or even an `evaluate` slot) in your project:

```
// put the part in the Help folder by default
SetPartFrameSlot('labels, '_help) ;
```

When the OS loads the package, it will go through each part frame. Some slots from the part frame will be copied over to the icon structure used by the Extras Drawer. In this case, the `labels` slot is used to specify the initial folder.

For packages that contain multiple parts, you should set the `labels` slot of the particular part you want to be filed. You can only do this in the project that build the particular part. In other words, if you have an application that includes a help book, you will probably have two projects. One is used to build the help book, the other may be used to build the application. The second project would include the output (the .pkg file) from the first project. Since the help book is the part you want to file, you would set the labels slot of the help book in the help book project. Using `SetPartFrameSlot` in the application package, would set the `labels` slot of the application, not the help book.

Under rare circumstances, you may need to file parts into other default

folders. The default Extras Drawer folders and their symbols are listed in Table 1.

Folder	Symbol
Unfiled	NIL
Extensions	'_extensions
Help	'_help
Setup	'_setup
Storage	'_soups

Table 1. Symbols for Extras Drawer built-in folders

Having said all this, you should carefully consider where you are placing icons. In general, not setting a labels slot is the correct thing. The icon will show up in the Unfiled icons drawer and the user will be able to file it where they wish.

API's

There are two important API's that are not included in the beta version of the Newton Programmer's Guide. First, SetExtraInfo used to be implemented as a platform file function for the 1.x world. It is now a method of the Extras Drawer and has expanded functionality. Second, buttonToggleScript is part of the new way that applications are opened from Extras Drawer.

SetExtrasInfo

In 2.0 you can send the SetExtrasInfo message directly to the Extras Drawer. However, for SetExtrasInfo to work, your icon must have an app slot that corresponds to your appSymbol. For form (that is, application) parts, this slot is created for you and is set to the kAppSymbol constant. For other types of parts you have to manually add the slot.

To create the app slot, you can use the SetPartFrameSlot call in your Newton Toolkit project. However, in order for the OS to check for the app slot, you must also have a text slot. The text slot is used for the name in the Extras Drawer. So for an auto part, you would minimally need to do the following:

```
// give autopart an app slot so SetExtrasInfo can access it
SetPartFrameSlot('text, kAutoPartName) ;
SetPartFrameSlot('app, kAppSymbol) ;
```

One side note: there is no programmatic way to file an auto part into the Unfiled folder. Doing so will currently throw an exception and not move the part. You can file it in any other folder. If you need an auto part interface in the Unfiled drawer, you can use a Script Icon (see below.)

Once you have an app slot in your icon, you can call SetExtrasInfo. However, the call has changed a bit since 1.x. The syntax is:

```
SetExtrasInfo(paramFrame, newInfoFrame);
```

The first argument is now a frame. Specifying an appSymbol still works, but this is just for compatibility. Expect that to change in the next major release of the OS. The more slots you specify, the faster the call to SetExtrasInfo will execute.

Slot	Req/Opt	Value
appSymbol	Required	application symbol, must be the same as the app slot of the icon
store	Optional	store that the icon resides on
packageName	Optional	package name for the icon

Table 2. Slots for the paramFrame argument to SetExtrasInfo

The second argument is a frame that specifies what to change in the icon. You can change multiple things with one call.

Slot	Type	Effect
icon	Bitmap	Changes the icon to the new bitmap
text	String	Changes the text under the icon to the new string. It can be a Rich String.
labels	Symbol	Files the icon in the new folder. The symbol must be a valid file folder symbol, not NIL.

Table 3. Slots for the newInfoFrame argument to SetExtrasInfo

So to change the text of an application icon to the string "wiggy", you would use the code:

```
local ed := GetRoot().extrasDrawer ;
local paramFrame := {appSymbol: kAppSymbol,
    packageName: kPackageName};
local newInfoFrame := {text: "Wiggy"};

ed:SetExtrasInfo(paramFrame, newInfoFrame) ;
```

You can further limit the search that SetExtrasInfo needs to make by specifying the store your package is on. Since the user can move your package, you need to find out the store each time you make the call. You can find the store by using GetVBOStore in combination with ObjectPkgRef.

