Newton® Technology

JOURNAL

Volume I, Number 5 *November 1995*

Inside This Issue

Newton®

Newton Directions

Newton 2.0: What's the Big Idea?

by Joseph Ansanelli & the Newton Platform Marketing Team, Apple Computer, Inc.

A lot has happened since Apple brought the industry's first personal digital assistant (PDA) to market in 1993. Many companies have entered the market, and some competitors have already fallen by the wayside. Technology has advanced. Improvements have been made on many fronts – software, communications capabilities, and services options have grown more sophisticated and useful. Today, though the world is less inclined to view PDAs as a panacea for all technological and business problems, there are many customers who are championing the usefulness of PDAs.

Most important, the Newton PDA platform continues to gain momentum in the market. With the advent of Apple's second-generation PDA software platform – Newton 2.0 – more and more companies and individuals will have reasons to make Newton their platform of choice. With a significant number of improvements – all derived from customer feedback – Newton 2.0 offers a great solution for keeping professionals organized and helping them communicate, while providing more robust integration with personal computers. At the same time, we created a richer platform and more tools for software development.

The market and customers for mobile devices

On the front lines of the business world, life is

continued on page 26

New Technology

Technical Overview of Newton 2.0: The Developer Perspective

by Christopher Bey, Apple Computer, Inc.

Version 2.0 of the Newton System Software brings many changes to all areas. Some programming interfaces have been extended; others have been completely replaced with new interfaces; and still other interfaces are brand givesnew. This article a brief overview of what is new and what has changed in Newton 2.0, focusing on those programming interfaces that you will be most interested in as a developer.

NewtApp

NewtApp is a new application framework designed to help you build a complete, full-featured Newton application more quickly. The NewtApp framework consists of a collection of protos that are designed to be used together in a layered hierarchy. The NewtApp framework links together soup-based data with the display and editing of that data in an application. For many types of applications, using the NewtApp framework can significantly reduce development time because the protos automatically manage many routine programming tasks. For example, some of the tasks the protos support include filing, finding, routing, scrolling, displaying an overview, and soup management.

The NewtApp framework is not suited for all Newton applications. If your application stores data as individual entries in a soup, displays that

continued on page 29

Newton Technology
JOURNAL

Volume I, Number 5 *November 1995*

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
Joseph Ansanelli, Christopher Bey, Garth Lewis, Julie McKeehan, Neil Rhodes, Bill Worzel

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

John Kawakami • *Editorial Assistant*

Judith Chaplin • *Art Director*

Editor's Note

Letter From the Editor

by Lee DePalma Dorsey, Apple Computer, Inc.

Newton 2.0 – Out of Infancy

Let's cut to the chase here. What exactly is so great about Newton 2.0 and why should I care? The answer is: lots. Lots that we think you as developers will love – and even more that we think Newton platform customers will love. That's probably the most important part about the whole 2.0 launch. We haven't developed Newton 2.0 just because we wanted to. We developed it because it is a powerful technology that solves real customer problems, and that's the reason we're all in business – to solve customer problems and meet unfulfilled needs (and hopefully make a profit doing it!).

So, here we are announcing a major new release of the Newton operating system; but first, we have a little history to get past. We think Newton 2.0 is jam packed with so many improvements, history may just be forgotten. But let's take a minute to look at that history and evaluate exactly why it was important for the platform's evolution.

In 1993, Apple and Sharp launched their first products in the new PDA category – the Apple MessagePad and the Sharp ExpertPad – based on Apple's Newton OS. Technology enthusiasts eagerly bought them up, hoping to realize the promise of a brand new technology. They wanted to be among the first to communicate wirelessly, to enter data via handwriting input, and to organize their lives on a computer small enough to fit into their pockets. The press, too, thrilled at the promise of a revolution in technology, grabbed at the devices and began their evaluation.

Reactions were not exactly what Apple had hoped for. Almost immediately, the press panned the Newton OS based primarily on one feature alone – handwriting recognition. But, a really positive thing, perhaps the most important thing of all, happend in response to those less than stellar reactions. Customers and developers together have been evaluating the technology, realizing its capabilities and potential, and creating real world solutions that harness the platform's power to handle real customer problems in a cost-effective, time-saving way. Some of these solutions have been technology experiments and some have been hard-core, positive ROI products for business professionals and vertical market customers. And they all resulted in some major accomplishments. They got people thinking about the possibilities. They got people doing real things with PDAs. And they helped the Newton Systems team at Apple learn to listen long and hard to customer needs while continuing to refine the technology and improve on it for its next stage in life.

What technology savvy folks realized, at the time the Newton OS was launched, is that new technology is never born fully mature. Like most life forms, technology must also go through many life stages. In infancy, its mere existence is a miracle. It grows into the crawling years when functionality begins to provide some freedom of movement with exploration of anything it can get into. It finally gets to its feet and toddles along until adolescence, when the growth rate is phenomenal. Only then does the technology enter a mature stage that could satisfy the needs of its users. Newton devices, like most life forms, have needed some time to grow up in order to reach their potential. There's no doubt that the technology is incredible and has been since introduction. But the Newton operating system is now well on its way through the toddler years and into adolescence. Its potential is being realized, its

continued on page 10

Ensuring Newton 2.0 Compatibility

by Neil Rhodes & Julie McKeehan, Calliope Enterprises, Inc.

This article provides guidelines to help you ensure that your applications are compatible not only with current Newton devices, but also with future devices that Licensees may create, and with Newton 2.0. Failure to follow these guidelines may cause your application to break in the future.

Undocumented Global Functions

Don't use undocumented global functions. Documented functions will continue to exhibit their documented behavior. Undocumented functions may act differently in the future, or may not exist. Here are some examples of undocumented functions to avoid:

```
CreateAppSoup
    (use the platform file function RegisterCardSoup instead)
SetupCardSoups
    (use the platform file function RegisterCardSoup instead)
MakeSymbol
    (use the documented global function Intern)
GetAllFolders
```

Test by:

Compile your application with NTK 1.6 which will warn about calls to undocumented global functions.

Undocumented Global Variables

Don't use undocumented global variables. Documented global variables will continue to contain their documented value. Undocumented global variables may act differently in the future, or may not exist. Here are some examples of undocumented global variables to avoid:

```
cardSoups
    (use the platform file function RegisterCardSoup instead)
extras
```

Undocumented slots in frames

Only use documented slots. Many protos and other objects use some slots internally, in addition to the ones they document. Only the documented slots are supported; others may be removed or changed. Here are some examples of undocumented slots in objects:

```
cursor.current
paperRoll.dataSoup
GetRoot().dockerChooser
GetRoot().keyboardChicken
```

Undocumented Magic Pointers

Don't use undocumented magic pointers (defined in the platforms file as @ numbers). These magic pointers indirectly point to actual objects in the Newton. If you use undocumented ones, they could point to different objects in future devices or system software.

Soups

Store 0 is internal; all others are unspecified

The only store that you can count on is that `GetStores()[0]` is the internal store. Don't rely on the length of the `GetStores()` array or on the relative positions of stores in that array (other than that the first is the internal store). Future Newton devices might support more than one PCMCIA slot, might support partitioning a large storage card into multiple stores, or might support virtual stores (for example, a remote volume appearing as a store).

Many applications that support the action button contain code that calls `GetStores()` to determine the title of the card action item in the picker. This is OK, but unnecessary. Instead, have your entry for the card action in your routing frame reference `ROM_cardaction`. For instance:

```
myRoutingFrame := {
  print: …,
  …
  card: ROM_cardAction,
}
```

Test by:

Use the List command in NTK to search your project for calls to `GetStores()`. Examine how you use the result. Do you assume that the length will be no more than two? Do you assume that the second entry will be the external store?

Union Soups

Use the RegisterCardSoup/UnregisterCardSoup platform file functions

At one point in time, shortly after the Newton was released, developers were told to add a `RegisterCardSoup` and `UnRegisterCardSoup` slot to their base template with code in `RegisterCardSoup` which called `CreateAppSoup` and `SetupCardSoups`. Later, platform file functions with the same names were created, and developers were told to use those instead. The platform file functions are guaranteed to do the right thing, even on future system software, whereas direct calls to `CreateAppSoup` and `SetupCardSoups` aren't supported.

Test by:

Use the List command in NTK to search your project for `CreateAppSoup` or `SetupCardSoups`.

Or, run your application with the Compatibility app (available in the Llama Lounge area on eWorld) which will print to the Inspector if your application calls `CreateAppSoup` or `SetupCardSoups`.

soupChanged could be called more often

`BroadcastSoupChange` is called for your soup by the system when a store containing your soup is inserted or removed. In the future, it may be called by the system when a soup is created, when a soup is deleted, or in any other cases where a soup is modified.

Test by:

Run your application with the Compatibility app which will call `soupChanged` from within `UnRegisterCardSoup` and each time an entry is added to a soup. Make sure your application works correctly.

Constituent soups don't necessarily exist on all stores

For example, a store which is read-only won't have a soup created on it. In the future, constituent soups might only be created on the default store as needed; if no entries are added on a particular store, the soup wouldn't exist.

Therefore, don't write code like:

```
defaultStore:GetSoup("myUnionSoup"):Add(…);
```

Instead use:

```
GetUnionSoup("myUnionSoup"):AddToDefaultStore(…);
```

The first code snippet may not work since the constituent soup may not yet have been created.

unionSoup:AddToDefaultStore may throw an exception

`AddToDefaultStore` is documented to throw an exception in case of an error; it may start doing so.

