



Newton[®] Technology

J O U R N A L

Volume II, Number 2

April 1996

Inside This Issue

Newton Directions C++ and NewtonScript	1
New Technology Apple Announces New MessagePad 130 with Newton 2.0!	1
Communications Technology Finite-State Machines: A Model for Newton Communications	3
Technology Update Package Deal	6
New Technology Suppressing and Freezing Packages (Newton's "Shift Key")	10
Communications Technology Move It!	12
Communications Technology Writing a Transport	17
Business Opportunities Seven Ways to Create the Killer Newton Application	22
Letter From the Editors Dear Newton Developer	23



Newton

Newton Directions

C++ and NewtonScript

Walter Smith and Rick Fleischman,
Apple Computer, Inc.

With the release of Newton C++ Tools later this year, Newton software developers will have the ability to incorporate C++ code in their Newton applications. However, the Newton software architecture is rather non-traditional, so the use of C++ will be somewhat different on the Newton platform than on other platforms. NewtonScript will continue to be the high-level application development language, while C++ can be used when necessary for optimized performance, or when you have existing C or C++ code you would like to reuse.

In this article, we explore the relationship between C++ and NewtonScript. We will discuss the languages themselves and the motivations behind their designs, as well as the practical aspects of combining them in the Newton environment.

LANGUAGE DIFFERENCES

The designers of NewtonScript and C++ had very different goals and priorities. Not surprisingly, the resulting languages are different.

C++

C++ was intended to be a "better C" – a language that could supersede the C language and add features for larger-scale programming, such as classes. Some of the critical design goals were:

- Compatibility with C
- Speed and space performance equivalent to C
- Integration with existing development systems

New Technology

Apple Announces New MessagePad 130 with Newton 2.0!

by Korey McCormack, Apple Computer, Inc.

Apple Computer, Inc. has announced the newest member of the MessagePad family of products, the MessagePad 130. The MessagePad 130, with the award-winning Newton 2.0 operating system, is the first Newton PDA for mobile professionals that offers user controllable backlighting for on-demand use, and a new non-glare screen for viewing and entering information under any lighting conditions. The MessagePad 130 also provides additional system memory for improved performance with Internet communications applications and multi-tasking.

THE POWER OF NEWTON 2.0

The MessagePad 130 with Newton 2.0 allows users to organize their work and personal life, seamlessly integrate information with personal computers, communicate using faxes, wireless paging and e-mail, and expand its capabilities with a wide range of third-party solutions. Improved handwriting recognition allows for easy data entry, recognizing printing, cursive or a combination of both that transforms the handwriting into typed text. It has a built-in notepad, to-do list, datebook, telephone log, and address file for organizing personal and business affairs, as well as Pocket Quicken (US only) to help organize personal and business expenses.

continued on page 21

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

J. Christopher Bell, Michael S. Engber, Rick Fleischman,

Guy Kawasaki, Korey McCormack, Jim Schram,

Walter Smith, Bruce Thompson

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

John Kawakami • *Editorial Assistant*

Matt Neuburg • *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.

Guest Editorial

An Address to Developers from Newton WW Sales and Marketing

Dear Newton Developer,

The Newton Systems Group has just concluded its most successful quarter to date. In addition to the Best-Of-Comdex honors for 2.0, an avalanche of positive press has followed. In validation of what you may already know, sales at 150% of expected volumes tell the world that Newton 2.0 is a hit.

Going forward, our collective challenge will be to build upon the increasing awareness and goodwill being generated by the launch of 2.0 to focus all our energy on solutions development. The key to our mutual success is to dominate in the applications space.

Customers tell us that they are impressed with the applications available for Newton devices, and are purchasing MessagePads for much more than their integration, communication, and organization capabilities. Third party applications often offer customers a complete mobile solution which parallels their work on the desktop, but lets them work away from the office with a truly mobile device.

At our recent SI/VAR developers conference over 400 developers (our other customer base) saw demonstration after demonstration of vertical, horizontal and peripheral solutions for the platform. It was obvious to everyone in attendance that critical mass is quickly approaching. In the closing remarks executives Jim Buckley (President of the Americas), Dave Nagel (Senior VP R&D) and Mike Markula

all commented that the future for Newton was indeed a bright, if not a critical part of Apple's strategy.

For our part, we will continue to innovate and make systems facilities and tools available so you can realize your best applications ideas in the Extras Drawer. We will also continue our platform marketing and advertising, and we will continue to pursue licensees to help proliferate the platform.

For your part, we hope you remain as committed to our shared vision as you have been and continue to show the world how great Newton is by putting forth your best work on our platform first.

Best Regards,

Mike Lundgren

Acting Director WW Sales and Market Development

Finite-State Machines: A Model for Newton Communications

by Jim Schram and Bruce Thompson, Apple Computer, Inc.

Communications between the Newton and other devices tends to be a complex task. Between managing the endpoint and handling the interactions with the user, there is a lot going on. Modeling the communications using a finite-state machine simplifies the task of designing a communications-based application.

WHAT IS A FINITE-STATE MACHINE?

A *finite-state machine* (FSM, or simply *state machine*) is a collection of states, events, actions, and transitions between states. Figure 1 shows an example of a simple FSM.

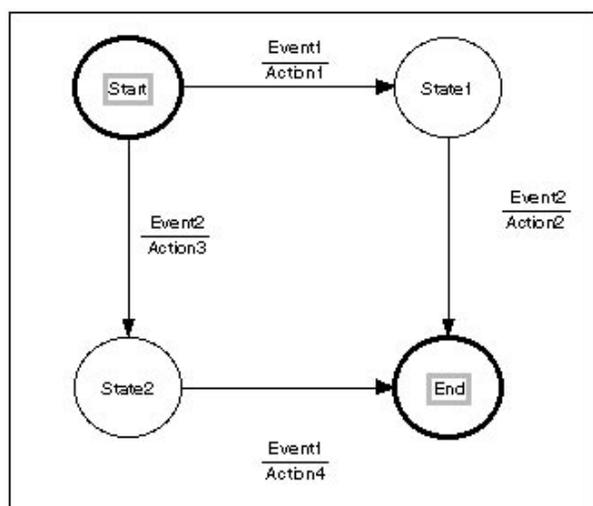


Figure 1: Simple Finite State Machine

This FSM has four states (Start, State1, State2, and End) and responds to two different events (Event1 and Event2). The actions occur when moving from one state to the next. For example, if the FSM is in the Start state and Event1 occurs, then Action1 will be performed as the FSM is moving to State1. It is often easier to draw a state-transition diagram (like Figure 1) than it is to describe in words (and often code) what the actions are.

Given a state-transition diagram, it is easy to create a state-transition table like the following:

Current State	Event	*Event?*
	Event1	Event2
Start	(Action1, State1)	(Action3, State2)
State1	—	(Action2, End)
State2	(Action4, End)	—
End	—	—

The state-transition table is the key to creating FSMs for Newton communications. With a state-transition table, NTK, and `protoFSM` (described below), building a finite-state machine is actually quite easy.

PROTOFSM

`protoFSM`, `protoState`, and `protoEvent` are a set of user prototypes that can be used to easily construct finite-state machines. The state machine, the states, and the events are laid out as if a view were being created. The state machine is the parent, with the states as children. Each state contains event children for each event that the state responds to. The event contains an action function and the symbol of the next state to transition to after the action has completed.

The state machine itself has a few additional slots. The `vars` slot is a frame that contains any additional variables that the actions may need to use. An endpoint is a good example of something to put into the `vars` frame, because many of the action procedures will need to access the endpoint in a communications-based state machine. There are also slots that reflect the current state, the current event, and the current action procedure.¹

Once a state machine is set up, using it is simply a matter of calling the `doEvent` method of the machine. `doEvent` takes two parameters: the first is the symbol of the event, and the second is an array of additional parameters for the action procedure. The action procedure is invoked in the context of the state machine, with the current state, event, and additional parameters passed in. After the action procedure returns, the state will be changed according to the `nextState` defined for the event.

WHY USE A STATE MACHINE?

Many Newton applications that perform communications have two main “tasks” operating essentially in parallel.² The main “tasks” of a communications application are user-interface management (in general, the primary operation of the application) and communications management. An example of this separation is the Llama-Talk sample from Newton DTS. This application has user-interface elements to send various kinds of objects over an ADSP connection. The user-interface elements (buttons) queue up requests to send the objects, and an idle-script actually performs the communications.

A state machine runs in a similar fashion. The user-interface elements will typically post events to the state machine based on what the user has requested. This includes operations like initiating a connection, disconnecting, sending items, and so on. The response to the event (the action procedure) will perform the actual endpoint calls asynchronously, with the completion scripts also posting events to indicate the success or failure of the action.

AN EXAMPLE OF A STATE MACHINE

To help illustrate all this, let’s look at a state machine for doing simple

endpoint setup and tear-down. The endpoint will establish a serial connection, send and receive simple items, and do a disconnect in response to a user action (or whenever an error occurs). First, we need to draw a state diagram to show the behavior of the endpoint state machine (see Figure 2).

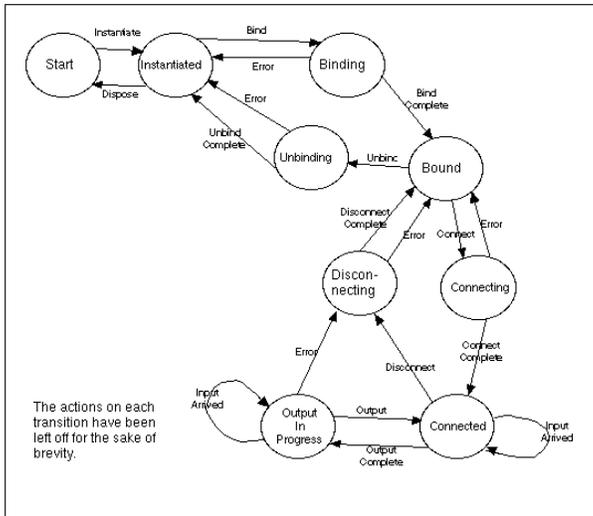


Figure 2: State Diagram

The next step is to construct a state-transition table from this diagram. For the sake of readability, the table is being presented more as an outline. This helps match the form the state machine will take in NTK.

- Start State:
 - Instantiate Event:
 - NextState: Instantiated
 - Action: Create an endpoint frame in vars, call `ep:Instantiate()`
- Instantiated State:
 - Bind Event:
 - NextState: Binding
 - Action: call `ep:Bind` asynchronously; the `completionScript` will post either `Error` or `BindComplete`
 - Dispose Event:
 - NextState: Start
 - Action: call `ep:Dispose()` and throw away the endpoint frame
- Binding State:
 - BindComplete Event:
 - NextState: Bound
 - Action: none
 - Error Event:
 - NextState: Instantiated
 - Action: Post a `Notify` that an error occurred. Could also post a `Dispose` event from here!
- Bound State:
 - Connect Event:
 - NextState: Connecting
 - Action: call `ep:Connect` asynchronously; the `completionScript` will post either `Error` or `ConnectComplete`
 - Unbind Event:
 - NextState: Unbinding
- Connecting State:
 - ConnectComplete Event:
 - NextState: Connected
 - Action: Setup the `inputSpec`. The `inputScript` will post an `InputArrived` event, the `completionScript` will post an `InputError` event if an error occurs
 - Error Event:
 - NextState: Bound
 - Action: Post a `Notify` that an error occurred. Could also post an `Unbind` event from here!
- Connected State:
 - InputArrived Event:
 - NextState: Connected
 - Action: Handle the input somehow
 - Output Event:
 - NextState: OutputInProgress
 - Action: Call `ep:Output` asynchronously; the `completionScript` will post either an `Error` or `OutputComplete`
 - Disconnect Event:
 - NextState: Disconnecting
 - Action: Call `ep:Disconnect` with the `cancel` option selected. The `completionScript` will post either `Error` or `DisconnectComplete`
- Disconnecting State:
 - DisconnectComplete Event:
 - NextState: Bound
 - Action: none
 - Error Event:
 - NextState: Bound
 - Action: Post a `Notify` that an error occurred. Could also post an `Unbind` event from here!
- Unbinding State:
 - UnbindComplete Event:
 - NextState: Instantiated
 - Action: none
 - Error Event:
 - NextState: Instantiated
 - Action: Post a `Notify` that an error occurred. Could also post a `Dispose` event from here!