ObjectPkgRef takes a reference in your package and returns the package ref. Since packages are stored as VBO's in 2.0, you can then use GetVBOStore to get the store. Note that you must make sure the argument to ObjectPkgRef is a reference.

```
local ed := GetRoot().extrasDrawer ;
local paramFrame := {appSymbol: kAppSymbol,
    packageName: kPackageName};
local newInfoFrame := {text: "Wiggy"};

// the proto of a view should be a template in the package.
local myStore := GetVBOStore(ObjectPkgRef(self._proto)) ;

if myStore then
  paramFrame.store := myStore ;

ed:SetExtrasInfo(paramFrame, newInfoFrame) ;
```

buttonToggleScript

In 1.x, when the user tapped an icon in the Extras Drawer, the OS would send the Toggle view message to the base application view. In 2.0, this is no longer the case. The reason for the change is that any application can now be the background. If the user taps the icon for the background application, we may want different behavior from tapping the icon of a non-background application. In addition, if the launched application has a splash screen and takes some time to come up, it is possible for the user to get two taps on the Extras Drawer icon. The result is that your application base view can come up over your splash screen.

To take care of this circumstance, the OS send the `buttonToggleScript` message to your base application view if it is defined. The message gets sent each time the user clicks on your icon in the Extras Drawer. The syntax is:

```
buttonToggleScript(top)
```

`top` is the topmost application view. In general you will never use this parameter. If you return true from your `buttonToggleScript`, the default system action will not happen. You will be responsible for taking appropriate action based on the current blessed application and your application's current state.

The most common use of this script is to make sure your splash screen stays on top until it is closed. Assume you have a slot in your base view called `mySplash` that points to your splash screen when it is open, and is set to NIL when your splash screen closes. Then your `buttonToggleScript` would look like this:

```
myBase.buttonToggleScript := func(top)
begin
  // if the splash is open, make sure it is frontmost
  if mySplash then
  begin
    mySplash:MoveBehind(nil) ;
    true ;
  end;
end;
```

Special Icon Types

The 2.0 OS has added two specialized icon types. Script icons let you specify an icon that executes a specified function when the user taps it. Soup icons let you group more than one soup under one icon.

To use the special icons you need to know about a few functions:

```
kAddExtraIconFunc(extraType, paramFrame, pkgName, store)
```

This is a function defined in the 2.0 Platform file. It will create an icon of type `extraType` on the specified store. `pkgName` is a unique identifier for the icon which may just be your application package name.

It is important to remember that `kAddExtraIconFunc` does not check if your icon already exists. If you add your icon four times, there will be four copies of it in the Extras Drawer. You must use `GetExtraIcons` (see below) to check if your icon exists before adding it.

`extraType` a symbol for the type (`'scriptEntry`, `'soupEntry`)
`paramFrame` parameters used in creating the icon (see below)
`pkgName` name of package to associate with the icon
`store` store to store the icon on

If the `pkgName` is the same as your main package, the special icon will be removed when your main package is removed.

```
extrasDrawer:GetExtraIcons(extraType, pkgName, store)
```

This method of the Extras Drawer returns an array of all icons of type `extraType` on the specified store that have the name `pkgName`. `pkgName` is the name assigned in `kAddExtraIconFunc` in the `pkgName` argument. If no icons are found, it returns an empty array. If an invalid `extraType` is specified, it returns NIL.