Test by:

Run your application with the Compatibility app which will cause `AddToDefaultStore` to throw an exception on a store full error. Check the "Simulate store full" checkbox which simulates a full store (`EntryChange` and `AddToDefaultStore` will simulate store full).

Don't use unionSoup:Add

Although `Add` is documented to work for union soups, don't use it (why would you want an entry to be added to an unspecified store, anyway?).

Test by:

Run your application with the Compatibility app which will print to the inspector if `unionSoup:Add` is called.

View System

Handle any screen size

Make sure to handle screen sizes using `GetAppParams`. The screen could be larger or smaller in one or both dimensions than currently shipping products. For example, licensees might make units which are wider than they are tall. If you have a minimum size and screen is too small, put up a slip stating the case. If you have a maximum size, use it (don't blindly make your app the size of the screen if it doesn't make sense).

Test by:

An application which simulates different screen sizes may be released shortly; test with it.

Don't rely on the order of viewQuitScript messages

The `viewQuitScript` message is sent first to the view which is `Close`d. The order in which descendants receive the `viewQuitScript` is undefined and may change. If you need to guarantee the order in which messages are sent, have your top-level `viewQuitScript` send its own message (maybe `myViewQuitScript`) recursively to descendants.

Test by:

Look at your `viewQuitScript` methods everywhere but in your base template. Do you send messages to your descendants or access slots from your descendants? What about your ancestors?

Strings

Use only documented string functions to manipulate strings

- use `StrLen` to find out the length of a string
- use `StrMunger` to perform operations that change the length of a string.

```
//e.g. append "abc" to the end of a string
      StrMunger(str,StrLen(str),nil,"abc",0,nil);

//e.g. delete 4 character from the end of a string
      StrMunger(str,StrLen(str)-4,nil,"",0,nil);
```

- Do not use the following functions on strings:

`Length` (except to get the size of the binary object which may be unrelated to the length of the string)
`SetLength`
`BinaryMunger`
any of the `StuffXXX` functions

- Do not use undocumented string functions.
- Do not put nulls inside strings.

e.g. `string[10] := $\u00` as a quick way to truncate a string to 10 chars.

Test by:

Run your application in conjunction with the Compatibility application which will print to the Inspector if functions other than those documented for strings are called on a string. Note that the Compatibility application won't catch calls to `Length`.

Use Platform File Functions

When an API is provided, use it.

Use RegFindApps and UnRegFindApps instead of modifying findApps global

The findApps global variable is documented to contain an array of symbols of applications which support global find. However, rather than directly accessing this array, use the platform file functions `RegFindApps` and `UnRegFindApps`.

Test by:

Use the List command from NTK to search for `findApps`.

Don't access the userConfiguration global directly

Instead, use the platform file functions `GetUserConfig`, `SetUserconfig`, `FlushUserConfig`.

Test by:

Use the List command from NTK to search for `userConfiguration`.

Don't send any messages to the IOBox app

Instead, use the `Send` platform file function.

Test by:

Use the List command from NTK to search for `GetRoot().outBox`.

Undo/Redo

From within your undo action, call AddUndoAction

In Newton 1.x, the user interface supports two levels of undo. Newton 2.0 provides an Undo/Redo facility instead. In order to specify the redo action to Newton 2.0, your code needs to call `AddUndoAction` when the user chooses Undo. Thus, you need to call `AddUndoAction` from within the call which is executing an undo action.

On 1.x systems, calls to `AddUndoAction` while an undo action is being executed are ignored; on Newton 2.0, the same call would register a redo action.

For example, here is code handling deleting an item

```
MyDelete := func(item)
begin
  EntryRemoveFromSoup(item);
  AddUndoAction('MyUndoDelete, [item]);
end;

// here is the code to undo the delete
MyUndoDelete := func(item)
begin
  mySoup:AddToDefaultStore(item);

  // Newton 1.x ignores this call
  // Newton 2.0 treats it as a Redo of the delete
  AddUndoAction('MyRedoDelete, [item]);
end;

// here is the code to redo the delete
MyRedoDelete := func(item)
begin
  EntryRemoveFromSoup(item);
  AddUndoAction('MyUndoDelete, [item]);
end;
```

Routing

Use the slip symbols 'printslip, 'mailslip, 'faxslip, 'beamslip only in your routing frame

Don't try to access `GetRoot().printSlip`, for instance. In addition, don't check those symbols from your `SetupRoutingSlip` method.

Test by:

Use the List command from NTK to search for each of the four symbols. Make sure they appear only as the contents of the `routeSlip` slot within your routing frame.

Don't rely on category symbols, 'printcategory, 'faxcategory, 'beamcategory, 'mailcategory

Don't check for these symbols from your SetupRoutingSlip method, for instance.

Test by:

Use the List command from NTK to search for each of the four symbols.

Use only the 'body slot for your data in the fields frame

The body slot should be used for any data you wish to preserve until printing/faxing time. If you are beaming or mailing, the body slot will be overridden with the target.

Test by:

Check your `SetupRoutingSlip` method to see whether you write to anything other than `fields.body`.

Don't read the iobox soup

The format of items in the iobox may change.

NTJ

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

NewtonScript Techniques

Converting Newton 1.x Apps to Newton 2.0

by Neil Rhodes & Julie McKeehan, Calliope Enterprises, Inc.

Introduction

This article tells you how to convert your existing Newton application to take advantage of features of Newton 2.0. (Note that much of this material will not make sense if you're not already familiar with creating an application within the 1.0 Newton operating system.) We'll cover three types of changes you need to make: getting your 1.0 application working in the 2.0 world; redoing your existing 1.0 code to take advantage of enhanced existing features; and adding support for new 2.0 features.

In the section that deals with existing features, we'll cover soup changes first. Newton 2.0 gives you increased control of soup information, as well as offering you the ability to speed up your queries and entry displays. Next, we'll look at Filing and Find changes and then move on to the new way that routing is handled. For most of these topics we'll provide you with source code examples.

In the section that deals with new features, we'll point out the new methods or changes that you might want to implement within your application. As you can imagine, this is a grab bag of topics, and is not covered in any particular order. This section is not a comprehensive treatment of the new features; rather, we cover some of the more important ones. Notice, for example, that one of the most important aspects of the Newton 2.0, NewtApp, is not even discussed. This is because NewtApp is better suited for applications written from scratch.

Making Your 1.0 Application 2.0 Compatible

The most basic upgrades involve both using NTK 1.6 and making minor changes to ensure compatibility. Review the following steps as you upgrade your application.

Check your existing application

The first step you need to take is to make sure that your existing application works correctly under Newton 2.0. If it doesn't, then you should work through the changes described in "Newton Compatibility" elsewhere in this issue.

Build with NTK 1.6

Although it is not necessary to use NTK 1.6 to build Newton 2.0 applications, it is desirable, and since you'll be making changes to your application anyway, you may as well modify it to use NTK 1.6.

Open your NTK 1.0 project within NTK 1.6, then build and download it. Run it through the standard set of paces to verify that the application works.

Use the Newton 2.0 platform file

From the Project Settings dialog, select the Newton 2.0 platform file rather than the MessagePad platform file. Rebuild and download and then verify that your application works.

Remove NTK 1.0 build order

From within the Project Settings dialog, uncheck *NTK 1.0 build order*. By doing this, NTK will no longer add pt_*filename* slots to the base template for user protos. Now, change all of your code that references pt_*filename*. For example, if you have code that looks like this:

```
GetRoot().(kAppSymbol).pt_filename
```

change it to:

```
GetLayout("filename")
```

Now build your application. You will most likely find that the order in which files are built needs to be changed. In such a case, NTK will notify you that the order is wrong with the following error message:

"The file *filename* has not been processed yet"

Note that a file referenced by `GetLayout` must be built before the call to `GetLayout`.

To change the order in which your files are built, you will use the Process Earlier or Process Later menu items. You can do this while viewing the project window in NTK 1.6.

Taking Advantage of Enhanced Newton 2.0 Features

Many 1.0 features have been modified in the new system. Indeed, changes have been made in virtually every feature. Some features have been so completely overhauled that they bear little resemble to their earlier versions; an example of this is routing. Other features have been pepped up or have been given increased capabilities, but operate pretty much as they did in the past; Find is a perfect example of this.

Data storage

Many changes have been made to the way data storage is handled in the Newton 2.0 world. Some of the changes are fairly simple, such as how you register your soups; other changes involve implementing a whole new set of calls, such as the new soup notification routines.

Soup registration

Instead of using RegisterCardSoup and UnRegisterCardSoup, you will use RegUnionSoup and UnRegUnionSoup.

Note that these new routines take soup definition frames that provide information about the soup, its indexes, a user-visible name, and so on.

Soup change notification

In the 1.0 system, soup change notification was fairly rudimentary. You would be notified that a soup had changed, but receive no information

about what had changed. In the Newton 2.0 world the situation is improved, and the granularity for notification is much finer. You can be notified about a wide variety of events, such as: an entry has been added, an entry has been modified, a soup has entered the union soup.

To use this new form of notification you must:

- Call the auto-xmit routines when you change any aspect of a soup
- Register for the new type of notification

You will no longer add your application symbol to the soupNotify array, but instead will call RegSoupChange and UnRegSoupChange. Also, you will use the new Query soup method instead of the old standard Query global function.

Thus, your old query code that looked like this:

```
Query(soup, …)
```

Will now look like this:

```
soup:Query(…)
```

Tags for folders

The slow folder display in the 1.0 system has been fixed with the introduction of tags. Thus, if your application supports filing, it is essential that you use tags to speed up you displays when the user is switching folders.