Using this table, it's easy to lay out the elements of the state machine in

NTK. The final concept worth noting is that you can pass parameters to the action procedure along with the event. For example, the Output event could take an additional parameter: the item to be output. The error event could take an additional parameter: an exception frame, etc. In general, it's best to make the actions as simple as reasonably possible, and to try to capture as much of the behavior in the states as possible.

SUMMARY

Finite-state machines are a simple and elegant way to model the behavior of applications. By referring to a state diagram, it's easy to see if all contingencies have been handled. `protoFSM` provides a nice clean way to specify state machines for Newton applications. Communications protocols are often specified in terms of a state machine. Being able to easily transcribe a protocol definition into a finite-state machine will help cut down on development time and reduce the complexity of the resulting communications code.

Communications on the Newton platform is complex enough; using state machines helps reduce that complexity to a more manageable level. Because state machines lend themselves better to performing communications asynchronously (synchronous communications calls

involve a fair amount of overhead), applications will gain a performance advantage.

Communications on the Newton platform is complex enough; using state machines helps reduce that complexity to a more manageable level. Applications using asynchronous communications are generally more efficient than those using synchronous communications because the latter calls involve a fair amount of overhead. However, asynchronous communication applications can be more difficult to program. But, since finite state machines simplify the programming of asynchronous communications by reducing complexity. Thus, the end result can be applications that have improved performance.

¹ `protoFSM` is a Newton DTS sample that should be available by press time. You can find `protoFSM` and the accompanying sample code on AppleLink and at the Newton WWW Site (<http://dev.info.apple.com/newton/newtondev.html>) The next Newton Developer CD will also contain this sample.

² The word "task" should not be confused with the idea of any sort of multitasking. Although the Newton 2.0 OS is a multitasking (or, more properly, *multiprogramming*) operating system, this is not available to NewtonScript-based applications. 



If you have an idea
for an article
you'd like to write
for Newton Technology Journal,
send it via Internet to:
NEWTONDEV@applelink.apple.com
or AppleLink: NEWTONDEV



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: NEWTONDEV
or Internet: NEWTONDEV@applelink.apple.com

Package Deal

by Michael S. Engber, Apple Computer, Inc.

This article discusses some of the changes to packages in Newton OS 2.0. Several related topics, such as the details of the package format, Unit Import/Export, and the new Extras Drawer hooks, are mentioned peripherally to point out their relationship to packages and parts. In-depth coverage of those topics is beyond the scope of this article.

To benefit from this article, you should have some experience using the Newton Toolkit (NTK) to write Newton applications for Newton OS 2.0, and be familiar with the basic concepts of packages and parts. Familiarity with the “Bit Parts” article (see “References and Suggested Reading,” at the end of this article) would also be helpful.

Note that some of the details of creating parts with NTK are specific to NTK version 1.6. Earlier versions of NTK may not support parts, and future versions of NTK may support them in a different, more convenient, way.

BASICS

Most people think they understand the concepts of packages and parts, but I’m continually hearing nonsensical terms like “auto package,” or “my package’s RemoveScript.” People often confuse packages and parts. Sometimes it’s just a slip of the tongue. Sometimes it’s indicative of deeper misunderstandings.

A package consists of a collection of zero or more parts. (Yes, it’s possible – but pointless – to have a package containing no parts.) A modest example is shown in Figure 1 (in hex).

```
7061636B616765306E6F6E6500000000
00000001000000020000000200000036
00000000000000000000000000000036
000000000058
```

Figure 1: The World’s Smallest Package (54 bytes)

Most packages are somewhat larger than 54 bytes, and consist of one or more parts. Table 1 lists the most common kinds of parts.

Type	Frame part?	Description
form	Yes	Applications
book	Yes	Books
auto	Yes	Tricky hacks
font	Yes	Additional fonts
dict	Yes	Custom dictionaries

soup	No	Store w/ read-only soups
------	----	--------------------------

Table 1: Part Types

Notice that the second column of Table 1 indicates if a type of part is a frame part. Frame parts are basically just a single NewtonScript object – a frame. Conventions for what goes in the frame vary, depending on the type of part. For example, form parts keep their base view’s template in a slot named `theForm`.

You can access the actual part frames (of a package’s frame parts) via the `parts` slot of the frame returned by `GetPkgRefInfo`. There is bound to be some minor confusion in this regard. Although the argument passed to a form part’s `InstallScript` is commonly called `partFrame`, it is not actually a part frame. In version 1.x it was a clone of the part frame, and in 2.0 it is a frame that `protos` to the part frame. This detail may change. You should simply rely on the fact that `partFrame.slot` returns the specified slot from the part frame.

Most types of parts are frame parts (store parts being the most notable exception). There are other types of non-frame parts that can be created (e.g., communications drivers), but tools for doing so are not available yet.

PACKAGE OBJECTS

In Newton OS 1.x there was no way to directly access packages. The call `GetPackages` returned an array of information about the active packages, but there was no object returned that was itself a package.

In Newton OS 2.0, packages are stored as virtual binary objects (VBOs). A reference to a package VBO is called a package reference (abbreviated `pkgRef` in many of the related function names). In most ways, package VBOs are the same as regular VBOs created with `store:NewVBO`. For example, `GetVBOStore` can be used to determine the store on which a package resides.

The downside of representing packages this way is that, like ordinary VBOs, package VBOs are kept in a soup, providing a tempting target for hacking. The details of this soup are private and subject to change. APIs are provided for the supported operations. Anything you do outside of the supported APIs is likely to interfere with the current system (in some subtle way), not to mention breaking in future systems.

PACKAGE REFERENCES VS. ORDINARY VBOs

For most purposes, you can think of a package VBO as simply containing the binary data from the package file you created with NTK. There are a few place-holder fields (like the time of loading) that are filled in as the package is loaded, but apart from those, they are byte-wise identical – or so it seems.

If you create a VBO, set its class to ‘package’, and then use `BinaryMunger` to copy in the bytes from a “real” package, you’ll wind up with something that looks like a package but is still not the real thing. Try

passing it to `IsPackage`.

The differences are subtle. One obvious difference is that the real package is read-only. Another, more fundamental, difference is that the fake package is missing the relocation information that is associated with a package during the normal package-loading process. This relocation information is not part of the binary data and cannot be accessed via `ExtractByte`.

The only way to get a real package from the fake one is to pass it to `store:SuckPackageFromBinary`, which will create a real package from the data. There is no way to convert it in place.

THE EXTRAS DRAWER

In 1.x, the Extras Drawer was used solely to launch applications. The icons in the Extras Drawer corresponded to form parts and book parts. Package removal was handled by the “Remove Software” button, whose pop-up menu listed packages, not parts.

In Newton OS 2.0, the “Remove Software” button is gone. The Extras Drawer is one-stop shopping for all your package needs. Therefore, the model underlying the Extras Drawer icons is different. For example, a package that contains no form parts still needs an icon to represent it so the user will have something to delete.

The two obvious solutions are to have one icon per package or one icon per part. Neither of these solutions turns out to be satisfactory. Consider the following examples:

- Some multi-part packages require two icons (e.g., form + book part)
- Some multi-part packages require only one icon (e.g., form + auto part)
- Non-frame parts have nowhere to provide title and icon data (i.e., no part-frame)

The approach adopted by the Extras Drawer is a hybrid of the two, sort of a Miranda rule for packages: “You have the right to an icon. If you cannot afford an icon, one will be appointed for you.”

Any frame part in a package can have an Extras Drawer icon by providing a text slot (and optionally, an icon slot) in its part frame. If no frame part in the package provides an icon, then a generic icon labeled with the package name is created.

This approach preserves the 1.x representation of existing packages, as well as providing flexibility for more complex packages created in the future. The majority of existing packages, which consist of a single form part, get a single icon, just like they always have. Multi-part packages are also handled correctly. The form + book part example gets two icons, and the form + auto part example gets only one icon.

Unfortunately, existing packages that contain no form or book parts get the “court-appointed” representation: a single generic icon (see Figure 2) labeled with their package name. Without more information, there is no way to do much better. Use NTK 1.6 to rebuild these packages and provide a more aesthetically-pleasing title and icon.



Figure 2: Generic Icon

For form parts, the title and icon slots are created from the settings in the Project Settings dialog box (under Output Settings). For other frame parts, you’ll have to create these part-frame slots manually (this is something that could change in future versions of NTK). For example, in one of the project’s

text files you would define the title and icon slots with code like the following:

```
SetPartFrameSlot('title, "foo");
SetPartFrameSlot('icon, GetPICTAsBits("foo picture", true));
```

You shouldn’t feel that you need to provide an icon for every part in your package “just because you can.” For most packages, a single icon for the main form part will suffice. Extra icons serve no purpose, and will only confuse the user.

If your package doesn’t contain any frame parts (e.g., it contains just a soup part) you can avoid getting the generic icon by adding in a dummy frame part that specifies the title and icon. For example, you might add a form part that displays information about the package.

ACTIVE VS. INACTIVE PACKAGES

Executing a package’s `InstallScript` (and, in the case of form parts, creating the base view) is part of the process of activating a package. Executing the `RemoveScript` is part of the process of deactivating a package. Prior to Newton OS 2.0, packages were activated when they were installed and deactivated when they were removed. There was no concept of packages existing on a store and being inactive — unless you used a third-party utility. (By inactive package, I mean that the `InstallScript` has not yet run or, if the package was previously active, the `RemoveScript` ran during deactivation.)

In Newton OS 2.0, it’s possible to have packages visible in the Extras Drawer that are not active. They are identified by either a special snowflake icon, if the user purposely “froze” the package, or by an X’d-out icon, if they are inactive due to an error or because the user suppressed package activation. (See the article “Suppressing and Freezing Packages,” in this issue, for further details.)

For the most part, applications need not concern themselves with these details. Any package that correctly handles being installed and removed (by either deletion or card-yanking) should work correctly with user-controlled activation and deactivation.

INVALID REFERENCES

Invalid references are references to NewtonScript objects residing in a package that is either inactive or unavailable (i.e., on a card that is in the process of being removed). This topic is discussed in the article “Newton Still Needs the Card You Removed” (see “References and Suggested Reading,” at the end of this article).