`extratype` a symbol for the type (`'soupEntry`, `'scriptEntry`)
`pkgName` name of package to associate with the icon
`store` store to search for the icons

IMPORTANT: do not rely on the format of the items returned in the array. They could drastically change in the future.

```
extrasDrawer:RemoveExtraIcon(extraIcon)
```

This method is used to remove a special icon that has been previously added. The `extraIcon` argument is an element from the array returned by `GetExtraIcons`.
`extraIcon` element of array returned by GetExtraIcons

paramFrame Common Slots

The `paramFrame` argument to `kAddExtraIconFunc` is similar to a part frame. You can specify the icon bitmap, the text shown and other slots. Most of the slots are common to Script and Soup icons. See Table 4 for a list The ones that are different will be covered below.

Slot	Req/Opt	Description
`text`	Required	string for the Extras Drawer icon
`icon`	Recommended	icon to show in the Extras Drawer
`app`	Recommended	symbol used by SetExtrasInfo to find your extras icon
`labels`	Optional	symbol for folder icon should appear in
`ownerApp`	Optional	symbol of owner app for soupervisor (see below)

Table 4. Common slots for the paramFrame

Script Icons

A script icon is just a visible wrapper for a function. When a user taps on the script icon, the closure associated with it is called. The main use for this icon type is to open some user interface for a transport.

Script icons are created by passing the symbol `'scriptEntry` to `kAddExtraIconFunc` as the `extraType`. The tap action is a function of no arguments. Note that the closure is currently stored in a soup.

You should make it as small as possible. To specify the tap action add a `tapAction` slot to the `paramFrame`.

In general, the `pkgName` for the script icon will be the same as the package that it accesses. For instance if you are adding a transport with a package name of "myTransport:SIG", you would use the same package name for the script icon. This allows the OS to remove the script icon whenever your main package is removed.

The code below assumes that you are writing an auto part that defines a user interface slip in the layout "MySlip.t", and that there is a resource file called "pictures" in the same directory as the project that contains a PICT resource named "myIcon". The tap action will open up the slip defined in the MySlip.t layout.

```
// a symbol for the configuration slip
DefConst('kMyConfigSlipSym,
   Intern("configSlip:" & kAppSymbol));


// get the icon picture
r := OpenResFileX(HOME & "pictures");
DefConst('kMyScriptIcon, GetPictAsBits("myIcon", nil));
CloseResFileX(r);

// name for the script icon in the Extras Drawer
constant kScriptIconName := "ScriptIcon Slip" ;

// the tap action, small and simple
DefConst('kTapScript, func()
   GetGlobalVar(kMyConfigSlipSym):ButtonToggle()
)

// Construct the paramFrame for the AddExtrasIcon call
DefConst('kScriptIconParamFrame,
   {text: kScriptIconName, // name in the Extras Drawer
    icon: kMyScriptIcon,     // icon in the Extras Drawer
    app: kAppSymbol,           // so can call SetExtrasInfo
    tapAction: kTapScript    // call when icon is tapped
    }
)

InstallScript := func(partFrame, removeFrame)
begin
   local mySlip := GetLayout("MySlip.t") ;

   // install the slip
   DefGlobalVar(
      EnsureInternal(kMyConfigSlipSym),
      BuildContext(mySlip));

   // install my script icon for accessing the Config slip
   local ed := GetRoot().extrasDrawer ;

   // get my store by using a reference in the package
   local myStore := GetVBOStore(ObjectPkgRef(mySlip)) ;

   // only add if it is not already there
   if Length(
      ed:GetExtraIcons('ScriptEntry, kPackageName, myStore))
         = 0 then
      call kAddExtraIconFunc with('ScriptEntry,
         kScriptIconParamFrame,
         kPackageName,
         myStore) ;
end ;


RemoveScript := func(removeFrame)
begin
   // remove the slip
   UnDefGlobalVar(kMyConfigSlipSym);

   // NOTE: we do not have to remove the script icon since it is associated with our package...
   // it gets removed when we do.
end;
```

Soup Icons

It used to be that you needed a third party utility to see and edit soups on Newton. On 2.0 all you do is go to the Storage folder in the Extras Drawer. For some applications this is all that is needed since the soup definition frame allows you to specify a user visible name for your soup. However, if your application uses multiple soups, this may not be enough. To help you with this, 2.0 provides a Soup Icon that represents a collection of different soups.