Add a tag index to your soup

In your soup definition frame, add an index to the indexes array. It should look like this:

```
indexes: […,
  {structure:'slot, path: 'labels, type: 'tags, tags:[]}
]
```

Note that the tags array can be empty. As entries are added to the soup, the array will be updated automatically.

Use tag index when doing your initial query

In your initial query, use the tagSpec slot of the query spec to specify the folder. You also need to remove any code in the validTest that was used to find entries in a folder. Here's a before-and-after example:

```
soup:Query({type: 'index,
    indexPath: …
    validTest: func(e) begin
      return labelsFilter =  '_all or e.labels = labelsFilter
    end,
  });
```

Here is the way your new code should look:

```
if labelsFilter <> '_all then
  myCursor := soup:Query({type: 'index,
    indexPath: …
    tagSpec: [labelsFilter],
  })
else
  myCursor := soup:Query({type: 'index,
    indexPath: …
  });
```

Call query from your FilingChanged method

You will also need to call your new query from within your FilingChanged method to avoid the slow display speed of 1.0. Here is an example of how your new method will look:

```
app.FilingChanged = func()
begin
  if labelsFilter <> '_all then
    myCursor := soup:Query({type: 'index,
      indexPath: …
      tagSpec: [labelsFilter],
    })
  else
    myCursor := soup:Query({type: 'index,
      indexPath: …
    });
  // redisplay
end
```

Modify your tags when the user changes or deletes a folder

You must also update the tags if a folder is deleted or changed. You needn't worry about updating the tags when adding a folder, because the tags are automatically added the first time an entry that contains the new tag is added.

Here is code that handles the updating:

```
app.FolderChanged = func(soupName, oldFolder, newFolder)
begin
  //do nothing if a folder was added
  if oldFolder then begin
    // iterate through updating each entry
    local s := GetUnionSoup(kSoupName);
    local cursor := s:Query({type: 'index,
      tagSpec: [oldFolder]});
    local e := cursor:Entry();
    while e do begin
      e.labels := newFolder;
      EntryChangeXMit(e, kAppSymbol);
      e := cursor:Next();
    end;

    // update tags
    if newFolder = nil then // a folder was deleted
      s:RemoveTags([oldFolder])
    else    // a folder was modified
      s:ModifyTag(oldFolder, newFolder);
  end;
end
```

Entry Aliases

In the old 1.0 world, you had to go through a bit of contortion when you wanted to uniquely identify a soup entry. To create such a reference you had to store a combination of entry unique ID, soup ID, and store ID. In Newton 2.0, things are much simpler. Now, you create an entry alias. You can then use it to uniquely reference an entry and to retrieve that corresponding entry.

The new calls that accomplish this are MakeEntryAlias and ResolveEntryAlias. You will use these anywhere you need to save references to entries. For example, to save the currently-displayed entry, your code should look like this:

```
app.viewQuitScript := func()
begin
  local currentEntryAlias := MakeEntryAlias(myCursor:Entry();
  // save currentEntryAlias in with other application preferences
  …
);
```

Now, to open the application at the last-displayed entry, all you have to do is this:

```
app.viewSetupFormScript := func()
begin
  …
  local currentEntryAlias;
  // initialize currentEntryAlias from savedpreferences
```

```
  local currentEntry := ResolveEntryAlias(currentEntryAlias);
  if currentEntry then
    myCursor:Goto(currentEntry);
  …
end;
```

New Query Specs

You will no longer be using startKey and endTest in your query specs. Instead, you will use beginKey or beginExclKey as a replacement for startKey. Likewise, you will use endKey or endExclKey instead of an endTest.

There is also the nice little addition of being able to use an indexValidTest instead of a validTest if your validity testing depends only on the value of the slot being indexed. This will speed cursor operations up (because only the index value is used, the entry itself doesn't need to be read to determine validity).

Multi-Slot Indexes

In the Newton 1.x, you could index on only one slot. This is a thing of the past in Newton 2.0, which provides multi-slot indexes. Here's an example of an index that sorts first by last name and then by first name:

```
soup:AddIndex(
  {structure: 'multiSlot,
  path:['lastName, 'firstName],
  type:['string, 'string]
  });
```

Here's an example of a query that will use our new multi-slot index:

```
soup:Query({type: 'index, indexPath: ['lastName, 'firstName]});
```

There are some other features that you can set up in your indexes as well. These include:

- Specifying the sort order (ascending or descending) of keys
- Counting the number of entries in a cursor (cursor:CountEntries)
- Resetting to the end of the cursor (cursor:ResetToEnd)
- Having string indexes be case and/or diacritical sensitive

Find and Filing

Many of the changes that have been made to Find and Filing in Newton 2.0 are similar, so they will be covered together. Some changes are simple, such as how you register with the system; other changes involve the addition of new methods.

Find registration

Use the platform file functions RegFindApp and UnRegFindApp instead of manipulating the findApps array directly (these platform file functions work in 1.x as well as Newton 2.0).

Filing registration

Instead of adding to the soupNotify array, use RegFolderChanged and UnRegFolderChanged.

Find date equal

Within Newton 2.0 there is a user interface for finding an entry with a particular date. As a result, you need to support finding particular dates. Here is sample code for DateFind that supports finding dates before, dates equal, and dates after (it assumes entries are indexed on the `timestamp` slot):

```
app.DateFind :=  func(findTime, findType, results, scope,
statusForm)
begin
  constant kMinutesPerDay := 1440;

  if statusForm then
    statusForm:SetStatus("Searching in" &&
      kAppName & $\u2026);

  local theSoup := GetUnionSoup("Notes");
  if theSoup then begin
    local queryFrame := {
      type: 'index,
      indexPath: 'timestamp,
    };
    if findType = 'dateBefore then
      queryFrame.endExclKey := findTime;
    else if findType = 'dateOn then begin
      queryFrame.beginKey := findTime;
      queryFrame.endExclKey := findTime + kMinutesPerDay;
    end
    else // dateAfter
      queryFrame.beginExclKey := findTime;
    local theCursor;

    theCursor := theSoup:Query(queryFrame);
    if theCursor:Entry() then
      AddArraySlot(results,
        {_proto :soupFinder,
          owner: GetRoot().(kAppSymbol),
          title: kAppName,
          cursor: theCursor,
          findType: findType,
          findTime: findTime,
        });
  end;
end
```

Filter by store

The folder tab can now filter not just by folder but also by store. If a storesFilter slot exists in the parent inheritance chain of the folder tab, then stores as well as folders are shown in the folder tab. The value of the storesFilter slot is nil if all stores should be displayed. Otherwise, the slot is set to a store object. When the user changes a store in the folder tab, the FilingChanged method will be called.

Note that your `Query` calls should take into account the current `storesFilter` (just as it takes into account the value of the current `labelsFilter`).

Card Routing

The Newton 2.0 user interface has changed the location of card routing. Moving to and from cards is now in the folder slip rather than in the action picker. By default, this feature is disabled in the new system.

If you support card routing in your application, you should remove the corresponding card slot from the routing frame. Next, you should add a doCardRouting slot (with a value of true) to your application base template. If your application doesn't support folders, but does support card routing, you will also need to add a protoFolderButton to your status bar and set the value of doCardRouting in your base template to 'onlyCardRouting.

Routing

Many of the changes in the new system make routing much easier to implement. Changes have been made in both the registration process and in how various routing actions are implemented. Thus, you will need to add new code as well as clean up your old code.

Remove the old 1.x routing registration

An important first step is to fix the way in which registration is handled. Start by removing the code in your InstallScript that installs a routing frame in the routing global and that calls BuildContext to add a view to the root view.

Likewise, get rid of the code in your RemoveScript that removes the routing frame from the routing global and the corresponding view from the root view.

Adding route actions

To support actions (like duplicate and delete), add a routeScript slot to your base template (actually, from any ancestor of the protoActionButton). The value of this slot is an array of frames. Each frame contains the following slots:

```
title:
routeScript: func(target, targetView)
icon:
```

Using Stationery For Routing

In Newton 2.0, routing is handled by the use of view stationery. Thus, the items that appear in the routing button depend entirely on the type of the data that is currently being viewed (the `class` slot of the current target). Thus, within this new system you need to handle some new tasks: setting the target's class slot and stationery registration.

Setting the class of target

First, you should make sure that the target has a class slot that specifies the kind of data in this item. For instance, a check-writing application might have checks with a class slot of '|check:MySignature| and deposits with a class slot of '|deposit:MySignature|. It is easiest to set this class slot when entries are created.

Stationery registration

To handle stationery registration you will be installing one data def for each kind of data. For example, in a checkbook application, you would install one data def for '|check:MySignature| and another one for '|deposit:MySignature|. Typically you will do this installation in your InstallScript and remove each data def in your RemoveScript.

Here's an example of registering:

```
RegDataDef(kCheckDataDefSymbol, {
  _proto: newtStationery,
  symbol: kCheckDataDefSymbol,
  superSymbol: kAppSymbol,
  name: "Check",
  version: 1
});
```

Handling routing

You'll have to do some further conversion to handle print formats in your project. You will create a print format data view by adding a title slot (the same title that appears in the format picker) and a symbol slot (containing a symbol unique to your data view) to your templates that proto from `protoPrintFormat`.

If your application supports sending frames (like Beam and Mail), then you will also need to create a targetFrameFormat data view:

```
{
  _proto: targetFrameFormat,
  title: "my title",
  symbol: '|target::check:MySignature|, //unique for the data view
}
```

If your application supports a text representation (like Mail), then you need to add a textScript slot to your data def which will be used to create text from the target.