A reference to an object in a package goes bad after the package is deactivated. In 1.x, attempting to access such an object resulted in a -48214 error. In Newton OS 2.0, after a package is deactivated, existing references to it are replaced with a special value. This means that instead of getting a cryptic -48214 error (which also occurs when loading a bad package), you get a more specific error message in the NTK Inspector:

```
!!! Exception: Pointer to deactivated package
or
<bad pkg ref>
or
bus error with value 77
```

A related problem occurs for packages on a card that is in the process of being removed. Obviously, the objects in the package are unavailable, and attempting to access them results in the dreaded card-reinsertion dialog.

You'll be relieved to hear that the card-reinsertion dialog has not gone away in Newton OS 2.0. However, it has been somewhat improved. It now displays the name of the package that used the invalid reference. There are other, less common, causes for the card-reinsertion dialog appearing (e.g., attempting to access the store) which do not allow for the package to be easily determined. In these cases, the package name is not displayed.

Knowing the name of the offending package is a great help to users trying to figure out why they can't remove a card, but this does little to help a developer figure out why his package is plagued by the "grip of death." There are plans to provide a tool to allow developers to find out what invalid object was accessed.

There is also a new function available for dealing with these problems. Previously, it was impossible to determine if a reference was valid before attempting to access it. In Newton OS 2.0 you can use the function `IsValid` to make this determination.

NEW PART-FRAME SLOTS

Newton OS 2.0 offers you some new part-frame slots. Some of the slots are data, and some are methods (i.e., they contain functions). You set their values using the `SetPartFrameSlot` function in NTK, as was previously described.

New Part-Frame Data Slots

`app`
`icon`
`title`

In 1.x, the `app`, `icon`, and `title` slots were used to specify a form part's `appSymbol` and Extras Drawer icon and its Extras Drawer title, respectively. In 2.0, they can be used for any frame part. For non-form-parts, the `appSymbol` is used to identify the part when using `SetExtrasInfo`.

`labels`

This slot is used to pre-file an Extras Drawer icon. For example, use `'_SetUp` to specify that the icon initially goes in the `SetUp` folder. See the article "Extra, Extra" (see "References and Suggested Reading," at the end of this article) for a more in-depth example.

New (Optional) Part-Frame Methods

`DoNotInstall`

The `DoNotInstall` message is sent (with no arguments) before the package is installed on the unit. It gives a package a chance to prevent itself from being installed. The message is sent to every frame-part in a package. If any of them return a non-nil value, the package is not installed. You should provide the user with some sort of feedback, rather than silently failing to install. For example, a package wanting to ensure it was only installed on the internal store could have a `DoNotInstall` like the following:

```
func()
begin
  if GetStores()[0] <> GetVBOStore(ObjectPkgRef('foo')) then
    begin
      GetRoot():Notify(kNotifyAlert, kAppName, "This package was
not installed.
      It can only be installed onto the internal store.");
      true;
    end;
  end;
end
```

`DeletionScript`

The `DeletionScript` message is sent (with no arguments) just prior to the

package being deleted. This script allows you to distinguish the user scrubbing a package from the user yanking a package's card. A `DeletionScript` typically does "clean up" like deleting soups, deleting local folders, and eliminating preferences from the system soup.

`RemovalApproval`

`ImportDisabled`

These scripts are used to inform a package that some of the units it's importing are being removed. `RemovalApproval` give the package a chance, prior to removal, to warn the user of the consequences of removing the unit. `ImportDisabled` is sent if the unit is subsequently removed. See the "MooUnit" DTS sample code for further details on units.

NEW PACKAGE-RELATED FUNCTIONS

`GetPackageNames (store)`

return value – array of package names (strings)
store – store on which the package resides

`GetPackageNames` returns the names of all the packages on the specified store. Note that it returns the names of all the packages, even those that are not active (e.g., frozen packages).

`GetPkgRef (packageName, store)`

return value – package reference
packageName – name of the package
store – store on which the package resides

`GetPkgRef` returns a reference to a package given the package's name and the store on which it resides.

`ObjectPkgRef (object)`

return value – package reference (nil if object is not in a package)
object – any NewtonScript object

Determines which package an object is in and returns the package reference. Returns nil for immediates and other objects in the NewtonScript heap, including soup entries.

A package can get its own package reference by calling `ObjectPkgRef` with any non-immediate literal. For example, `ObjectPkgRef ('read`, `ObjectPkgRef ("my")`, or `ObjectPkgRef ([1,i,p,s])`. Note that the `InstallScript` of a form part is cloned (by `EnsureInternal`), and the clone is executed. This means that the above examples wouldn't work because the entire code block – including the argument to `ObjectPkgRef` – resides in the NewtonScript heap. A workaround is to get an object from the package via the argument passed to the `InstallScript` – e.g., `ObjectPkgRef (partFrame.theForm)`. Other types of parts do not have this problem.

`GetPkgRefInfo (pkgRef)`

return value – info frame (see below)
pkgRef – package reference

This function returns a frame of information about the specified package, as shown below.

```
{
  size:          <int>          // uncompressed package size in bytes,
  store:         <frame>        // store on which package resides
  title:         <string>,      // package name
  version:       <int>,         // version number
```

```

timestamp:    <int>,           // date package was loaded
creationDate: <int>,           // date package was created
dispatchOnly: <nil or non-nil>, // is package dispatch-only?
copyprotection: <nil or non-nil>, // is package copyprotected?

copyright:    <string>,       // copyright string
compressed:   <nil or non-nil>, // is package compressed?
cmprsdSz:     <int>,         // compressed size of package in bytes

numparts:     <int>,         // #parts in the packages
parts:        <part data array>, // part-frames for the frame parts
partTypes:    <array of symbols>, // part types corresponding to data in parts slot
} // other slots are private and undocumented

```

IsPackageActive(pkgRef)
 return value – nil or non-nil
 pkgRef – package reference
 IsPackageActive determines if the specified package is active or not.

IsPackage(object)
 return value – nil or non-nil
 object – any NewtonScript object
 IsPackage determines if the specified object is a package reference.

IsValid(object)
 return value – nil or non-nil
 object – any NewtonScript object
 IsValid detects if the specified object is (was) in a package that is no longer active. If the package is on a card in the process of being removed, it will return nil and will not cause the card-reinsertion dialog. IsValid returns true for immediates or objects that do not reside in a package (e.g., in the NS heap or in ROM).

Note that IsValid does not deeply check the object.

MarkPackageBusy(pkgRef, appName, reason)
 return value – unspecified
 pkgRef – package reference
 appName – string describing the entity requiring the package
 reason – string describing why the package should not be deactivated

MarkPackageBusy marks the specified package as busy. This means the user will be warned and given a chance to abort operations that deactivate the package (e.g., removing or moving it). The appName and reason are used to generate the message shown to the user.

You should mark a package busy if its deactivation will cause you problems. For example, a store part might be providing critical data. Since the user may still proceed with the operation, you should attempt to handle this eventuality as gracefully as possible.

Be sure to release the package as soon as possible so as not to inconvenience the user.

Note that you do not need to use MarkPackageBusy on a package because you're importing units from it. Units have their own mechanism for dealing with this problem (RemovalApproval *et al.*).

MarkPackageNotBusy(pkgRef)
 return value – unspecified
 pkgRef – package reference
 MarkPackageNotBusy marks the specified package as no longer

busy.

SafeRemovePackage(pkgRef)
 return value – unspecified
 pkgRef – package reference
 SafeRemovePackage removes the specified package. If the package is busy, the user is given a chance to abort the operation.

SafeMovePackage(pkgRef, destStore)
 return value – unspecified
 pkgRef – package reference
 destStore – store to which to move the package
 SafeMovePackage moves a the specified package to the specified store. If the package is busy, the user is given a chance to abort the operation; moving a package requires deactivating it, moving it, and then reactivating it.

SafeFreezePackage(pkgRef)
 return value – unspecified
 pkgRef – package reference
 SafeFreezePackage freezes the specified package. If the package is busy, the user is given a chance to abort the operation.

ThawPackage(pkgRef)
 return value – unspecified
 pkgRef – package reference
 ThawPackage un-freezes the specified package.

REFERENCES AND SUGGESTED READING

Engber, Michael S., "Bit Parts." *PIE Developers*, May 1994, pp. 27 – 29.

This article discusses packages in Newton OS 1.x and creating them with older versions of NTK. Most of the information is still relevant. It is available from the various PIE DTS CDs and archives as well as for ftp from <ftp.apple.com/pub/engber/newt/articles/BitParts.rtf>

Engber, Michael S., "MooUnit." PIE DTS Sample Code, Fall 1995.

This sample code provides documentation on unit import/export and a simple example. It is available from the various PIE DTS CDs and archives.

Engber, Michael S., "Newton Still Needs the Card You Removed." *Dr NTJ Tap*, May 1994, pp. 12 – 18.

This article provides an in-depth discussion of invalid references (to objects in inactive packages). It is available from the various PIE DTS CDs and archives as well as for ftp from <ftp.apple.com/pub/engber/newt/articles/NewtonStillNeedsTheCard.rtf>

Goodman, Jerry, "Psychic Archaeology." 1980, Berkley Books.

This book discusses techniques for researching artifacts of mysterious origin about which very little factual information is known.

Sharp, Maurice, "Extra Extra." *Newton Technology Journal*, February 1996.

Suppressing and Freezing Packages (Newton's "Shift Key")

by Michael S. Engber, Apple Computer, Inc.

This article discusses the concept of package activation from the practical perspective of how it can be suppressed to help you deal with incompatible applications (even those that cause such severe system problems that you can't delete them). It also discusses the related concept of *package freezing*, a latent feature of Newton OS 2.0.

SUPPRESSING PACKAGE ACTIVATION

In Newton OS 1.x, if you had a package on the internal store that caused trouble at startup, your only option was to completely erase the internal store (reset with the power switch down). Similarly, if you had a package on a card that interfered with mounting the card, you had to force the card to be erased (insert the card with Prefs open).

In Newton OS 2.0, it's possible to suppress package activation. This allows the unit to start up or a card to be inserted without running any of the packages' code. This gives you a chance to delete the problem package instead of having to completely erase the store it's on. (Macintosh users will find this reminiscent of booting with the Shift key down to keep extensions from loading.)

To suppress package activation on the internal store, first reset the unit, then turn the unit over and hold the pen down in the left 1/4 inch of the screen. Keep the pen down until you see a message asking if you want to activate the packages on the internal store (see Figure 1). Select "No."

(Note: This procedure won't work if you place the pen too far to the left, so it rests against the raised edge of the plastic case; there's a small dead area at the border of the screen and the case. The message shown in Figure 1 will come up before the splash screen disappears. Therefore, if the splash screen disappears you need to try again, holding the pen a little further in from the edge.)



Figure 1: Package Activation Dialog Box

When you open the Extras Drawer, you'll see X's over some of the icons (see Figure 2). This indicates that those packages are not active. This procedure does not affect the packages that are built into ROM; only the packages you've loaded or that were preloaded onto the unit at the factory

can be disabled in this way.