In Figure 1 you can see the Storage drawer with lots of individual soups for the UberSoup application. Figure 2 shows you what a Soup Icon can do for you. Notice that there is now just one icon labeled "UberSoup Items."

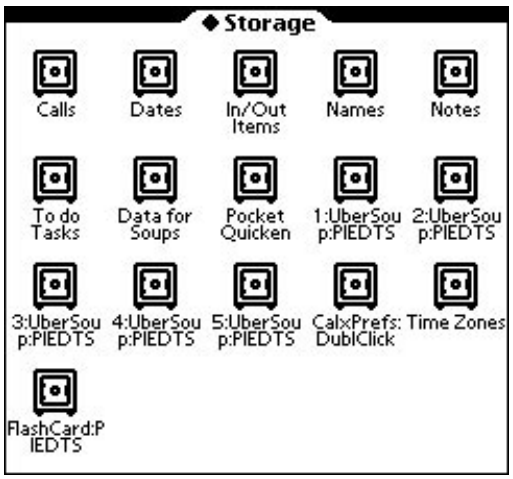


Figure 1: Storage folder with 5 of UberSoups

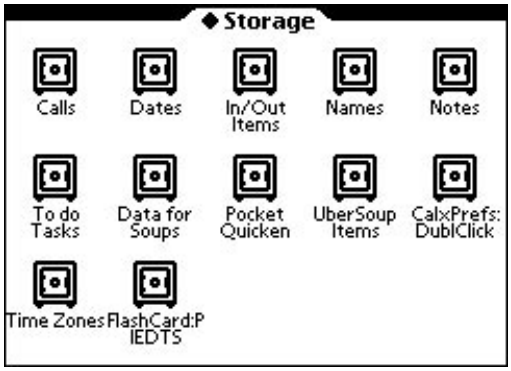


Figure 2. One icon for lots of UberSoups

Creating a soup icon is somewhat different from creating a script icon. One big difference is that you do not want your soup icon removed when your application is removed. Consider the situation where you use union soups and your application is on a storage card. If the card is removed, your soups may still exist on the internal store. If your soup icon is removed with your package, all your component soups will show up.

To prevent this you need to install your soup icon in the internal store and give it a different package name. However, you will eventually want to delete your soup icon. A good place to do this is your `deletionScript` which is called only when your application is actually deleted (e.g., scrubbed) by the user.