Handling the In/Out box

In order for your application to display your data in the In/Out box, you should create a data view. To do this you need to do several things:

1. Create a layout file with templates that can display your data (note that methods in these templates can reference `target`, an inherited slot which will contain the target frame).
2. In the topmost template add the following slots to make the template a viewer data view:

```
title: "Check Viewer",
symbol: kCheckViewerDataViewSymbol,// unique for data view
type: 'viewer,
```

3. Install all your data views in your InstallScript with calls to RegDataView:

```
RegDataView('|check:MySignature|, myDataView);
```

4. Unregister your data views in your RemoveScript with calls to UnRegDataView:

```
UnRegDataView('|check:MySignature|,
  kCheckViewerDataViewSymbol);
```

Printing Multiple Items Using Cursors

A new feature of Newton 2.0 allows users to handle routing tasks from the overview. For instance, they can select multiple items and print or fax them.

Note that, by default, each entry in the cursor is added separately to the Out Box. For view-type formats (printing/faxing) there is also a way to have just the entry in the Out Box (for instance, if you want to print multiple items on one page). See the Newton Programmer's Guide for more details.

Adding Newton 2.0 Features to Your Application

There are some important Newton 2.0 features that you should add to your application, even if you don't take advantage of all the new possibilities. You might think of this as the "must have" list. These features are:

- Supporting screen rotation
- Rich text

Another feature that you should consider adding depends on the type of application you have. If it makes sense for your application, you may also want to add support for being a Backdrop app (Backdrop apps are discussed later in this section).

Handling Screen Rotation

In order for your application to run while the screen is rotated, it must have a ReOrientToScreen method in its base view. This method gets called after the screen is rotated.

Note that if your app view uses view justification (or if it calls GetAppParams from the viewSetupFormScript), changes to the screen size can be handled by a call to SyncView. Therefore, the ReOrientToScreen method can often be as simple as:

```
app.ReOrientToScreen := func()
  :SyncView();
end;
```

You will also need to consider the layout of your views when your application is in a rotated state. Obviously, this will vary from application to application. Some applications will need to change how their views are laid out extensively, while others will adapt to the new orientation without much tweaking.

Rich Text

Rich text is new to Newton 2.0. Rich text is text that is not recognized but remains in ink form. The format of rich text is like a string, but with ink words embedded (the kInkChar character specifies that a particular character is actually an ink word).

Note that these ink words are stored at the end of the string. Thus, there may be data in a string after the end of the last character.

Comparing non-rich and rich text

Your application should support rich text everywhere it makes sense. Don't skimp here: the more places your application offers rich text, the more useful it will be to the user. For example, in a check-writing application, you might need to recognize only the amount (to keep track of a running total) and the check number (to check for duplicates and to sort by check number). Since other fields need not be recognized (payee and memo, for example), they should allow rich text. Another example is the built-in Names application. It allows rich text for the last name but asks for a letter to use for sorting.

protoRichInputLine and protoRichLabelInputLine

These protos work much like protoInputLine and protoLabelInputLine. To obtain the rich text in the proto, use the method GetRichString(). Note that you should not read directly from the text slot.

Allowing rich text in other views

By default, paragraph views and edit views don't support rich text. In order to enable rich text, add a recConfig slot to the view. A predefined configuration frame (suitable for recConfig) that will allow text or ink is ROM_rcInkOrText. To obtain the rich text, use MakeRichString, passing the value of the text slot and the value of the styles slot.

Becoming a Backdrop App

In order for an application to become the Backdrop application, it should:

- be full-screen (otherwise, there will be blank parts on the screen outside of the application).
- be able to rotate to "landscape mode" – although this is not an absolute requirement.
- be prepared to never close. For example, it would be important when the user changes some data to call EntryChange by means of a watchdog timer rather than waiting until the user closes the application or scrolls to another piece of data (the user may leave this data open for hours or days!).

Newton 2.0 Topics Not Covered

There are a number of other features that we are not covering here but you should consider adding to your application. These features include:

- Extending your app with stationery
- DILs
- NCK
- Communications
- Extending Built-in apps using stationery

Summary

In this article we covered the three different types of changes you need to make to an application to bring it into the Newton 2.0 world. The easiest and most crucial changes simply involve ensuring that it will run "as is" on a Newton 2.0 machine. Next, we discussed some of the modifications to existing Newton features that you will want to take advantage of in your code. As many of the changes involve significant speed increases for the user, they are well worth your time and attention. Last of all we covered some of the new features that you will want to add to your application.

If you implement everything that we discussed here, you will be well on your way to having a good transition application from the 1.x to the Newton 2.0 world.

NTJ

continued from page 2

Newton 2.0 – Out of Infancy

functionality is growing and maturing, and customers are employing it in ways that will further enhance its evolution, and its business promise over time.

We're thrilled to be able to bring you all of the latest news on the Newton 2.0 OS in this issue of the Newton Technology Journal. Our cover stories will introduce you to all of the new features built into Newton 2.0, from a marketing and business perspective as well as a technical one. We've covered the critical issues around moving an application from 1.x to 2.0 and gaining compatibility with your existing 1.x applications. We've also included an article on the new communications features in Newton 2.0. Future issues of this publication will cover additional areas of 2.0, including programming in-depth, and will run features on whatever we find to be development trouble spots – although we hope there will be none!

So, with this issue of NTJ and the recent Newton Development Conference, we are proud to bring you the Newton 2.0 operating system. We've worked long and hard on it and we're proud of the growth and changes our infant has gone through. We listened to customers and developer feedback, and we've delivered on the next generation of the platform. It is probably not yet a fully grown adult, but an adolescent full of power, capabilities, flexibility, and the agility of youth. With this generation of the Newton OS, users will more easily and more powerfully organize data, communicate, and integrate with their desktop PCs. And it's your Newton 2.0 applications that will help them do it. The Newton platform has graduated from high school. We'll look forward to seeing the new solutions that harness its abilities and help launch it into a world of possibilities for mobile professionals and the vertical markets.

Newton 2.0 Communications Overview: Peeling the Onion

by Bill Worzel, Arroyo Software, ArroyoSeco@eworld.com

Introduction

Newton 2.0 is designed with communications integrated throughout the system software. In any Newton application the user will use similar interfaces to dend information regardless of the medium he or she is using to send the information. The general idea is, whatever you can see, you can send.

From a programming point of view, presenting a unified communication interface appears complicated. Fortunately, the Newton's communications architecture is designed in a layered way, and there is a great deal of built-in communications software that we can use, so that, unless your requirements are fairly unusual, little new programming needs to be done.

Figure 1 shows the different layers in the Newton 2.0 system. Most of the layers are in the NewtonScript level, but the communication tools are written in C++. Within the NewtonScript layer there are five main pieces to the system which are accessed from four separate APIs. We will look at these layers one by one.

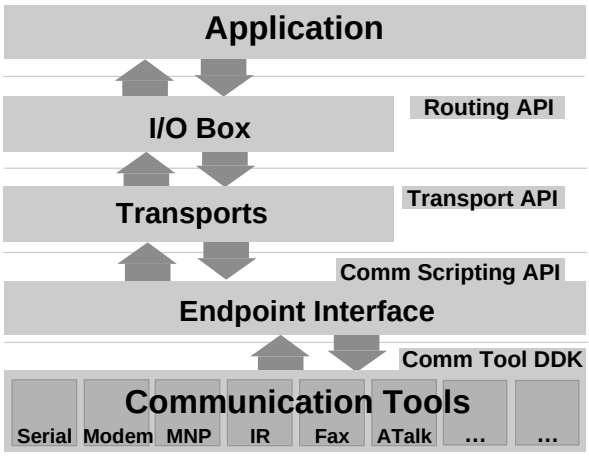

Figure 1: Newton Communications Layers

Routing

Newton 2.0 communications are built around a store-and-forward model, with target data stored in the In/Out Boxes. Store-and-forward describes how messages are routed (directed) to a distant communications object – or from such an object to a Newton application – through an intermediate holding area (the In/Out Boxes). Target data is any piece of information that is routed in or out of the Newton.

The In/Out Boxes are a single application that provides a user interface to view and manage messages, which are stored internally as a soup. Figure 2 shows what the In/Out Boxes might look like when there are messages pending. Note that at any time the user can switch between In Box and Out Box with the radio buttons at the top of the view.

Figure 2

Routing is usually triggered by adding the protoActionButton to a view. The protoActionButton, when tapped, displays a picker showing available actions (transports and routeScripts.) as illustrated in Figure 3. Some applications will have one protoActionButton in the status bar; others will have one in each of several views. The Names application, for example, has a single Action button, since normally only one name at a time is viewed. The Notes application has an Action button attached to each note, since there may be many notes on the screen at any given time.

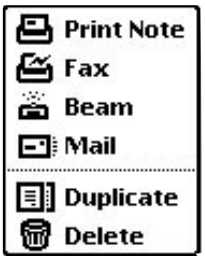

Figure 3

Each target object that is routed must have a class slot identifying the type of data associated with this kind of object. Normally each application will supply its own class of data for routing, such as 'Note or 'Form or the like. This class is used by the system to look up (in the View Definition Registry) the list of routing formats that may be used to route the data. From these routing formats, the system creates a list of communications methods (faxing, printing, beaming, etc.) that can route the target data. (These communications methods, or transports, are discussed in more detail in the following section.) The net result is that when the user taps the Action button

a list of destinations appear which are “possible” for the target data.