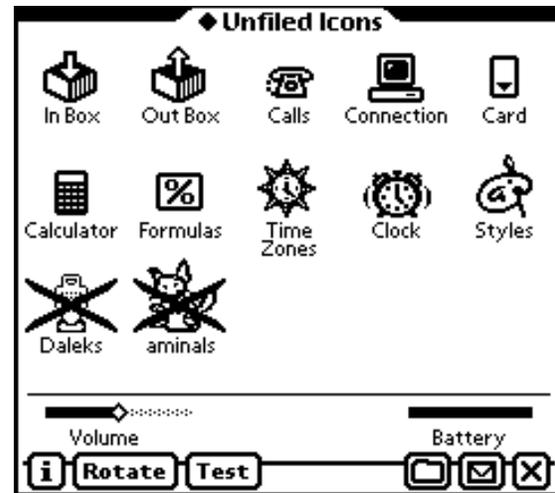


Figure 2: X'd-Out Icons in the Extras Drawer

Tapping an X'd-out icon causes its package to be activated. Activate them one at a time until you identify the culprit. For example, if the symptom is that the Names application won't open, tap an X'd-out icon and, once the X disappears, see if you can open Names.

If at this point you can delete the guilty icon, great. If the problem is nasty enough to mess up the Extras Drawer, then you'll have to reset again, suppressing package activation. This time you should simply delete the icon, without first activating it.

When searching for offending apps, remember to look in the Extensions Folder (or show "All Icons"). In 1.x systems there were some packages that didn't have a corresponding icon in the Extras Drawer. In 2.0, every package has at least one icon.

The same procedure can be applied to storage cards. First, insert the card. After you lock it, hold the pen down near the left edge of the screen, as described above. You'll be asked if you want to activate the packages on the card.

A MORE ADVANCED TRICK

If you reset the unit and hold down the pen near the top edge of the screen (instead of the left edge), in addition to suppressing package activation, the unit's orientation and backdrop application will be reset to their defaults: portrait orientation and Notepad as backdrop. This is useful if you have an application that seems to be working fine until you make it the backdrop.

The top edge has these additional effects only when you reset the unit. When inserting storage cards, you can use the top or left edge to suppress package activation.

OTHER REASONS ICONS ARE X'D OUT

An X'd-out icon indicates that a package is inactive. Even if you don't suppress package loading, there are other circumstance when you might run into inactive packages. The most common instance occurs when you try to load two copies of the same package. For example, if package "foo" is on the internal store and a card containing "foo" is inserted, the icon for "foo" on the card will be X'd out.

There might be other circumstances where you briefly see an X. For example, moving a package between stores causes it to be deactivated, moved, then reactivated. We try not to show the X in this situation because the package is only temporarily inactive, but in some circumstances the X is briefly visible.

If you see an inactive icon, feel free to go ahead and tap it. The system will attempt to activate and launch it. If there's a problem, you'll get an error message. With any luck, it will be an informative one, like "The package 'foo' (on store *my card*) was not activated because a package by the same name (on store *internal*) is already in use."

PACKAGE FREEZING

If you don't know what frozen packages are, read on (and no, they're not

something you buy at the supermarket). A number of third-party developers are providing utilities that turn on this latent feature of the Extras Drawer.

Frozen packages are inactive packages. The difference between frozen packages and the suppressed packages described earlier is that frozen packages are purposely made inactive by the user, and stay inactive until the user reactivates them. Suppressing package activation deactivates packages only temporarily. If you reset the unit (or reinsert the card), the icons that were X'd out won't be X'd out any more. Frozen packages, on the other hand, are deactivated at startup, and remain so until the user specifies otherwise.

Activating packages uses up some of the system's working memory and takes time. Cards with a large number of packages can take a long time to mount and can cause you to run low on memory. A way to work around this is to selectively "freeze" (deactivate) the packages you don't use very often. You freeze a package or group of packages by selecting them in the Extras Drawer and choosing Freeze from the Action button. The package's icon will turn into a snowflake (see Figure 3) indicating that the package is frozen. You thaw (activate) a frozen package by simply tapping its icon.



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: NEWTONDEV
or Internet: NEWTONDEV@applelink.apple.com

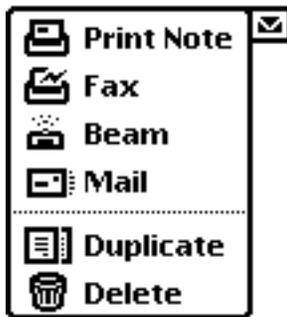


To send comments or to make requests for articles in Newton Technology Journal,
send mail via the Internet to: NEWTONDEV@applelink.apple.com

Move It!

by J. Christopher Bell, Apple Computer, Inc.

When users first access applications on a mobile Newton device, their primary concern is to enter information (or, in some cases, to retrieve information). Only after they enter information do they move it outside their Newton, whether to a printer or to a friend's email account. Moving the information outside the Newton – or deleting or duplicating it – is done by tapping the Action button (the envelope icon), which is ubiquitous in the Newton user interface. Tapping the Action button reveals a list of ways to move the currently viewed information.



The ability to move information using the Action button is called *routing*.

The items above the line in the Action list are services that move information outside the Newton device. These communications services are called *transports*. The Newton OS includes built-in transports, but developers can add choices to Action lists by creating new transports that users can install as packages.

The items below the line are application-specific actions, which often include Duplicate and Delete. Every application can add these actions, called *route scripts*, to its own Action lists. Applications also have control over which transports appear in the top part of the list.

Applications do not enable transports by listing by name which transports they can support. If that were the case, application developers would need to upgrade their products with a new transport list whenever other developers created new transports. Instead, applications specify, in general terms, what types of transports would be appropriate based on the characteristics of the information they manipulate. A *dataType* is the generic classification used to describe the way in which information will be sent outside the Newton device. Common dataTypes are plain text (the `'text'` dataType used by Mail), NewtonScript frames (the `'frame'` dataType used by Beam), and imaging layouts (the `'view'` dataType used by Print and Fax). You can also define your own dataTypes, although applications would have to know about them in order to take advantage of them.

Using dataTypes to specify categories of transports offers many benefits. The greatest benefit is that third-party transports can be just as important as built-in transports for Newton platform solutions. This is because transports

that support at least one standard dataType automatically work with most applications. If appropriate, in-house or vertical developers can also write custom transports. Applications and transports do not need to know anything about each other's details in order to work together.

Before the user chooses among transports and route scripts to move information, the Action button must present the Action list. Determining which actions should appear in your application occurs when the user taps the Action button, not when an application is installed. This interface offers some interesting advantages. For instance, transports can register and unregister dynamically, yet maintain a consistent user interface. Because of the dynamic nature of this process, you have flexibility in how and where you implement your routing code. You must help the system build the Action list by supplying certain methods and slots, as well as using some global registries. Writing code to use the Action button and communicate with the In/Out Box is what it meant by implementing routing.

BEFORE WRITING ROUTING CODE

Before coding your application to support routing, you should think about the following issues. The most important question to answer before implementing routing is: "How many different types of data are in my application?" You must identify each type of data with a unique symbol that contains your developer signature. For instance, if your application had two main views, where one manipulated llama information and one manipulated llama rancher information, you might represent the data using two types of data. The types of data – called *data classes* – might have the symbols `'|llama:jX|` and `'|rancher:jX|`.

To determine how many data classes you have, it might help to think about which types of transports can route your data. For instance, in our example, perhaps the rancher information could be exported to a text-only email system, but the llama information might consist of only a picture; therefore, sending to a text-only transport would make no sense. This difference indicates that our information consists of multiple data classes, and we must figure out what they represent.

When users print information from your application, would some printing layouts (called *print formats*) be accessible only from certain parts of your application? If so, the subviews of your application might represent different data classes, and you must figure out what they represent.

Before you write your routing code, you should think about the things that your application or its subviews must handle when the user taps the Action button. The answers to these questions may affect your application design if you determine that you have more data classes than originally expected. For instance, do you know how to get references to the information the user is viewing or has selected? Also, should some route scripts appear in only some circumstances? For instance, in our sample application, the "Feed" route script might appear when the user selects an

item in the llama viewer, but not in the rancher viewer.

Answering these questions and looking at the rest of your application design will make most remaining design decisions more straightforward.

ROUTING FORMATS

In Newton 2.0, print formats and pre-routing initialization are encapsulated into objects called *routing formats*. These objects format the data for use *outside* your application. This section discusses the various types of routing formats and how to create them.

For printing and faxing, you will base your routing format on `protoPrintFormat`. The bulk of your work will be designing your views so they look nice, handling view justifications properly, and using as little NewtonScript memory as possible. Your print format code must create views to display the contents of the `target` variable. Note: Do not write to `target` or access `fields.body`, since the behavior of doing either is undefined.

Your format's `PrintNextPageScript` method must return a non-nil value while there is more data to route. You can design your format's `PrintNextPageScript` to call the `RedoChildren` view method and let its view methods recreate child views with new information. Alternatively, you can update the view contents directly, for instance using code like `SetValue(myParagraph, 'text, newText)` to change the contents of text views.

Some fax protocols will time out after a few seconds of inactivity. If you must perform time-consuming calculations or prepare complex drawing shapes in your print layouts, do it your format's `FormatInitScript` method. This method is guaranteed to be called before connecting to a fax machine.

To enable transports supporting the `'frame` or `'text` dataTypes (Beam or Mail, for example), you will base your routing format on `protoFrameFormat`. Your application may not need to perform special pre-routing formatting if simple NewtonScript frames are routed, but registering formats in this way is required to inform the system that transports like Beam or Mail can move your frame or text information.

By default, `protoFrameFormat` handles both `frame` and `text` dataTypes. If you need text-only or frame-only formats, you can override your format's `dataTypes` slot. For instance, if your routing format supported sending only NewtonScript frames, it would look like this:

```
LlamaFrameFormat := {
  _proto: protoFrameFormat,
  symbol: kFormatSym,
  title: kFormatTitle,

  // override if you don't want both 'frame & 'text
  dataTypes: ['frame']
}
```

FROM FORMATS TO TRANSPORTS

A common question is: "How does the system decide which transports to show in the Action list?" The short answer is that the Action button looks for routing formats that can route the data, then looks for transports that can handle at least one of those formats. Here is a more detailed description of what occurs both before and after the user taps the Action button:

1. In NTK, you create routing formats. Slots in your base view will reference those formats.

2. Your application registers your formats with `RegisterViewDef(format, dataClassSym)` in its `installScript`. Access formats with code like `partFrame.theForm.myBaseViewSlot`

(Note: Don't use the `GetLayout` NTK function for this. See the DTS Routing samples for more information.)

3. The user opens your application and selects and views some data, called the *target*.

4. The user taps the Action button.

5. The Action button determines **the data** to route. The Action button uses inheritance to find `:GetTargetInfo('routing)`, which must return a frame like `{target: ..., TargetView: ...}`. For instance, if the user is viewing a llama rancher in our application, the target might be a frame of the form

```
{class: '|rancher:jx|, rancherID: 929, name: {...}}
```

(Note: If your application already maintains `target` and `targetView` slots, implementing `GetTargetInfo` is optional.)

6. The Action button determines **the class of data** to route. The system uses the `ClassOf` function to determine the data class of the target.

7. The Action button determines **which formats** can display or route this data class. The Action button uses the data class of the data and the View Definition registry to build a list of registered formats for that data class. The View Definition registry contains the formats you registered with `RegisterViewDef`. (Note: This list does not include on-screen view definitions that are not routing formats; see the "Routing Gotchas" section of this article for more information.)

8. The Action button determines **which transports** can display at least one of these formats. Both transports and formats have a `dataTypes` slot, and at least one `dataType` in each must match in order for that transport to appear. For instance, if the only available format from the last step is my custom `LlamaFrameFormat`, and its `dataTypes` slot is `['frame']`, then only transports having the value `'frame` in their `dataTypes` array will be included in the Action list.