The only addition to the `paramFrame` is a `soupNames` slot that contains an array of soup names to be encapsulated by the soup icon. The `extraType` for `kAddExtraIconFunc` is `'SoupEntry`. The code below is from the package that was used to produce the screen shots in Figure 1 and 2. Figure 1 is the package installed without the soup icon, Figure 2 is with the soup icon installed.

```
// some useful constants
constant kSoupName := kPackageName ;

// the soup names
DefConst('kSoupNamesArray,
   call func()
   begin
      local result := Array(5, nil) ;
      for soupID := 1 to 5 do
         result[soupID - 1] := soupID & ":" & kSoupName ;
      result;
   end
   with ()
)

//  name for soup icon
DefConst('kUserSoupname, kAppName && "Items");

// package name for the soup icon
DefConst('kUberSoupPackageName,
   "Soups:" & kPackageName) ;

// Param Frame for the soup icon.
DefConst('kExtraIconParamFrame,
   {soupNames: kSoupNamesArray,   // soups to encapsulate
    text: kUserSoupName,          // user name for icon
    ownerApp: kAppSymbol,         // soupervisor hook
    app: kAppSymbol,              // access via SetExtrasInfo
    }
)

// InstallScript - install the Soup icon
InstallScript := func(partFrame)
begin
   // register the soups
   foreach soupName in kSoupNamesArray do
      RegUnionSoup(kAppSymbol,
         {name: soupName,
          ownerApp: kAppSymbol,
          ownerAppName: kAppName,
          indexes: '[]});


   // Add Soup icon
   local ed := GetRoot().extrasDrawer ;
   // always put on internal store
   local theStore := GetStores()[0];

   // make sure soup icon not installed
   if Length(
   ed:GetExtraIcons('SoupEntry, kUberSoupPackageName,
      theStore)) = 0 then
      call kAddExtraIconFunc with
         ('SoupEntry, kExtraIconParamFrame,
          kUberSoupPackageName, theStore);
end;

// DeletionScript to remove the soup icon
SetPartFrameSlot(
   'DeletionScript,
   func()
   begin
      // remove the constituent soups
      foreach store in GetStores() do
         foreach soupName in kSoupNamesArray do
         begin
            local soup := store:GetSoup(soupName) ;
            if soup then
               soup:RemoveFromStoreXmit(kAppSymbol);
         end ;

      // remove the  soup script icon
      local ed := GetRoot().extrasDrawer ;
      // always on internal store
      local theStore := GetStores()[0];

      foreach icon in ed:GetExtraIcons('SoupEntry,
      kUberSoupPackageName, theStore) do
         ed:RemoveExtraIcon(icon);
   end) ;
```

SOUPERVISOR

Tapping on a storage icon in the Extras Drawer, gives you a slip like that in Figure 3. This slip is part of the Soupervisor API. With the slip as shown, it is only possible to delete all items from all of the component soups. However, Figure 4 shows a similar slip with a filing button. This allows the entire contents of the union soup to be moved to a different store and/or filed to a particular folder as in Figure 5.

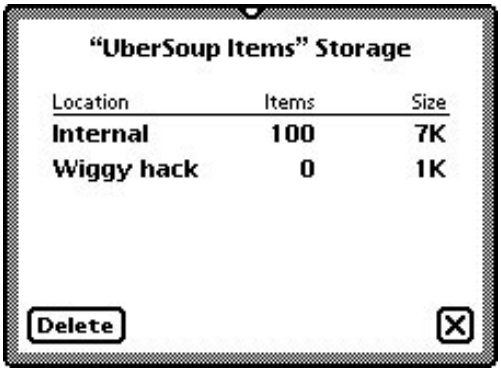


Figure 3. The Soupervisor Slip

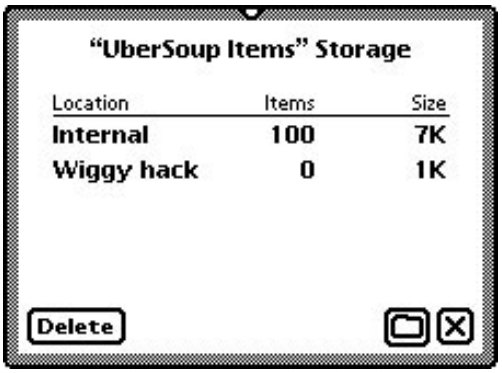


Figure 4. Soupervisor with Filing

Figure 5. Filing and Moving Soup Contents.

The default Soupervisor slip does not contain a filing button. When

a user taps on a soup icon, the system checks if that icon has an ownerApp slot. If so, it assumes that the value of the slot is an application symbol and checks the base view of that application for a Soupervisor slot. The Soupervisor slot determines if the filing button shows up and what types of filing are available.

Note that applications with only one soup will get a soup icon in the Storage folder, but this icon will not have an ownerApp slot. If you want to support the Soupervisor you will have to create a soup icon whose `soupNames` array contains your soup name.

The slots you can specify in the Soupervisor frame are given in Table 4.