Figure 4 shows how this is stitched together. An application, usually in the InstallScript, will install into the View Definition Registry one or more formats named for the classes of data it will route. These formats will consist of one or more dataTypes that describe what form the target data can take when routed. The system uses this to search a list of installed transports; when it finds a transport that supports one of the dataTypes, it adds the transport name to the list to be displayed in the Action Button.

In Figure 4, the application has installed into the View Definition Registry a frame, |forms:PIEDTS|, which supports two routingTypes of 'views and two of 'frames. At least one of the types is supported by each of the built-in transports for printing, faxing, beaming, and mailing, so these options appear in the Action button. Note that it doesn't pick up the transport whose dataType is 'binary.

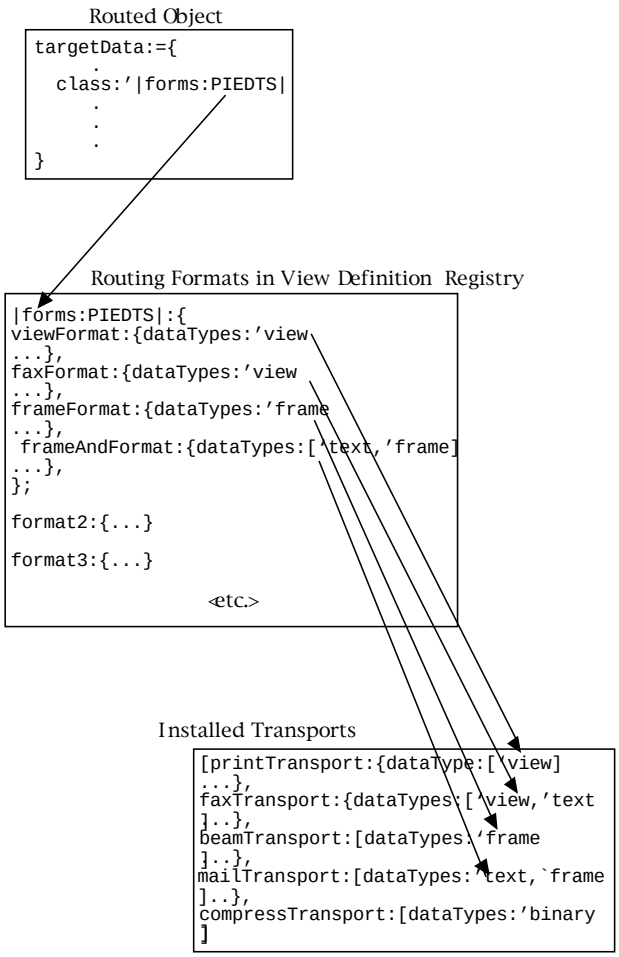

Figure 4

When the user selects a transport from the Action button, a routing slip is displayed and all formats in which the data can be displayed appear in the format picker, as shown in Figure 5. Formats are View Definitions that describe how the target data should be organized before sending it to the appropriate destination. When printing, for example, there may be several formats – letter, memo, two-column, etc. – that describe how the target data will be printed or faxed.

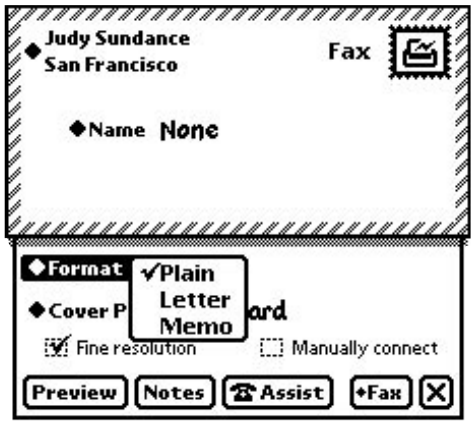

Figure 5

When the user selects a format for the target data and sends it off, the appropriate transport is then messaged with information about the target data, and it is placed the target data the Out Box for further disposition.

TRANSPORTS

The simplest definition of a transport is “something that routes your data.” But a clearer definition is that a transport is a globally available service offered to applications for sending or receiving data. Because of the global nature of transports, it is not necessary, or even likely, for an individual application to define a transport.

The built-in transports include printing, faxing, mailing, and beaming, but one might imagine additional transports such as messaging (point-to-point, real-time message passing), scanning, and even compressing, or archiving data. Thus, while transports are usually associated with hardware (printers, mail servers, and scanners, for example) they are not limited to hardware, and a service may be offered that alters the data being routed without sending it to any outside hardware.

Transports are built as “auto-load” packages; which means that they do not appear in the Extras Drawer. Instead, a transport's InstallScript registers the transport with the system by calling the function RegTransport. As described in the routing section above, if appropriate target data is routed, the transport may then appear in the Action Button picker in an application when the user taps the button.

Because most transports have communications code that will be used to send or receive the target data, they will typically have communication endpoint code that communicates with the destination.

Transports usually work with an application through the In/Out Box application. Figure 6 shows the interactions between the NewtonScript application, the In/Out Box, and a transport during a send request.

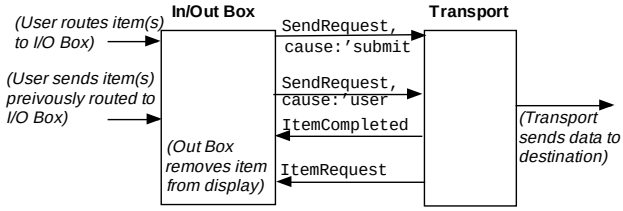

Figure 6

In particular, a transport receives a SendRequest whenever an item is routed to the Out Box. The request sent to the transport has a cause slot that describes why the transport was notified. In the case of target data that is held in the Out Box, the cause slot will have a value of 'submit, indicating that it is posted but should not be sent yet. If the cause is 'user or 'item, then the transport should send the target data.

In either case, after processing a SendRequest message, the transport should call ItemRequest to see if there are any further items posted in the Out Box.

A request to receive data, is a little more complicated. In the simplest case, when the user selects a transport from the Receive button list in the In Box, the selected transport is sent a ReceiveRequest message. The transport will connect to the remote source, get any pending data, and add it to the In Box list.

In a slightly more complex situation, the transport may simply get a description of what is available at the source (for example, the title of an email message) and post it to the In Box. In this case, the transport must add a remote slot to the request and set it to true. This flags the item so that the In Box knows that the body of the data has not arrived. The user may later select the item from the In Box and ask to see it, at which point the In Box will send the transport another ReceiveRequest message, but with the cause slot set to 'remote. The transport will then be responsible for getting the body of the data so the In Box can display it.

As with sending a message, the transport should then check for other pending receive requests by calling ItemRequest.

Note that ItemRequest is defined in protoTransport, so the transport sends itself a message for the purpose of getting information from the system – in this case, the next item to be transported.

As mentioned above, the main proto used to create a transport is protoTransport. protoTransport is powerful and in many cases surprisingly little code other than the actual endpoint code needs to be written. This is because the defaults typically "do the right thing" to provide an interface and default behavior for the transport.

In particular, tasks such as displaying the status of a routing request, logging of routed items, error handling, power-off handling, and general user interfaces are handled well by the defaults if the transport simply sets or updates a few slots when appropriate. Only the actual service code (such as communications) will be different between transports.

The key methods and variables that are usually implemented in a transport are as follows:

```
status                  // used to display the status of a routing
                        // request sent to a transport, set using the
                        // method: SetStatusDialog
actionTitle             // title that appears in the transport's
                        // routing slip
appSymbol               // the transport's symbol, used to match the
                        // transport with the item being transported
    icon                // icon that appears in the transport's
                        // routing slip

    SendRequest         // message sent to the transport from
                        // the In/Out Box data when item posted
                        // or requested to send out
    ReceiveRequest      // message sent from the In/Out box to
                        // receive or to complete receipt of
                        // items previously received in summary
                        // form
    CancelRequest       // method to stop transport of data
    NewItem             // message from the In/Out Box to create an
                        // item for display in the In/Out Box, usually
                        // overriden by a transport to add transport
                        // specific information about an item
    ItemRequest         // message sent by the transport to self
                        // to request next item pending in a
                        // send/receive action
    ItemCompleted       //message sent by the transport to
                        // self to notify the system item
                        // transport action (send/receive) has
                        // finished
```

In addition to these methods and slots, there are many other features that may be added to a transport to augment or override user interface features controlling the display and format of data being transported or the action of the transport.

The DTS sample ArchiveTransport shows a minimally implemented transport that provides a global, frame-based transport for archiving frame information in an "archive soup."

ENDPOINTS

Endpoints are the primary API for programming communications on the Newton in NewtonScript. They provide a "virtual pipeline" for all communications. Endpoints are designed to hide the specifics of a particular communications media as much as possible, and once connected, present a generic byte-stream model for input and output.

Endpoint code to receive data from an AppleTalk network can be identical to code to receive data through a modem, which can be identical to code to receive data over a serial line, etc. Things such as packetization – which occurs in any network protocol – are hidden from the endpoint user during sending and receiving, as are flow control, error recovery, etc.

The only exceptions to this rule occur when there are specific hardware limitations that cannot be hidden by the endpoint API. For example, IR beaming is a half-duplex protocol (that is, it can send or receive, but not both at the same time) while serial, AppleTalk, or modem communications are all full-duplex (that is, they can send and receive at the same time).

Of course, while sending and receiving are media-independent, the connection process is necessarily tied to the media being used. So, for example, with AppleTalk it is necessary to specify network addresses; for modem communications, a phone number; for serial communications, speed, parity, stop bits; and so on.