9. The Action button determines **which route** scripts to add. The Action button uses inheritance to find a `routeScripts` slot containing an array of route scripts. Route script frames have the form

```
{routeScript: 'myExplodeScript, title: "Explode", icon:
kMyIcon}
```

(Note: If you must determine route scripts dynamically, see the *Newton Programmer's Guide* for more information on the `GetRouteScripts` view method.)

10. The Action button displays the newly created list. In this list, a line separates the transports from the route scripts.

11. If a transport is selected, the Action button invokes the current format's `SetUpItem` method, then opens the transport's custom routing slip. If the user switches formats, the system calls the new format's `SetUpItem` method.
12. If a route script is selected, the Action button sends itself the message. If the route script example above were selected, the Action button would execute code similar to

```
self:myExplodeScript(target, targetView);
```

MULTIPLE-ITEM TARGETS

If your application can handle overviews or multiple selection, your routing code must prepare for multiple-item targets. The terminology can be a bit confusing, because there are two types of objects that relate to multiple-item targets. A *multiple-item target* is a special array that contains information about multiple objects (and can be put into a soup), which may include soup entries stored as soup entry aliases. To create a multiple-item target, use the function `CreateTargetCursor`.

You can read the contents of a multiple-item target by calling the `GetTargetCursor` function. It takes a multiple-item target and returns a target cursor, which is an object that responds to the basic soup cursor messages `Next`, `Prev`, and `Entry`, returning `nil` when there are no more items. If the current item was stored as a soup alias, the target cursor methods will resolve the entry alias and will return the soup entry.

Your route script functions must handle multiple-item targets if you have any overviews in your application. Note that you can check to see whether a target is a multiple-item target with the `TargetIsCursor` function. Even if it isn't a multiple-item target, `GetTargetCursor(item)` will correctly return a target cursor containing a single item. Here is an example of how to use `GetTargetCursor` in a route script:

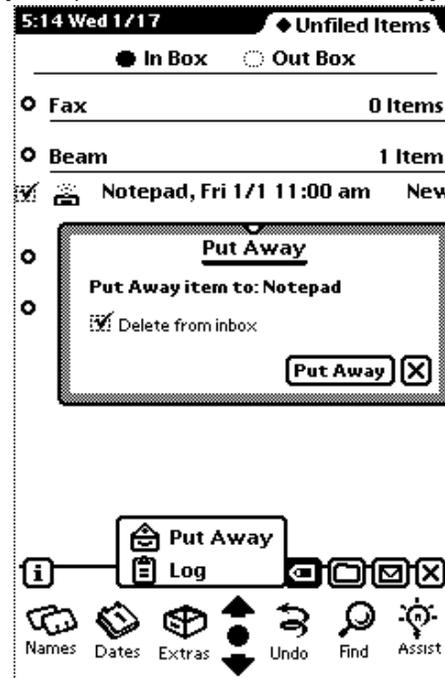
```
myDeleteScript := func(target, targetView)
begin
  local current;
  local tc := GetTargetCursor(target, nil);
  while (current := tc:entry()) do
  begin
    self:HandleMyDelete(current);
    tc:Next();
  end;
end;
```

Routing formats have two different flags to indicate support for multiple-item targets. The `storeCursors` slot determines how the item may be stored in the Out Box. If the format's `storeCursors` slot is `nil`, then multiple-item targets are split into separate Out Box items rather than stored as a multiple-item target. Also, even if `storeCursors` slot is `true`, transports that do not support multiple-item targets will split multiple-item targets into separate Out Box items. The default for `storeCursors` is `true` for `protoPrintFormat` and `nil` for `protoFrameFormat`.

The `usesCursors` flag tells what data print formats are designed to handle. If the format's `usesCursors` slot is set to `nil` (the default), your format will be created once for each item. After printing an item, the value of `target` will change to the value of the next item, and the system will create your print format again. If `usesCursors` is non-`nil`, your format must handle multiple-item targets and use `GetTargetCursor` to iterate through the items.

OTHER ROUTING HOOKS

You might also receive multiple-item targets when putting away items. In the In Box, users can select an item, tap the Transport button (the luggage tag icon) and put away the item from the In Box to another application.



For instance, if a friend beams a business card to you, you can put away that item to the Names application. To allow your application to support Put Away, you must provide a `PutAwayScript` method in your base view that takes one argument: the item. The method must verify that your application can handle the data, perform an appropriate action, and then return `true` to indicate that it succeeded. For instance, the `PutAwayScript` for the Names application might verify that the item is a valid single Names card, create a Names soup entry, and then return `true` if successful (otherwise, it will return `nil`). Note that your `PutAwayScript` may be called when your application is closed.

If you use the `protoActionButton` for routing, items sent to the Out Box will usually have an `appSymbol` slot that is set by the Action button to the current value of `appSymbol`. The `appSymbol` slot is found in your base view using inheritance (NTK creates an `appSymbol` slot in your base view if you haven't already added one). If a user selects Put Away on an item that contains an `appSymbol`, and that application is installed, the application will appear as a choice in a Put Away slip created by the In/Out Box (see the previous illustration).

There is another way to tell the system that you can put away information of a certain data class (which might be necessary if other applications could create or route your data). Do this by registering your data class symbols with the `RegAppClasses` function in your `installScript`. If more than one application could put away an item, the Put Away slip displays a picker to let the user choose a destination application.

Another helpful function is the `Send` function. Some applications want to send items to the Out Box without using a `protoActionButton` and an Action list. In those cases, you must first determine what transport or transport group is appropriate. A *transport group* is a collection of similar

transports in which only one transport is active at one time. These transport groups are identified by symbols, an example of which is the 'mail' group. For instance, we might want a button to automatically email someone. That button might use code like

```
Send('mail, {body: yourData, toRef: [aRecipient]}).
```

The important slots to add to the item are a `body` slot containing your data (the target), and the recipients in the `toRef` slot. You can also add a `title` slot (the subject line) and an `appSymbol` slot (for Put Away). See the *Newton Programmer's Guide (NPG)* for a list of other slots you could add to the item, although some items' slots do not apply to all transports.

The recipients in the `toRef` slot are represented by an array of `nameRefs`, which are recipient frames returned from choosers based on `protoListPicker` (this includes the common "to" and "cc" lines in routing slips). If you have a reference to a Names soup entry and wish to convert it to a `nameRef`, you should use the `nameRef`'s data definition — an object that describes how to manipulate `nameRefs` of a specific class (for example, `|nameRef.fax|` and `|nameRef.email|`). Note that every transport stores its preferred `nameRef` class symbol (its *addressing class*) in its `addressingClass` slot.

For instance, imagine we want to fax a letter programatically. If we wanted to send to someone who is not in the Names file, we could substitute a frame containing the minimal slots for addressing (which varies among addressing classes and transports). To fax the letter, you could use code like the following example:

```
faxLetter := func()
begin
  local transportSym := '|faxSend:Newton|;

  rancher := {
    name: {last: "Bell", first: "J. Christopher"},
    phone: SetClass("408 555 1212", '|string.fax|)
  };

  target := {class: '|letter:jX|, style: 'VisitUsAgainSoon};

  // TransportNotify sends a message to our transport
  // to create a new item that we pass to Send(...)
  item := TransportNotify(transportSym, 'NewItem, [nil]);

  aClass := GetItemTransport(item).addressingClass;

  item.body := target;
  item.toRef := [GetDataDefs(aClass):MakeNameRef(rancher,
aClass)];

  // Register a format with RegisterViewDef (see rest of article...)
  // It must have this symbol in its symbol slot.
  item.currentFormat := kMyViewFormatSym;

  Send(transportSym, item); // submit the item to the outbox
end;
```

If you are using `Send` with a transport that supports text, that transport must export items to text by calling your routing format's `textScript` method. That method must convert `item.body` (`item` is one of the arguments to your `textScript` method) and return the text to be sent.

WHERE DO I IMPLEMENT THIS?

Some of the routing hooks use inheritance to find the methods or variables. More specifically, the Action button uses its message context (its `_proto` and `_parent` chain) and sends messages to itself. For example, if you implement a `GetTargetInfo` method in your base view, the Action

button will find the base view method by parent inheritance. However, when `GetTargetInfo` executes, `self` will be the Action button. Although many developers put these messages in their base view, the context-sensitive nature of routing allows flexibility in where you implement your code. For instance, if you want your subviews to route different information or handle it differently, you can implement some variables and methods in your subviews.

Note that formats and applications exist in separate places. The Action button finds routing formats because they are registered globally (see `RegisterViewDef`, mentioned above). You can use this architecture to allow other developers to extend your application. If you publish the format of your data, other developers can globally register new formats for your data classes. For example, you can register a `protoPrintFormat` on the 'person class so that users of the Names application (and other applications that use the 'person class) will see your new format as a choice in the Print or Fax routing slip.

Common Routing Questions	Where to Implement Relevant Code
What is the target?	Action button message context
What route scripts are available?	Action button message context
Where to implement routeScripts?	Action button message context
What routing formats are available?	ViewDef registry
What types of transports should be used?	ViewDef registry and the format's <code>dataTypes</code> slot
How to visually represent the data?	Routing format (based on <code>protoPrintFormat</code>)
Where to export text before it is sent?	Routing format (<code>TextScript</code> method)
Where to manipulate data before sending?	Routing format (<code>SetupItem</code> method)
Where to execute slow pre-fax code?	Routing format (<code>FormatInitScript</code> method)
Where to implement PutAwayScript?	Application base view

CHECKLISTS FOR ROUTING TO BUILT-IN TRANSPORTS

Required for Routing:

- Add a `routeScripts` slot[†]
- Add a `GetTargetInfo` method[†]
- Add an Action button (a `protoActionButton`)
- Add `[Un]RegisterViewDef` calls in your `installScript` and `removeScript`
- With `RegisterViewDef`, access formats with `partFrame.theForm.myBaseViewSlot` (Note: Don't use the `GetLayout` NTK function for this! See the DTS Routing samples for more information.)

Required for Print, Fax, and future view transports:

- Create layouts based on `protoPrintFormat` in NTK
- Create a `symbol` slot containing your unique format signature
- Create a `title` slot containing your format title
- Add child views (draw out subviews and/or use `viewSetupChildrenScript`)
- Add `printNextPageScript` method
- If you need to do pre-routing setup, add a `SetupItem` method
- If you need to do slow pre-fax initialization, add a `FormatInitScript` method
- Put a reference to the format in a base view slot using the NTK `GetLayout` function. You will reference this from your

`installScript` (see `RegisterViewDef`, above).

Required for Beam, Mail, and other frame or text transports:

- Create a format based on `protoFrameFormat`
- Create a `symbol` slot containing your unique format signature
- Create a `title` slot containing your format title
- To support text export, supply a `TextScript` method
- If you need to do pre-routing setup, add a `SetupItem` method
- Put a reference to the format in a base view slot. You will reference this from your `installScript` (see `RegisterViewDef`, above).
- For support for Put Away, add `[Un]RegAppClasses` calls in your `installScript` and `removeScript`
- For support for Put Away, add a `PutAwayScript` method to your base view

† Often in base view, but context-sensitive from Action button

ROUTING GOTCHAS

Here are some of the common mistakes or “gotchas” to keep in mind when implementing routing in your application.