Slot	Req/Opt.	Description
`type`	Required	one of 'moveOnly, 'fileOnly or 'all
`FileSoup` your soup entries	Optional	hook for you to file and move all
`FileEntry`	Optional	hook to file an individual entry
`MoveEntry`	Optional	hook to move an individual entry

The `type` slot determines what types of filing can be done to your soups. `moveOnly` allows the user to move entries between stores. `fileOnly` allows bulk filing of all soup entries. all allows the user to both move and file entries.

The `FileSoup` function is called whenever the user chooses the File button in the slip in Figure 5. Note that one press of the button could result in both filing entries to a new folder and moving entries to a new store. If you define a `FileSoup` function, you are responsible for doing all the work of filing and moving. The syntax is:

```
FileSoup(newLabels, newStore)
```

The `newLabels` slot is either NIL, a symbol for the new folder or some invalid value. You must check this (see below). The `newStore` slot is either a new store or NIL. Remember that an individual entry could already be in the selected folder or on the selected store.

```
FileSoup: func(newLabels, newStore)
begin
   // should we file
   local filingP := newLabels = NIL OR IsSymbol(newLabels);

   if filingP OR newStore then
      // iterate through the stores
      foreach store in GetStores() do
      begin
```

NTJ

continued from page 1

Gotcha! Common Problems Converting From 1.x to 2.0

But before you start converting, review your old workaround code. At best the code will do something that the ROM will now do for you. At worst it will cause your application to throw. Some of the more common things now done in ROM are: finding which word or character was clicked on (`PointToCharOffset`, `PointToWord`) and capturing signatures (stroke bundles).

The rest of this article goes over areas of the system where you are likely to make mistakes. Each section deals with a specific section of the ROM. This is so you can refer back to it at two in the morning before you start tearing your hair out. Naturally you will read the entire article first.

Soups

Many things have been added to soups: tags for fast queries, multi-index queries, entry aliases and expanded soup notification are some of the ones you are likely to use. Of course, you have to form some new habits...

Empty Soups

The first error you will hit is trying to initialize your soup. See if you can figure out what is wrong with this first attempt at initialization code:

```
// register the soup
RegUnionSoup(kAppSymbol, kSoupDef);

local mySoup := GetUnionSoup(kSoupName) ;
for i := 1 to 10 do
begin
  local newEntry := {first: GetRandomWord(3,10)};
  mySoup:AddToDefaultStoreXMIT(newEntry, kAppSymbol) ;
end;
```

The chances are this code will throw. The error is using `GetUnionSoup` since the union can be empty. This is because soups are not created on a given store until something is added to them.

You can fix the code in two ways. The easy way is to use the result from `RegUnionSoup`. However, since you do not want to register your soup all over the place, you can also use `GetUnionSoupAlways`.

Along the same lines, you may think of using the `initHook` in a soup definition frame. It allows you to specify a closure that is called when the soup is created on a store. Note that it is called when the soup is created, not when you register the soup. So the `initHook` is not useful for seeding a soup with initial values.

Storage Folder

If you have played with the extras drawer, you will note there is a folder called Storage. This is a place where the user can go and work with all data from a given application. Remember that your soup can be deleted from this folder. Do not assume that your soup will always be

there.

Query Quandary

It used to be that `startKey`, `endTest` and `validTest` were the way to get efficient queries. That has changed. `beginKey`, `endKey` and `indexValidTest` have replaced them. Check your old queries, they probably use the less efficient or obsolete ways.

A good way to check is to do a Search in Newton ToolKit for `Query`. This can find some lurkers you may have forgotten. Remember that the new way to do things is to use `soup:Query`.

On the subject of `indexValidTest`, carefully read the documentation. Note that truncation of keys occurs. Make sure your valid test takes this truncation into account.

Xmit Files

Upon reading the documentation, you will note that most of the functions you knew have been replaced by an `Xmit` (transmit change) version. Although the old functions (e.g., `RemoveIndex`, `AddToDefaultStore`, etc.) still exist, you should use the new `Xmit` form. In most cases the change is only in the notification, but in some cases there is extra behavior that is required and that only exists in the `Xmit` form.