Figure 7 shows the life cycle of an endpoint. An endpoint is initially defined as a frame in an application that is based on the protoBasicEndpoint proto. This has several slots describing the settings of the endpoint and methods that may be called by the system during the course of its existence. However, this frame is not an endpoint. That is, it describes what an endpoint might look like, but it is not a NewtonScript object. To create such an object it must first be instantiated. Note that since the objects used most often in the Newton OS are views, and since the view system automatically instantiates the view object when it is opened, we usually don't see much instantiation code in apps. But with an endpoint, because it is independent of the view system, we must explicitly instantiate it to create an endpoint object.

Life Cycle of an Endpoint

```
EP:Instantiate()
EP:Bind()
EP:Connect()
EP:Output.()/SetInputSpec()
EP:Disconnect()
EP:Unbind()
EP:Dispose()
```

Note: EP is a fictitous reference to a NewtonScript frame which is based on protoBasicEndpoint

Figure 7

Once the endpoint is connected, it should be bound to a particular address, node, etc., depending on the media. An AppleTalk endpoint, for example, is bound to a node on the network. This is done by sending the endpoint the Bind() message.

After binding the endpoint, the Connect message is sent to connect to the particular media being used. For a remote service that is accessed through a modem endpoint, the endpoint would dial the service and establish the physical connection (but not protocol items such as logging on, supplying passwords, etc.; these are part of an ongoing dialog that the application and the service must engage in once connection is established).

The endpoint method Listen() may be used to establish a connection instead of the Connect() method. In this case the endpoint is connected and ready to listen to an "offer" by the communications media. Based on the the particular situation with the remote media, an application may either reject the connection by sending the Disconnect() message to the endpoint, or accept it with the Accept() message.

After connecting, the endpoint is ready to send and receive data. Sending is fairly straightforward and is done by using the methods Output() and OutputFrame().. The latter method is used to send Newton frames to other entities that understand this data format, the former is used to send all other kinds of data. Such calls may be made either synchronously or asynchronously.

Receiving data is a little more complex. Incoming data is buffered by the system below the application endpoint level. An application must set up a description of situations to trigger processing of incoming data. This description is in the form of an inputSpec. For example, an inputSpec could be created which looked for the string "login:", or it could be set to trigger when 200 characters were received, or after 100 milliseconds. To some extent it can be set to notify the endpoint of incoming data after a combination of these events (e.g., after the string "login:" is seen or after 100 milliseconds, whichever comes first).

When an inputSpec input condition is met, an appropriate message is sent to the endpoint. This message differs depending on the cause of the trigger; for example, a PartialScript message will be sent if the condition causing the event is a 100-millisecond wait, while an InputScript message will be sent if a specified string has been received or character limit reached.

At any given time, the endpoint will have only one inputSpec active. By default, the active inputSpec will remain active until told otherwise. InputSpecs are activated by sending the message SetInputSpec() to the endpoint with a reference to an inputSpec frame.

By chaining inputSpecs together, a communications state machine of arbitrary complexity can be created. For example, before connecting to a service, an application might call SetInputSpec() with an inputSpec that looks for the string "login:". Once that string is seen, the InputSpec() method within the inputSpec might call InputScript () to activate an inputSpec that looks for the string "password:". When this string arrives, the InputScript code might call SetInputSpec to activate an inputSpec that triggers every 100 characters or when a carriage return character is received. In this way, endpoints can be used to build a communications protocol based on expected behavior of the service.

Endpoints deal with different forms of data well. Since the Newton uses Unicode (16-bit) character representations, and since most systems still use ASCII, character strings sent or received through endpoints will do a Unicode-to-ASCII and ASCII-to-Unicode translation by default. This default may be overidden by adding an encoding slot to the endpoint with a system constant describing a translation table to be used for all character data.

At a slightly higher level a data form may be used to convert incoming or outgoing data as appropriate. Figure 8 shows a list of the data forms that can be used to format incoming or outgoing data. These forms control how the data will be treated with, for example, 'char and 'string forms going through translation table mappings, with 'number being interpreted as a Newton 30-bit integer.

Data Form	Description
`'char`	Default value for sending characters, does ASCII-to-Unicode or other translations based on encoding slot.
`'number`	Converts to or from 30-bit integer.
`'string`	Default for sending and receiving strings, Unicode-to-ASCII conversion is done unless overridden by the encoding slot. Termination character is added to the string.
`'bytes`	A stream of single-byte values. No translation is done on the values.
`'binary`	Used to receive or send raw binary data.
`'template`	Used to exchange data with a service that expects C-type structures.
`'frame`	The data is expected to be a frame. For output, the frame is flattened into a stream of bytes prior to being sent; for input, the byte stream is unflattened and returned as a frame. Particularly useful when beaming or communicating with a desktop machine using DILs (DILs are descibed below).

Figure 8

Perhaps the most intriguing of these data forms is the 'template form, which can be used to map a series of data values into different formats in the same way a C-structure describes how to read successive values in memory. This is also used to describe endpoint options – the state settings for the endpoint – when creating or modifying the endpoint object.

When an application is done sending and receiving data and wishes to tear down the endpoint, there are messages that may be sent to an endpoint to break down the state built up during the process of connecting and communicationg with the media.

The first step is to terminate any outstanding inputSpecs by sending a Cancel message followed by a SetInputSpec() message with a nil inputSpec. This terminates the current inputSpec and establishes the fact that no more input will be accepted.

Then, the connection is broken by calling the Disconnect() method.

Next, the Unbind() message is sent to break the address association between the endpoint and the media.

Finally, a Dispose() message is sent to destroy the endpoint object. Note that the endpoint may be left instantiated or disconnected if it is anticipated that it may be reconnected later in the life of the application.

Low-Level Communications Tools

Below the NewtonScript level there are built-in system tools that provide basic communications functionality. When an endpoint is instantiated, one of the things that must be defined is the type of the endpoint. This definition causes the endpoint to be connected to one of the existing low-

level tools that are written in C++ and run in a separate task thread.

While the details of these tools are beyond the scope of this article, at some point Apple will release the necessary programming interfaces and tools to support the development of third-party communications tools.

DILs

Not shown in the diagram in Figure 1, but still important to Newton communications programming, are a set of libraries called the Desktop Integration Libraries, or DILs. The first and most important thing about DILs is that they don't run on the Newton. The DILs are libraries for Macintosh and Windows development environments, and are used to create apps which can establish a communications link between Newtons and desktop machines.

Figure 9 shows this relationship. Essentially, endpoint code on the Newton, whether hooked directly to an application or via a transport, sends data from the Newton to a desktop machine. On the desktop machine, an application that uses the DILs sends and receives data which is then displayed on the desktop machine. Currently, DILs are available for Mac OS and Windows platforms.

All of the DILs are libraries written for C. On the Mac OS platform, there are MPW, Think C, and MetroWerks libraries. On the Windows platform, DILs are implemented as a DLL and therefore should be independent of particular C language implementations.

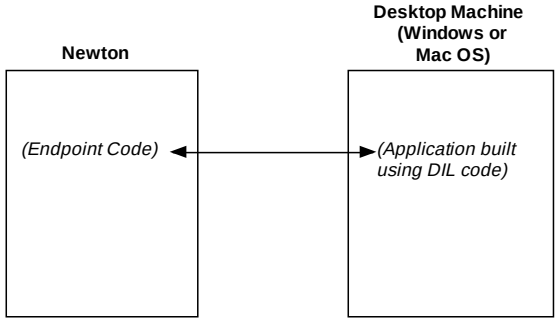

Figure 9

The DILs' main feature is that they abstract the connection to the Newton to a virtual pipe for bytes, and they provide control over things such as ASCII-to-Unicode conversions and Newton data structures and types such as frames and 30-bit integers.

As shown in Figure 10, there are three DILs which build off of one another: CDILs, FDILs, and PDILs. CDILs provides basic connectivity to a Newton. To use FDILs and PDILs, you must use CDILs to establish a connection. FDILs provide a relatively simple way to map NewtonScript frames to C structures and also provide a mechanism to handle data that was added dynamically to the frame. PDILs provide an easy mechanism for synchronizing data between a Newton application and a desktop application. As I write this, PDILs are not yet available, but will be available in the future.

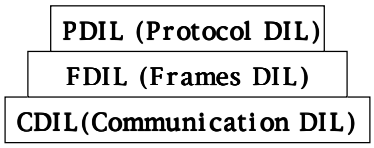

Figure 10

CDILs

CDILs essentially have the following phases: initialization, connection, reading or writing, disconnecting (sound familiar?). The idea is to create and open a virtual pipe to the Newton and then communicate using some predetermined protocol by sending and receiving messages or data down the pipe.

The CDInitCDIL function must be called before anything can be done with CDILs. On Windows machines, the routine CDSetAppplication must be called next. There is no equivalent call on the Macintosh. Next, the routine CDCreateCDILObject() is called to create a CDIL pipe. CDCreateCDILObject returns a pointer to a pipe; this pointer must be used for all subsequent calls involving that pipe.

CDPipeInit initializes a pipe object so that it is "open for business." In particular, it defines the communications options, including the media details such as connection type (serial, AppleTalk, etc.) and relevant media options (speed of connection, dataBits, modem type, etc.).

Next, the pipe waits for the connection from the Newton using the function CDPipeListen. When the Newton contacts the desktop machine, the application using the CDIL may accept the connection by calling CDPipeAccept. At any time in this process the desktop application can cancel an attempted connection by calling CDPipeAbort.