The difference between routing formats and on-screen view definitions (viewDefs) confuses many people. They are both registered and unregistered using the same functions, but they serve different purposes. For more information on on-screen stationery and viewDefs, see the *NPG*'s Stationery chapter. On-screen viewDefs cannot be used as routing formats, and routing formats cannot be used as on-screen viewDefs. This is because of a `type` slot that indicates the type of viewDef. For routing formats, which are based on `protoRouteFormat`, the type will be `printFormat` or `routeFormat`. This is to distinguish routing formats from on-screen viewDef types like `viewer` and `editor`.

If you have a layout you want to use for both on-screen viewing and printing, you must create two different layouts. However, since both layouts will be similar, you will end up laying out two views that are almost identical. To save development time, you can create a user proto (see the NTK user manual) that encapsulates the common behavior, and use that as the basis for your on-screen view and print formats. The only gotcha is that print formats automatically set up a `target` slot that contains the data to display. In order to standardize the behavior of your user proto, you may need to make modifications to your on-screen view to create a `target` slot containing your on-screen data.

Another common problem is setting the data class incorrectly. When your `GetTargetView` method returns the `targetInfo` frame, the `targetInfo` frame must have a `target` with a meaningful class. That means that `classof(target)` represents a unique symbol representing your data class. Since the `target` is usually a frame, this means that your `target` must have a `class` slot, or a class slot is added when `target` is returned by your `GetTargetInfo` method. As mentioned in the “From Formats to Transports” section of this article, the Action button uses the class of the data to decide what routing formats and transports are available.

Another gotcha is enabling routing from overviews. There is some special code in routing that makes overviews and multiple selections easier for some applications. If your application wants to register formats only for the types `'frame`, `'text`, and `'view`, and your print formats do not need special initialization, you can use the special `'newOverview` data class to represent multiple-item selections in overviews.

If your `GetTargetInfo` method returns a multiple-item target, you can use the code `CreateTargetCursor('newOverview, myItems)` to create a multiple-item target with this special data class. This enables your application to use built-in routing formats registered on the `'newOverview` class (do not register your own viewDefs on this data class). Since the defaults of the formats registered on this class have their `usesCursors` slot set to `nil`, the system will use the helpful default routing format behaviors mentioned in the “Multiple-Item Targets” section of this article, which is what most developers want for printing and routing multiple items.

There are some limitations to this approach. As mentioned above, your print formats might need special initialization in a `SetupItem` method or a `formatInitScript` method to avoid timing out during fax connections. If so, please note that your formats are not guaranteed to get their `SetupItem` or `FormatInitScript` messages when using the `'newOverview` class for your multiple-item targets.

If you want to enable datatypes other than `'frame`, `'text`, and `'view`, or need the `SetupItem` or `FormatInitScript` messages for your print formats, you must use your own data class symbol when creating the multiple-item target with `CreateTargetCursor`. For instance, let's suppose our application can print and fax from a llama layout, but cannot use other transports like beam and mail. Since the default `'newOverview` behavior enables transports like beam and mail because it registers formats with `'frame` and `'text` datatypes, we cannot use the `'newOverview` behavior. Instead, we can use code like the following to create our multiple item target:

```
CreateTargetCursor(kLlamaClassSym, selectedItems);
```

For our `kLlamaClassSym` data class, we register only routing formats based on `protoPrintFormat` so that we would see only Print and Fax as transport choices in the Action list.

If you want your routing formats to handle multiple items, set their `usesCursors` slots to `true` so that you can print multiple items on a page (using `GetTargetCursor` to traverse the item list) or route multiple items in a different way. For instance, a format based on `protoFrameFormat` could traverse the list of items in its `SetupItem` method and set `item.body` to a new Virtual Binary Object (VBO) representing all the items (see the *NPG*'s Data Storage chapter for more information). Since the system stores VBOs on a store, not in NewtonScript memory, you can route large amounts of data in a single Out Box item without running low on NewtonScript memory. Be sure to give your Virtual Binary Object a meaningful class symbol so that your application can check for this data class during Put Away.

On a similar topic, some developers use the optional `NewtApp` application framework to design their applications (see the *NPG* for more information). If your application uses the `NewtApp` framework, and you use the layout proto called `newtOverview`, and you do not want to use the `'newOverview` multiple-item behavior mentioned above, you must do some extra work. You must supply an `overviewTargetClass` slot in your layout (or somewhere in its `_parent` chain) containing a symbol that will be used as the data class for multiple-item targets in that layout. NTJ

These are the basics you need to know before implementing routing in your Newton 2.0 application. This information will get you started with routing; once you've mastered these concepts, you'll be able to move around all the information you want.

Writing a Transport

by J. Christopher Bell, Apple Computer, Inc.

WHAT IS A TRANSPORT?

When you tap that ubiquitous envelope icon, the Action button, it opens up a list of ways you can move information.



The items above the line in the Action list are services that move information outside the Newton device. For instance, you can Beam a package from the Extras drawer, Print a note, or Mail an item to a friend. These communications services are called *transports*. There are built-in transports, but developers can add choices to Action lists by creating new transports that users can install as packages. In one sense, transports are a user interface for complex communications code. This article discusses some of the things to keep in mind before writing a transport for Newton 2.0.

Before reading this article, you should be familiar with *routing*, which is the general term for moving information using the Action button and the In/Out Box. Routing is covered in more detail in the *Newton Programmer's Guide (NPG)* and in the "Move It!" article in this issue of the *Newton Technology Journal*.

The most important term to understand is *dataType*. A *dataType* is a generic classification of the way in which information is sent outside the Newton device. Common *dataTypes* are plain text (the 'text *dataType*), NewtonScript frames (the 'frame *dataType*), and imaging layouts (the 'view *dataType*). You can also define your own *dataTypes*, although applications must know details about your custom *dataType* in order to take advantage of this ability. See the "Move It!" article in this issue for more information.

You also should know the basics of *Routing Formats*. Print Format layouts and pre-routing initialization are encapsulated into objects called Routing Formats. These objects format the data for use outside your application.

When you implement your transport, much of your time will be spent designing, coding, and testing the endpoint communications code with `protoBasicEndpoint`. Check out the *NPG 2.0* and DTS Communications samples for tips on designing and debugging your endpoint code.

Once you design the basic communications code, you can test it by creating

a `protoTransport`, registering it with the `RegTransport` function, and hooking your endpoint code into the `SendRequest` and `ReceiveRequest` messages, which will be described later. If your transport can send data (it could just receive data), you will probably design a routing slip to test your code with different recipients and options. This article discusses design issues and the basic transport APIs to get you started writing a transport.

BEFORE WRITING A TRANSPORT

If your communications service is similar to Beam or Mail, you probably will structure your code as a Newton transport. However, not all communications code is well suited to becoming a transport. You might want to take time to determine whether a transport is best for your project. Although they may not apply in all situations, here are some guidelines to follow in transport design. When you read these guidelines, keep in mind that your transport does not have to both send and receive data; it can just receive or just send.

Transport Guidelines

If you answer "yes" to all of the following questions, a transport might be a good approach:

- Would your endpoint code have a user interface similar to a built-in transport (for example, routing slips, using the In/Out Box, status views)?
- Would queuing items to send make sense?
- Would all your communications setup preferences belong in the In/Out Box?
- Would invoking Send and/or Receive from the In/Out Box make sense?

The following are indications that you might not want to structure your code as a transport:

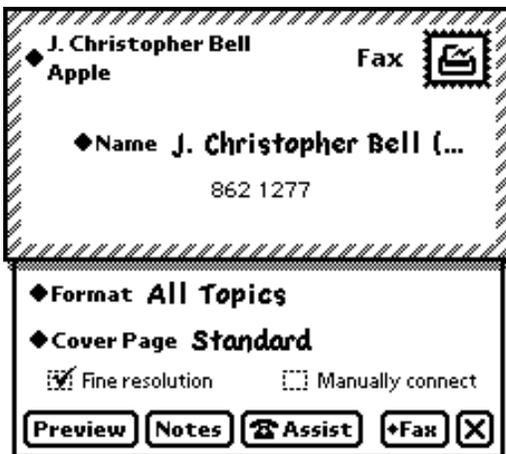
- You answered "no" to one of the Transport Guidelines above.
- Incoming and outgoing info is a stream, rather than distinct items to be previewed or queued.
- Your code's purpose is to synchronize Newton soups with a non-Newton database or application. Implementing a synchronize action within a single application might be simpler.
- Your transport will be available only to your own applications. (See note below)
- Your transport must dynamically manipulate In/Out Box items when

connected to other devices. It is possible to design transports that use remote items. This means that items in the In Box (not the Out Box) can download summary information from services that offer additional items for users to view or open. However, if your endpoint code must add items to the Out Box or dynamically add or delete records from the In/Out Box, designing your code as an application may more closely fit user expectations.

- Your transport is similar to Print or Fax, such that it draws views on a page. If so, you might need to write a printer driver. Contact the Newton Developer Relations Group at NEWTONDEV@applelink.apple.com for more information.

Note: Some developers want their transports to be available only to their own applications. Sometimes this is the best way to do the job, but not always. Performing setup and activity (preferences, initiating Send/Receive) within the application controlling the services might be the most straightforward user interface. As an alternative to implementing a transport that works only with one application, you might encapsulate the endpoint code within one application or use the Unit import/export mechanism to share code among several applications. See the DTS Q&As for more information on the Unit import/export mechanism.

You might find that according to the guidelines above, your transport differs from other transports. If so, your transport may need a slightly different user interface. For instance, if your transport creates lots of small Out Box items, check the *NPG* for more information about the item hidden slot, which allows items to remain invisible in the Out Box.



To make your transport available only to your applications, create a custom `dataType` like `'|MyDataBaseRecord:MYSIG|'` to be supported only by your own Routing Formats (in their `dataTypes` slot), instead of standard `dataTypes` like `'frame'` and `'text'`.

ROUTING SLIPS

Designing a basic routing slip is easy when you use the handy built-in `protoFullRouteSlip`. It contains most of the user interface elements in a routing slip, including the Format picker, the envelope appearance, the return address `proto`, the close box, and the Send button. The most complex design decisions for routing slips involve your recipient information. Note that the top part of the routing slip contains recipient controls, and the

bottom part contains formatting controls.

If your transport is like Beam, which needs no recipient controls when its preferences are set to Send Later, this will be easy. For most transports, you will need to do work to let the user tell your transport who should receive this item.

A common user interface element for choosing recipients is `protoAddressPicker`, which allows users to select recipients from the Names file for “to” or “cc” components of a message. If your transport can use the addressee type used by Mail, then you don’t need to do anything special. If you want to use the addressee type for phone numbers or fax numbers, you can override the `class` slot in `protoAddressPicker`. For example, to pick phone numbers you would specify `'|nameRef.phone|'`.

If you need custom addressing information that the built-in classes do not supply (such as email that can’t use Internet addresses), you will need to add your own subclass of the `nameRef` data definition. See the *NPG* (`protoListPicker`, `protoNameRefDataDef`) and DTS samples for more information on how to add your own subclass of `nameRef`.

In the bottom half of the routing slip, you can add formatting controls such as a subject editor or resolution settings, or other controls specific to your transport. We recommend making this part of the routing slip as simple as possible by putting most setup controls in your transport preferences.