View System

There have been many changes in the view system. There are several places where undefined behaviors have been replaced with new undefined behaviors. In other cases, we have replaced them with defined behaviors.

Screen Size/Rotation

One of the really cool features in 2.0 is the ability to rotate the screen. From a programmer's perspective, you need decide if you want to support a rotated mode. This may require considerable redesign of your layouts.

If you do support rotation, you must provide a `ReorientToScreen` method in any view that is a child of the root. Most people remember to add the method to their base application view. Most also forget to add it to things like splash screens or slips that are created with `BuildContext`.

The key thing to remember is that `ReorientToScreen` is both a method that allows you to perform some action upon rotation, and a magic cookie that enables your application to open in Landscape mode. If your application is running fine in rotated mode and some action causes a notify that tells you the current thing can not operate in rotated mode, you can be pretty sure you have missed a `BuildContext` view somewhere.

Refresh/Update/Redraw

The 1.x OS had a mostly undefined behavior that tended to create very large update areas. The result was a lot of redrawing that was completely unnecessary.

The 2.0 OS is much smarter about what it will update. This means that clipping is much more important that it was in 1.x. If you have a child that draws outside of it's parent, and the parent does not clip, you will see artifacts and glitches on the screen. Things may not update the way you expect.

On a related note, animation may look very jerky on 2.0. This is because the interpreter is faster and the task scheduler has changed. Although your changes may be written to the screen buffer quickly, the buffer may not get written out to the LCD as often as before. The only way to force a screen update is to use `LockScreen` (`DrawShape` and `CopyBits` will not do this). Even if your animation works now, wrap the code:

```
// wrap animation code in LockScreen to ensure timely LCD refreshes
LockScreen(true) ;
// ... do animation stuff
LockScreen(nil);
```

viewJustify

In 1.x a couple of the mixed sibling/parent justification modes had problems. In 2.0 these problems have gone away. If you start seeing weird placement, check your `viewJustify` and `viewBounds`.

Also, when you design your views, remember there are new justification modes (ratio and anchored).

Protos

There are many more protos in 2.0 and several old standbys have been fixed. But some of the new protos can be tricky...

listPicker, peoplePicker

These are both very cool and very useful protos. The gotcha is that you will briefly read the manual then try and implement a `listPicker`. Then you will read the manual some more. These protos are not as easy to use as you first think. Make sure you check out the sample code and understand the concept of a `nameRef` before you start coding these beasties. If an unexpected behavior occurs, check whether the behavior you want is specified in the `listPicker` or in the `pickerDef`.

MyTitle Moved

`protoTitle` now comes with an underline and a different font. If your title was close to full width before, it will likely wrap. Also note that it is possible to specify an icon for a title. This icon should be a small version of your application icon. Checkout the Connection application for an example.

Input & Recognition

From a developer's perspective there are two major changes in this area: rich strings and better access to stroke information.

Rich Strings

There are lots of potential gotchas here. Most of them revolve around one central point. A rich string is not a string, even though it may look like one. Here is the first typical offense:

```
local input := Clone(myRichInputLine.text) ;
```

The code assumes the `text` slot is a simple string. It you do get text this way, you will end up with box characters in the string and you will loose the ink information. The correct code is:

```
local input := myRichInputLine:GetRichString() ;
```

Here is a table of common mistakes:

Nope	Yep

`Length`	`StrLen`
`BinaryMunger`	`StrMunger`
`str[i]`	can be `kInkChar ( $\uF700)`
`SubStr(str,i,0)`	can be an "empty" but really an ink character

Tab Order

Now that there is an external hardware keyboard available, tabbing between fields is possible. In general, the tab order is the same order as views are shown in the Newton ToolKit browser window. It traverses recursively down the hierarchy through any view that is based on a protoInputLine.