Once a connection is established and working, data can be sent and received using the routines CDPipeRead and CDPipeWrite. As with most CDIL routines, these calls may be made either synchronously or asynchronously with a callback routine. (Mac OS programmer note: callbacks are not executed at interrupt time, so they are not subject to the restrictions placed on code running at interrupt time.)

From this point on, the desktop application and the Newton application will probably engage in an application-specific protocol where there will be a predictable exchange of messages and data via the CDIL's virtual pipeline.

When the decision is made to terminate the connection, the routine CDPipeDisconnect may be called. Once this function has completed, the connection has been broken and both sides must reestablish the connection before any more data can be sent or received.

Finally, when the desktop application is completely finished with the pipe, it must call the functions CDDisposeDILObject to tear down the pipe and CDDisposeCDIL to clear the CDIL environment.

FDIL

FDIL, or Frame Desktop Integration Library (also called HLFDIL for High Level FDIL), is used to support transferring NewtonScript objects to and from the desktop in an orderly fashion. A CDIL connection must be established, and is used to transfer frames.

Before FDIL calls can be made to move information to or from the Newton, the FDIL routine FDInitFDIL() must be called to initialize the library.

The simplest use of an FDIL is to map NewtonScript frames into C variables. If the frame shown in Figure 11 is going to be uploaded to a desktop machine, the desktop application can use FDILs to map this frame into the C structure shown in the figure.

```
aFrame:={ slot1:'b,
     slot2: {   slot1:24,
                slot2:{slot1:16,
                       slot2:$c}
            }
     slot3: "TROUT"}

struct {
   char slot1[5]; // whatever symbol length
   struct slot2 {
```

```
    long subslot1;
    char subslot2;
  };
  char slot3[32]; // whatever max strlen
}
```

Figure 11

To build the mapping between the NewtonScript frame and the C structure, make repeated calls to FDbindSlot. These calls match a Newton slot name (or an array object) to a C variable or buffer. Part of the call is a maximum size to ensure that the capture of the NewtonScript object does not overflow the memory reserved for the C variable.

In this example, there would be repeated calls to FDbindSlot, each of which would specify a slot name for an element in the frame and the address of the C variable or – in this case – structure member. Once this is done, the data can be transferred by calling FDget. When the Newton sends the frame data (presumably by calling OutputFrame) to the desktop, the FDIL will move the data into the appropriate locations on the desktop machine.

If data were to be sent to the Newton, the desktop application would call FDput to send the data at the addresses specified by the FDbindSlot calls to the Newton in a flattened frame format that the Newton can understand. In this case, on the Newton side it would be expected that an inputSpec would have been established which expected a data form of 'frame.

This is the easiest and most efficient way to move data to and from the Newton, but since NewtonScript frames can change dynamically, there needs to be a way to get information from the Newton which the desktop application may not have known about when the initial binding took place, or new information created after binding. This is done by transferring data to the desktop machine into a dynamic tree structure that can be parsed by the desktop machine. This data is called unbound data.

As before, a CDIL connection must be established; then, a call to FDget will transfer any data. If any of the data transferred is bound, it will be put in the appropriate location. Otherwise, it will go into dynamically allocated memory.

To get the unbound data, the routine FDGetUnboundList returns a pointer to a tree. This tree is organized with a branch for each element in the structure on the Newton. The list structure is an array of elements, each of which in turn may be a list of elements if the particular branch has sub-elements. For example, if the Newton frame shown in Figure 11 were brought into the desktop machine in an unbound form, there would be three branches (one for each element). Branch two would have two sub-branches, and branch two of branch two would have two branches.

Using this structure, you can write code to parse and use the unbound data. After the desktop application is done with the unbound data, it should be disposed of by calling FDGetUnboundList.

Examples of CDIL and FDIL usage can be seen in the DTS sample SoupDrink, which transfers the contents of any soup on the Newton to the desktop or sends frames from the desktop to the Newton.

NTJ

Newton Communications – A Point of View

by Eileen Tso, Apple Computer, Inc.

As the Communications Evangelist for the Newton platform, whenever I discuss the 2.0 operating system and its improvements beyond the 1.x communications story, I am proud to hear myself tout that our operating system uses an extensible architecture which allows for advanced communications capabilities and features.

I hope you'll notice that we have moved toward a truly modular system. Our architecture is "layered," and provides access to built-in tools that now exist for you. The I/O box, transports, and endpoints of 2.0 are just a glimpse of how we plan to continue moving our comms architecture forward, enabling you to write more powerful applications – as explained in the accompanying article.

To be fair, and ensure that we maintain a realistic level of expectation, we must acknowledge that it's imperative for us to continue this forward momentum. A lot of you are probably anxiously awaiting the release of our DDKs (driver development kits), and those are on their way. By the same token, a lot of you would also probably appreciate a TCP/IP stack for your comms solutions. Be assured, a TCP/IP stack is also on its way. Other issues, such as memory restrictions and more than one PC Card slot are, as you're well aware, hardware-dependent. And with the emergence of our 2.0 operating system, I think we'll all be watching for new hardware devices from Apple and our licensees which will enable us to really take advantage of the new communications functionality in 2.0.

I think you'll agree that we've made significant strides in our communications story, and I'm only the first to say that there are still issues to conquer. But as we continue to take these steps forward, I hope you'll agree that we really are moving in the right direction. As your first point of contact for communications-related development, please don't hesitate to let me know if you think otherwise!

Our objective is to make it easier for you to help us to complete our communications story, and I feel very strongly that we're making improvements towards that goal. We learned a great deal as we encouraged development of comms solutions for our 1.x product, and based on those insights, I'm proud to say the 2.0 realm has brilliant possibilities!

NTJ

Newton 2.0 User Interface – Making the Best Better

by Garth Lewis, Apple Computer, Inc.

If any single idea served as the guiding principle in the development of the Newton 2.0 user interface, it was this: *listen to the customer.* The feedback from Newton 1.x users, developers, analysts and the press fueled every phase of the re-design. Leveraging two years of feedback, from our own user studies and from the field, the Newton team set out to enhance an already acclaimed UI and to address whatever shortcomings existed. Despite greatly expanding functionality, the team strove to maintain the simplicity that is the essence of Newton's design. In short, the goal was to keep the best and fix the rest.

This overview examines what has changed in the new user interface, what prompted the changes, and how those changes have been received by users. It will not touch on every UI change in Newton 2.0. Instead, it highlights some of the more important changes and gives some of the rationale behind them.

Central to the user complaints about Newton 1.x were problems surrounding data entry. Recognition received its share of criticism, but other areas also frustrated users: difficulty writing in fields and expandos, lack of readability, problems with correction, memory problems (especially when using ink), a sense of UI inconsistency between applications. In short, there was a perceived lack of precision and predictability when it came to getting information into the MessagePad.

Therefore, an overarching goal of Newton 2.0 was to *ease data entry, manipulation and viewing.* Handwriting recognition, still an emerging technology when Newton launched in July 1993, has made big strides over the past year and a half. So too has the effort to find alternatives to recognition. The pervasive use of pickers, a new default font, greater support of electronic ink, and the introduction of "remote writing" all serve to help users get data into their Newton devices, and to make it useful once it's there.

Another overarching goal in the redesign process has been to *establish greater consistency*, in both look and functionality. System fonts, styles, and slip layouts all conform to a single style, thereby maximizing readability and ease of use. Functionality, too, has been generalized throughout the system: "New" and "Show" buttons have been generalized throughout applications, keyboard and recognition buttons are ubiquitous, routing and filing are supported in nearly all areas (including the Extras Drawer).

RECOGNIZE THIS!

Perhaps the most widely heard comment during user tests of Newton 2.0 has been how much recognition has improved. Experienced users perceive enhanced recognition performance using both printed and cursive handwriting styles; new users are impressed by how much better recognition is than they expected. Better accuracy is evident across the board: text and numbers, dictionary and non-dictionary words, and punctuation. Improvements stem in part from better algorithms, larger dictionaries (roughly 90,000 words at press time), less constrained recognition in fields, and an expanded character set.

Still, even with the improvements in recognition performance, studies show that there will continue to be a subset of users who do not have success with recognition (because of handwriting idiosyncrasies or other inhibitors). For those users, Newton 2.0 features a number of recognition alternatives.

In terms of user interface elements to control recognition, the recognition popup has evolved – from a three-button toggle to a popup (fig. 1). This design has many advantages including clarity and extensibility, as well as providing access to handwriting preferences. While the popup adds a tap in most cases, usage patterns suggest that users tend to leave the recognition control in a particular setting. (The need for momentary letter-by-letter setting was eliminated by advances in recognition.)

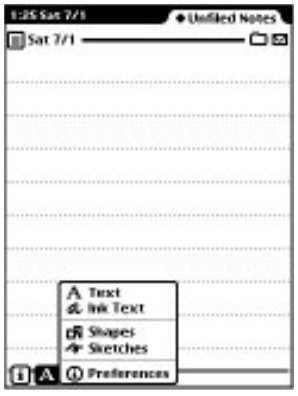

Fig. 1 – screen shot of rec pop

Another key improvement is the addition of remote writing. This feature, manifested by the caret (fig. 2) which indicates where a word is placed, solves the problem of unpredictability of word placement in Newton 1.x. With the caret, there is no ambiguity or guesswork involved. The user can write wherever it is comfortable to write and know with certainty where the recognized text will appear. Editing, too, is made easier. For example, if you want to replace a word in the middle of a paragraph, all you do is select the word and then write anywhere on the screen and the word is replaced. In Newton 1.x, the task of scrubbing, inserting space, and rewriting was often daunting to users. Users are extremely keen on this feature: "I love the fact that you know where the insertion point is! It makes editing and work addition so much easier, not to mention the punctuation marks."