There are some subtle variations in the size of this area of the routing slip that make view layout tricky. For instance, if there is only one format option, the Format picker is not visible and the height changes. To lay out views in the bottom part of the routing slip, the easiest way is to use the routing slip method `BottomOfSlip` to find the offset of the bottom of the routing slip.

When the user tries to send an item, your routing slip gets a `PrepareToSend` message. In your `PrepareToSend` method, extract information from your routing slip controls, store that information in slots in fields (which will become the Out Box soup entry), and call the inherited method to continue to send the item. Store recipient information in standard slots called `toRef`, `cc`, and `bcc`. Those must contain an array of `nameRefs`, which are recipient frames returned from choosers based on `protoListPicker` and `protoAddressPicker`.

One special routing slips interface relates to transport groups. You could make your transport a member of the ‘mail’ group by including that symbol in the transport’s `group` slot. This will display your transport in the Action list as “Mail,” rather than by the title of your transport (for example, “MySuperMail”).

When more than one transport in the same group is installed, the user must use the routing slip to switch between transport group members. The transport title in the routing slip will contain a diamond (indicating multiple choices), and the user can tap the transport title to open a list of other transport group members from which to choose. If the user switches transports using the Group Transport picker, the routing slip will close, and the new transport’s routing slip will open.

SENDREQUEST AND RECEIVEREQUEST

Most of your communications code will execute in response to a `SendRequest` or `ReceiveRequest` message. In `SendRequest`, you must iterate over the items by sending your transport the `ItemRequest` message to retrieve the corresponding items. Your endpoint code can execute asynchronously and call `ItemRequest` for more items until the value is `nil`. When an item is completed, your transport can notify the Out Box of the new status using `transport:ItemCompleted(item,`

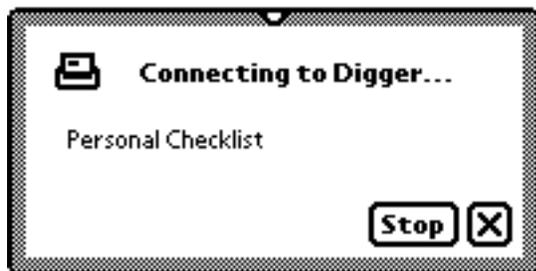
state, `errorNumOrNil`). If the state is 'sent, the item will be deleted from the Out Box. Pass `nil` for the state if there was an error, to indicate that the item should stay in the Out Box.

Most transports that receive items respond to the `ReceiveRequest` message by connecting to another device and retrieving as many items as possible. Submit the items to the In Box by using `ItemCompleted` with the status 'received. You can also implement a browse feature using remote items. To do this, download the title and recipient information, and set the item's `remote` slot to a non-`nil` value. The user will be able to sort or view the summaries, but tapping the item will cause a second `ReceiveRequest` to be issued with the argument `cause` set to 'remote, indicating that your transport must download the entire item and call `ItemCompleted` again. When creating the remote item, you will probably encode some message ID in the item to indicate how to retrieve the full item if the user taps it.

OTHER TRANSPORT HOOKS

If your transport's `dataTypes` slot contains the 'text' symbol (in other words, it supports text), you will use the `ItemToText` function to extract the text from the item. The primary function of the `ItemToText` function is to send the Routing Format `textScript` method and store the text in the item's `body` slot in the form `{class: 'text', text: "whatever..."}`. See the DTS Transport samples for more information about how to use the `ItemToText` function, which is currently available as a stream file to include in your NTK project.

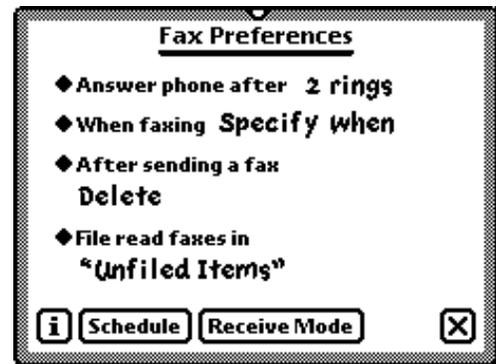
The basic `protoTransport` provides helpful defaults to handle views that show transport status to the user.



Most of the default status view behavior can be achieved using commands like `transport:SetStatusDialog('Connecting, 'vStatus, "Connecting to"&&printrname&"...")`. These defaults are probably what you want to use during development and debugging so that you can concentrate efforts on your endpoint code. Closing the dialog with `SetStatusDialog` is as easy as setting the first argument (the status) to 'idle. If you want a status indicator other than the built-in gauge indicator, page indicator, or barber pole indicator, you can fully customize the dialogs using `protoStatusTemplate` (see the DTS samples and the *NPG* for more information).

The In/Out Box is the main user interface to your transports, and there are several user interface hooks available to transport developers. (Note: Do not send messages directly to the In/Out Box application itself, because the implementation might change.)

The most often used transport hook in the In/Out Box is the transport preferences slip, accessible via the Info button in the In/Out Box (see the following illustration).



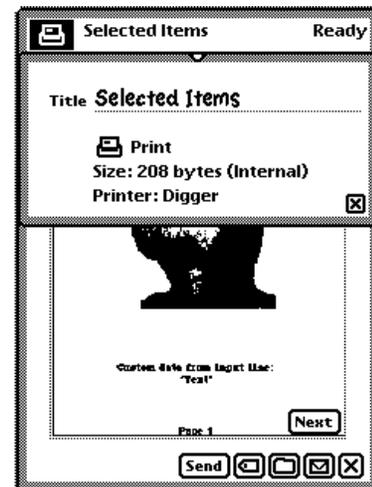
You provide a layout based on `protoTransportPrefs` that contains connection options, defaults, and logging and filing options. Some optional controls are provided by the proto.

When a user taps an item in the In/Out Box, the In/Out Box opens an item viewer (see the following illustration). Within the item viewer are several buttons, including an Action button to reroute items to another transport, a Show button that is sometimes visible, and the Transport button (the one with a luggage-tag icon). The Transport button opens a picker with default commands (Readdress, Log, and Put Away, depending on the situation), but your transport can add actions like Reply or Forward.

For example, to add a transport action called Explode, implement a `GetTransportScripts` transport method that returns an array of transport script frames (similar to route script frames) which includes a frame like

```
{title: "Explode", routeScript: 'myTransportMethod, appSymbol: kMyTransportSym}
```

You can also design an item info slip for your transport which is opened when the user taps the transport icon in the upper-left corner of the item viewer.



Default information includes the item's size and subject, but you can fully customize the slip and add transport-specific information or item history that might be relevant for In Box or Out Box items (see the *NPG* for more info on `protoTransportHeader`).

These are the APIs you should know about before writing your communications code and designing your own transport. Happy transporting!

continued from page 1

C++ and NewtonScript

C++ is intended to be a superset of C. This goal is considered critical, because C is the most popular systems programming language. Because C++ includes the low-level operations of C, it can be used for very low-level system code — manipulating hardware registers, for example. Algorithms can be written in C++ to take maximum advantage of the processor hardware. Most importantly, existing code written in C can be compiled by a C++ compiler with minimal changes.

Performance was one of the highest priorities in the design of C++. As a general rule, no feature was included unless it was possible to implement it at zero cost for code that didn't use it directly.

In order to be a viable C replacement, C++ had to be “pin-compatible” with existing development systems. In particular, the output of the C++ compiler had to link with C code, and C++ and C data structures had to interoperate.

The same operations that are so useful for low-level system code are easy to misuse accidentally. Common programming mistakes in C++ can corrupt the runtime environment, resulting in system crashes or data corruption. The program must be designed to deallocate memory manually, an error-prone process in which bugs are often difficult to locate.

In its early incarnations, C++ did not include features such as exception handling, multiple inheritance, parameterized classes, and run-time type information. Over time, these features have been added to the language, always in a manner consistent with the overriding priority of performance. C++ is now a complicated language; it requires years of experience to fully understand and use all its features.

NewtonScript

NewtonScript was intended to make writing Newton applications as easy as possible. Its critical design goals included:

- Natural interaction with the Newton application framework
- Small program size and low RAM footprint
- Safe execution
- Simplicity

The NewtonScript language is based on an object model that is used throughout the Newton system. In-line constructors and built-in access operators make it easy to construct and manipulate objects.

NewtonScript uses prototype-based inheritance, which is the most natural way to write user-interface code. The view system is designed around NewtonScript's object model, which makes it easy to reuse user interface elements provided by the system or the programmer.

Because the Newton OS is intended to support small devices, NewtonScript was designed for small amounts of memory. NewtonScript executable code is very small — several times smaller than equivalent native code. The prototype-based inheritance model makes it easy to minimize RAM usage by putting only the varying slots of an object in RAM.

NewtonScript is a safe language. All operations are checked for validity, and violating the rules causes a controlled exception rather than a system crash. This makes debugging much easier, and lessens the need for system “firewalls” between applications. Memory is deallocated automatically by the language at runtime; this eliminates memory-

management bugs such as dangling pointers and multiple deallocations, and also simplifies program design.

The language is small and relatively simple in syntax and semantics. Most programmers can read the code almost instantly, and learn to take advantage of all the important features in a few months. Despite its simplicity, however, NewtonScript is a complete general-purpose language.

The execution model of NewtonScript is divorced from the hardware; instead, it is a set of well-defined operations on typed data. There is no access to the underlying representation, and thus no “unsafe casts” between types. This makes the language safe and portable, but it also means the programmer can't take full advantage of the hardware to get maximal performance.

LANGUAGE RELATIONSHIP

C++ and NewtonScript have complementary strengths. The “C-ness” of C++ makes it good for manipulating hardware (device drivers, for example) and for writing very fast code, at the expense of unsafe features and lack of flexibility. NewtonScript's safety, expressiveness, and flexibility make it good for higher-level application code; the price is in performance and incompatibility with existing C/C++ source.

The native NewtonScript compiler included with Newton Toolkit allows you to reach roughly the same performance as C++ code for many programs. Algorithms based on NewtonScript objects are usually about the same speed in both languages. C++ code that uses the native data structures of C++ will generally be faster.

Internally, the Newton OS has always made use of both languages. NewtonScript is used at the high level to create the user interface objects and built-in applications, and to tie the system together. C++ is used for the low level, such as the OS kernel, graphics primitives, device drivers, and the NewtonScript interpreter. The choice of language is dictated by the requirements of the system component; when performance is critical, we write in C++, and when safety or space are more important, we write in NewtonScript.

This may sound familiar, because the idea of using a high-level object model as a system-structuring device is now entering the mainstream. SOM and OLE are examples of system-wide object models that connect lower-level code together. A SOM or OLE object may be implemented in C++ or some other language; the user of the object doesn't need to know which.

The Newton OS uses the NewtonScript object model for a similar purpose. The system is structured as a set of NewtonScript objects, but any particular function may be implemented in C++ if needed. An API is provided for C++ code to create and manipulate NewtonScript objects, so a C++ function can interact with the rest of the system just as well as a NewtonScript function.

NEWTON C++ TOOLS

With Newton C++ Tools, you will be able to write code in C++ that fits into the overall NewtonScript system framework. The basic idea is that a group of C++ source files can be compiled and linked into a “native module,” which is a chunk of native code with function entry points defined by you. When the native module is included in a Newton Toolkit project, each entry point is

made available as a NewtonScript function object.

The rest depends on how you assign these entry points to your objects. The simplest technique is simply to call the functions directly. You can rewrite NewtonScript methods in C++ by putting C++ entry points in slots in place of “func” expressions. A C++ class can be given a NewtonScript interface by creating a frame containing “wrapper” functions for the class methods.