You should test to make sure that the tab order is what you expect. You can do this using the tab key on any of the soft keyboards. If the tab order is not what you want, either re-arrange the order in NTK, or see the Newton Programmer's Guide for how to define a custom tab order.

Click and Recognize

This is more of a reminder of a new feature than a gotcha. It used to be impossible to do recognition in a child when both the child and its parent defined a `viewClickScript`. Returning true from the `viewClickScript` terminated system processing of the current stroke. But returning NIL from the child moved processing to the parent. Now you can return the symbol `'skip` from the `viewClickScript` of the child. This will prevent the parent from processing the click, but will allow the system to continue processing the current stroke.

ROUTING

Routing has been completely changed. The new system is much more flexible than the old system. Your formats can take advantage of similar types of transport without a modification to some global routing frame. As an example, in the old system you had to define a mail routing frame explicitly. Now you define a text based format that can be taken advantage of by any mail system.

2.0 Route Enabling

The biggest gotcha in converting from 1.x to 2.0 routing is making sure that your 1.x global routing frame is gone. If any part of it is installed or remains, it will disable all Newton 2.0 routing for your application.

Classy Target

Make sure your target includes a class slot. This slot is used to find which routing formats to use from the view definitions registry. That determines what type of routing you can do (beaming, print, fax, etc.) So if you find your action menu empty, check your target for a class slot. You may need to define a `GetTargetInfo` method for greater flexibility.

Beam/Mail "Formats"

In the old system, beaming and mailing were specified in a different way from printing and faxing. Beaming would grab the value of your target slot and jam it in the item frame. Mail required specifying a symbol in your print format and a method in your application. In 2.0 that has changed, however: beam and mail are a bit different from page-based formats.

Just like printing or faxing, you need to register a view definition using `RegisterViewDef`, but beam and mail need to be based on `protoFrameFormat` instead of `protoPrintFormat`.

Faxing Folly

In the 1.x world, adding the ability to fax was easy. Getting the fax to go through was a little more difficult. The main problem is the fax standard, which will drop the line if too much time passes with no activity. In the 1.x world it was a case of hurry up or hang up.

In the 2.0 world, the OS provides a `FormatInitScript` for fax formats. Do all of your slow operations (soup queries, building cached shapes, etc.) in this script and the fax should proceed without a hang up.

Item Hacking

In transports that send an item to another Newton (for example, beaming), the item sent is more than just your data. The item is a wrapper that provides information to the OS about what the data is and where it belongs. In the past, most of the item on the sending end got copied to the receiving end. This included arbitrary slots added or changed by your application.

However, just because the item appears to survive, does not mean you can rely on everything surviving. The only slots you can rely on arriving at the other side are the `body` and `title`. Anything else is fair game for change or deletion.

COMMUNICATIONS

What can we say – we made it better. Endpoints are now much easier to use. But there are a few bits and pieces that can get you...

protoEndpoint

This beasty is gone. It is not defined in the 2.0 platform file. Of course you can still find it, but you should not use it. OK, you really do not want to use it because `protoBasicEndpoint` works much better and is much cleaner. `protoEndpoint` is there only for 1.x applications. If you are converting, you must use the new endpoints.

FlushOutput Gone

In the old world you followed every output with a `FlushOutput`. In the new world `FlushOutput` is gone. That means if you call it you will get an undefined method error. The 2.0 output calls tell you if the output succeeded by either terminating (synchronous) or calling the callback (asynchronous).

In addition, `FlushOutput` used to cause many problems when called from input specs. Now that it is gone, you can do output from input specs without having to worry about the "FlushOutput Problem." NTJ

Method Tuples

Check your argument counts. Most methods of the new endpoints take two arguments. In 99% of the cases the second argument will be NIL. The second argument is a frame that lets you further specify a call. Things like: timeouts, synchronous/asynchronous and a completion script. In some cases you can pass additional options.