Through the APIs provided to C++ code, you can create and manipulate NewtonScript objects, call NewtonScript functions, and send NewtonScript messages. The interface between NewtonScript and C++ is always in terms of NewtonScript objects, but it’s easy to write “glue routines” to convert to and from C++ data types.

Newton C++ Tools works best on “engine” code — a fairly well isolated part of an application that deals with a core set of data structures and operations. Many applications consist of an engine surrounded by a user

interface. Newton C++ Tools can be used for the engine (or, preferably, just the speed-critical parts of it), while NewtonScript is used for the rest of the application. This is particularly useful when a complete, debugged engine already exists in the form of C++ code.

Newton devices tend to have fewer resources — such as memory — than machines for which existing C++ code was written. Thus, you should be careful when using existing code to ensure that it can operate within the constraints of the intended platform.

Newton C++ Tools is the latest example of how Apple is continuing to provide a more open, flexible development platform for software developers. Even with the limitations described above, we feel that many developers will be able to take advantage of Newton C++ Tools to enable them to more easily port existing routines written in C or C++ to the Newton platform.

NTJ

continued from page 1

Apple Announces New MessagePad 130 with Newton 2.0!

NEW LCD

The MessagePad 130 features a new transfective EL backlight which allows viewing of data in any lighting conditions. Be it in a dimly lit conference room, a dark storage facility, or at night in the car, with on-demand backlighting, information can now be easily viewed and accessed. The EL backlight is easy to turn on, with just a flip of the power switch, and its time-out feature is user-controlled via Preferences.

Third-party Newton developers will have access to the backlight APIs to take advantage of backlight controls in software if desired. Three NewtonScript calls have been added to control the backlight:

`BackLightPresent()`
Returns true if the unit has a backlight

`BackLightStatus()`
Returns true if the backlight is on

`BackLight(RefArg onOrOffArg)`
Turns the backlight on or off. Returns the previous state

MORE SYSTEM MEMORY

The MessagePad 130 comes with an additional 512K of system memory bringing the total to 1MB of system memory, which will allow for better performance with communications solutions such as TCP/IP and wireless LANs. The addition of system memory will also allow for better performance with multi-tasking support. The system heap is also expanded to utilize the extra memory.

The change in heap does not mark a new standard for Apple Newton based hardware. The additional heap is intended to enable more robust communications applications. The MessagePad 120 will continue to be sold in the channel alongside the MessagePad 130 and will have the same available heap as it does today. Please keep this in mind when designing and developing your applications.

MORE DURABLE, NONGLARE SCREEN

The MessagePad 130 now features a new, more durable nonglare screen which makes for easier viewing of information in a variety of lighting environments. With the improvements to the durability of the tablet, the MessagePad 130 provides for a more robust mobile forms and data capture tool.

THE MESSAGEPAD 130 OFFER

The MessagePad 130 comes with a variety of built-in productivity tools such as a Call Log, Time Zone map, a Formula application, and Pocket Quicken (US only) to name a few. Newton Backup Utility and serial cables for Mac OS- and Windows-based computers are also included for backing up and restoring information on a PC as well as installing packages onto the MessagePad.

THE APPLE MESSAGEPAD PRODUCT LINE

The MessagePad 120 is the entry-level Newton PDA from Apple which retails for \$699.00 in the US. The MessagePad 130 is the high-end Newton PDA with backlighting and more system RAM retailing at \$799.00 in the US.

NTJ

Seven Ways to Create the Killer Newton Application

by Guy Kawasaki

Note From The Editor:

September 1995 Newton Platform Developers Conference attendees have continually commented on the wisdom of the advice delivered in Guy Kawasaki's keynote speech. We've had repeated requests to publish the text of his presentation. While we are not able to publish the entire speech word-for-word, Guy has put together the following points, which are the key elements of his Newton application development strategy. Enjoy!

- 1) Ignore the past. "Porting" software from one platform to another is for wimps. Thinking that old application categories is what will sell on a new platform is dumb. If you want to create a killer application, you need to jump to a new curve, not milk a previous one.
- 2) Let a thousand flowers bloom. When people come up to you with weird ideas, listen to them. (Newton-based devices as tour guides for museums – Imagine that!) Apple didn't evangelize music software for Macintosh. That software, and therefore that market, developed because thousands of flowers were blooming.
- 3) Lead, don't follow, Apple. Desktop publishing was never planned by Apple. If it weren't for Paul Brainerd (PageMaker) and John Warnock (PostScript), Apple might not be around today. They led Apple into desktop publishing. Go where you want to go, and Apple might be able to follow. Even if Apple isn't following, go there alone.
- 4) Ignore your customers. Customers are great at telling you how to evolve a product – new features, etc. But customers suck as far as being able to tell you how to create a revolutionary product. If we had listened to customers in 1984, we'd be making the Apple XXII now. You have to take the shot – that's why you get the big bucks.
- 5) Create the product you want to use. Woz designed the Apple I and Apple II because they were the computers he wanted to use – not the result of data from some high falutin market research study or focus group. If you create the product you want to use, at least you know there's one customer for sure. That's more than research can tell you.
- 6) Don't raise too much money. Too little money is a much better problem than too much money. A scarcity of resources will force you to be more clever, more guerilla-like, and more aggressive. An abundance of

resources will only encourage you to buy nicer furniture, cooler stationary, and more shrimp for your press conference.

- 7) Don't worry, be crappy. Although I'll swear I never wrote this, during the early stage of a product like Newton (and it's still early), I recommend that you ship something right away to test your product concept. If you wait for that "perfect version" with all the features your beta sites requested, you might die on the vine. Get the essential features of your product done and ship. Validate the product. Then revise quickly.

Guy Kawasaki is an Apple Fellow and legend in his own mind. He is the author of How to Drive Your Competition Crazy and a Forbes columnist.

Stop the hegemony. Join EvangeList. Send an email to <macway-request@solutions.apple.com> for an automatic reply. (Any message will work.) Archives are at: <<http://wais.sensei.com.au/searchform.html>>.

NTJ



Newton®

If you have an idea
for an article you'd like to write for Newton
Technology Journal, send it via Internet to:
NEWTONDEV@applelink.apple.com
or AppleLink: NEWTONDEV

Dear Newton Developer,

A little over a year and a half ago, we sent the first issue of the *Newton Technology Journal* to the printer, with high hopes and expectations for how it would be received, and what it might become. And in these last 18 months, we have not been disappointed. The *Newton Technology Journal* has proven to be an excellent medium for explaining Newton programming techniques and covering some detailed areas of the Newton OS not covered in our documentation or other support tools. We've used it to deliver the latest news and marketing information from Apple's Newton Systems Group, and as a forum for announcing new products and encouraging you to keep up the great work.

While we're thrilled with the feedback we've gotten from you on the *Journal's* usefulness, we are not yet ready to settle and leave it at that. The last seven issues have been full of excellent articles written by Apple engineers, trainers, product marketing managers, and a few developers with a knack for a particular programming technique. While we want lots of those to continue, we also want the publication to become a forum for folks in the Newton development community to share programming tips and tricks, share market successes and deliver additional programming information. How about a Question and Answer column – something like Ask Dr. Llama? You send in the questions, we provide the answers. This is a publication dedicated solely to your pursuit of excellent Newton platform solutions. Let us know how you want the magazine to improve, and we'll make it happen. Also, let us know what you've enjoyed and want to see more of, so we can continue to deliver the content you need.

More than just providing us feedback, we'd like to see you start authoring articles. Now that so many of you are experienced Newton programmers, we'd like you to share your expertise with others in the community. Getting published is easy – all you need to do is let us know what your idea is, and we'll let you know how and when we can fit it into the publishing schedule. Or, if you've written an article already and want it published, just submit it! We'll let you know if and when we can print it. Of course there are some more details (like the legal stuff), but we'll pass them all on when you send us your ideas or request to be published. Just send your requests, ideas, or articles to newtondev@applelink.apple.com.

So, here's to another year of great articles and content. We look forward to hearing from you soon!

– *Newton Technology Journal Editorial Review Board*
Apple Computer, Inc.



Apple Developer Group

Newton Developer Programs

Apple offers three programs for Newton developers – the Newton Associates Program, the Newton Associates Plus Program and the Newton Partners Program. The Newton Associates Program is a low cost, self-help development program. The Newton Associates Plus Program provides for developers who need a limited amount of code-level support and options. The Newton Partners Program is designed for developers who need unlimited expert-level development. All programs provide focused Newton development information and discounts on development hardware, software, and tools – all of which can reduce your organization's development time and costs.

Newton Associates Program

This program is specially designed to provide low-cost, self-help development resources to Newton developers. Participants gain access to online technical information and receive monthly mailings of essential Newton development information. With the discounts that participants receive on everything from development hardware to training, many find that their annual fee is recouped in the first few months of membership.

Self-Help Technical Support

- Online technical information and developer forums
- Access to Apple's technical Q&A reference library
- Use of Apple's Third-Party Compatibility Test Lab

Newton Developer Mailing

- *Newton Technology Journal* – six issues per year
- *Newton Developer CD* – four releases per year which may include:
 - Newton Sample Code
 - Newton Q & A's
 - Newton System Software updates
 - Marketing and business information
- *Apple Directions* – *The Developer Business Report*
- *Newton Platform News & Information*

Savings on Hardware, Tools, and Training

- Discounts on development-related Apple hardware
- Apple Newton development tool updates
- Discounted rates on Apple's online service
- US \$100 Newton development training discount

Other

- Developer Support Center Services
- Developer conference invitations
- *Apple Developer University Catalog*
- *APDA Tools Catalog*

Annual fees are \$250.

Newton Partners Program

This expert-level development support program helps developers create products and services compatible with Newton products. Newton Partners receive all Newton Associates Program features, as well as unlimited programming-level development support via electronic mail, discounts on five additional Newton development units, and participation in select marketing opportunities.

With this program's focused approach to the delivery of Newton-specific information, the Newton Partners Program, more than ever, can help keep your projects on the fast track and reduce development costs.

Unlimited Expert Newton Programming-level Support

- One-to-one technical support via e-mail

Apple Newton Hardware

- Discounts on five additional Newton development units

Pre-release Hardware and Software

- Consideration as a test site for pre-release Newton products

Marketing Activities

- Participation in select Apple-sponsored marketing and PR activities

All Newton Associates Program Features:

- Developer Support Center Services
- Self-help technical support
- Newton Developer mailing
- Savings on hardware, tools, and training

Annual fees are \$1500.

New: Newton Associates Plus Program

This new program now offers a new option to developers who need more than self-help information, but less than unlimited technical support. Developers receive all of the same self-help features of the Newton Associates Program, plus the option of submitting up to 10 development code-level questions to the Newton Systems Group DTS team via e-mail.

Newton Associates Plus Program Features:

- All of the features of the Newton Associates Program
- Up to 10 code-level questions via e-mail

Annual fees are \$500.

For Information on All Apple Developer Programs Call the Developer Support Center for information or an application. Developers outside the United States and Canada should contact their local Apple office for information about local programs.

Developer Support Center
at (408) 974-4897
Apple Computer, Inc.
1 Infinite Loop, M/S 303-1P
Cupertino, CA 95014-6299

AppleLink: DEVSUPPORT

Internet: devsupport@applelink.apple.com



Newton