Fig. 2 – screen shot of caret

The corrector popup (fig. 3) has also been enhanced. Thanks to recognition refinements, if a word is misrecognized, the alternate word list is much more likely to contain the correct word. In addition, a new letter-by-letter correction tool provides the user a larger writing area in which to overwrite letters, correct segmentation problems, and so on. Research shows that many Newton 1.x users relied on the keyboard to correct misrecognized words. In Newton 2.0, overwriting is the first line of defense and the new corrector is a welcome fallback. As one user said, "The corrector is much preferred to the old method of just getting the keyboard, because previously, I would often have to use the arrow keys a lot to get to the character I wanted to correct. This makes it easier." The new corrector is especially useful when using smaller fonts (which makes overwriting more difficult). It also offers access via popups to letter alternates. The new corrector can also be used for bulk correction. In this scenario, the user leaves the corrector open and taps the words to be corrected. Developers can define corrector templates for different types of data (dates, times, etc.).

Fig. 3 – screen shot of new corrector

Other recognition changes include an improved "add word to word list" UI, a new word-expansion feature, and the addition of the highly readable "casual" font (which aids overwriting). Two new handwriting gestures were added: a backwards "L" creates a carriage return, and a "V" drawn at the base of two words will adjoin them.

INK TEXT? NOW THIS I LIKE!

Newton 2.0's UI strategy continues and enhances the original Newton approach of offering the user a variety of data entry methods and correction tools for different situations and needs. "Ink Text" (fig. 4) is a critical addition to that arsenal. Ink text is supported in virtually all entry fields in Newton 2.0 and, with few exceptions, ink text and text are interchangeable and can be combined freely. Ink is now displayed in the overview, for example. Advantages include ease of data entry, avoiding recognition, and speed. Ink text also has some memory savings over Newton 1.x ink. Developers are encouraged to support ink in any entry fields in their applications.

Fig. 4 – screen shot of ink text

Early user feedback on ink text is extremely positive. "Now *this* I like," exclaimed one first-time user. "I *love* it. I am much more productive and can fit more on the screen with the shrinking feature," says a veteran user. The wrapping and editing capabilities, combined with deferred recognition, allow users the flexibility of text with the ease of input of ink. "The new improved ink is a blessing," said one power user. Users find ink text especially useful in time-constrained situations (for note-taking at meetings, listening to voice-mail, etc.). Even name cards can be entered in ink (the user is asked under which letter to file it). While some users will choose not to use the feature because they require searchability or value the way recognition "cleans up" their handwriting, others will use it exclusively. As with pickers, ink text provides the user with an option to avoid recognition when the situation demands. "I can read my own handwriting where someone else (or the Newton) can't," explains one user. "Why argue with the Newton over word recognition when just jotting down a note is the desired purpose? It also has the personalized feeling of a handwritten note."

A PROLIFERATION OF PICKERS

One of the first things the new user experiences in Newton 2.0 is a sequence of screens (fig. 5) that helps to configure the Newton device. The purpose of the "setup" sequence is threefold: to allow users to enter the basic information they need to start getting work done; to introduce a variety of UI elements; and to help make their first experience with the product positive and productive. Cartoon-like graphics enliven what is essentially a fairly straightforward task. The end result, according to users, is a positive first impression, increased confidence, and a better out-of-box experience. As users themselves put it: "It leaves good, productive feelings about my first contact." "There's no question that this is a *much* better way to greet a new user than the previous 'blank' (normal) Newton screen."

Fig. 5 – screen shot of Welcome screen

In the setup sequence, the user is first exposed to an important Newton 2.0 UI enhancement: pickers. As in Newton 1.x, pickers allow the user to enter information in a way that is fast, fun, and intuitive. They have the added advantages of being easy to target and taking up minimal real estate. In Newton 2.0, the number and types of pickers have proliferated: date pickers, time pickers, location pickers, people pickers, list pickers and number pickers. Each of these pickers provides an alternative to recognition as a means of data entry. In most cases, the task can be accomplished in a couple of taps. As a power user remarked: "The pickers (in Newton 2.0) are a great leap forward."

The time (fig. 6) and year pickers display many of the pickers' attributes. The numbers are bifurcated to allow ease of targeting. Users like the readability and ease of manipulation they afford. Said one user: "It's easier to change it than trying to hit little arrows." The function of these pickers is not necessarily intuitive for all users, but the vast majority find their function to be easily learned and retained. Designed initially to replicate the old "mechanical digital" radio/alarm clocks, the look of the date/time pickers evolved so that only the flipping numbers remain.

Fig. 6 – screen shot of time picker

The people picker proto (fig. 7) offers users access to their Names data and the ability to make a persistent list of selections.

Fig. 7 – screen shot of people picker

The city picker (fig. 8) combines A-Z tabs and a diamond popup to access a large amount of data.

Fig. 8 – screen shot of city picker

Fig. 9 – screen shot of paper roll with New button popped up

Stationery, But Not Standing Still

One of the key additions to Newton 2.0, the advent of stationery, is facilitated by the New button popup (fig. 9). Status bar buttons that access

popups (signified by a diamond in the button itself) provide built-in extensibility for developers. The New button also answers a common user request for instant access to the end of the paper roll. Users have found the New button to be an intuitive "create" button and it has been generalized to other parts of the OS as well. "The New button is a welcome addition," said one user. "And it will be very helpful when users are viewing things that are not at the bottom of the paper roll."

Among the three stationery types that ship with the Newton 2.0 software, "Note" is the most familiar to Newton 1.x users. Even Notes are different, however. The icon at the left of the separator bar (fig. 10) acts not only as an identifier of stationery type, but as a mechanism for the titling of individual notes (oft-requested by users). The slip also contains pertinent information about the note (date and time of creation, size, and store).

Fig. 10 – screen shot of titling slip open

The checklist and outline stationery (fig. 11) provide the user with enhanced list-making capabilities. The bullet point allows the user to distinguish between discrete items and to create and collapse hierarchical lists. The three-button floating palette is comprehended immediately by both naive and power users. These list stationeries add much more precision and predictability to list-making, another common user request. As one user explained: "An integrated outliner might justify the purchase of a Newton all by itself.... It seemed to work very intuitively."

Fig. 11 – screen shot of outline stationery with sample data

Users seem to appreciate immediately the potential of the stationery metaphor. "My imagination is racing with ideas for other kinds of stationery – an excellent third-party opportunity."

The current date and time are now displayed in the upper left corner of most apps, replacing the spring-loaded clock on the status bar. The new approach brings more information to the top-level and frees up valuable space on the status bar.

In place of the analog clock, there is an "i" button (the international symbol for information). This new button (fig. 12) has various advantages: the user can access application-specific help and preferences, it's extensible, and it localizes well.

Fig. 12 – screen shot of "i" button

The addition of icons within popup menus (the New and Show buttons, the routing icon) is a way to establish distinct identities for the various elements as well as providing some visual appeal (fig. 13).

Fig. 13 – screen shots showing icons in New button and Action button

So Little Time...

Another part of the OS to receive extensive reworking is the Date Book. The graphically-enhanced New button provides the user access to a variety of

meeting and event types, and to-do items. The user can enter data into a slip, or write directly into the calendar's day view as before. In the slip (fig. 16), users can take advantage of a variety of pickers to select the date and time, choose invitees and location, set alarms and frequency. The title field features a picker and increased line spacing. A Notes button in the slip accesses a scrollable notes area.

Fig. 14 – screen shot of dates slip with picker open

The greater use of pickers, the increased line spacing, and recognition improvements in general makes data entry much easier than in Newton 1.x. In addition, the slip/picker approach helps raise what had been buried functionality. Instead of navigating slips buried inside other slips, the user gains accessibility and a greater sense of place.

The Show button (fig. 15) provides the user with a more precise way to access various views of their data. It is used to access day, week, month, and year views, to display the to-do list and Day's Agenda, and as a shortcut to today's date. In Newton 1.x, the way to access various views was not always obvious to users.

Fig. 15 – screen shot of Show button popped up

In the day (fig. 16) and week views, a meeting-type icon sits next to the duration bar. The user can tap the icon to open the meeting slip, and tap and drag to move the meeting. The duration bar is used exclusively to affect the length of the meeting. The result is a simplified UI, where no single element is over-burdened with functionality. As a user commented, "It's better to allow movement of an appointment only by touching the icon, and to change time with the duration bar. It's just easier."

Fig. 16 – screen shot of day view

In another example of addition by subtraction, shape recognition is not allowed in the calendar, except in the Notes area. Without shapes getting in the way of recognition, data entry is made easier.

The Day's Agenda (fig. 17) combines the meetings, events and to-dos into a single view. This feature was created in direct response to user requests to see all their pertinent information in one view. It also allows the user to display unlimited events, since real estate is not an issue. (The crib area – the upper left corner – in the Day view limits the user to seeing three events, after which a more events indicator is displayed.) User response to the Day's Agenda has been enthusiastic. One user put it succinctly, "I love the Day's Agenda. It's awesome."

Fig. 17 – screen shot of Days Agenda with data

Other Date Book enhancements: the first day of the week is user configurable, new AM/PM indicators, ink support, scrollable overview, default alarm times for meetings and events, and the ability to store all new items internally.

In sum, users value very highly the UI enhancements made in the Date Book as well as its integration into the rest of the unit. "It's greatly improved." "I love the amount of information that can be entered about a meeting. A+."

Fig. 18 – screen shot of To Do list

...So Much To-Do

The redesign of the To Do List (fig. 18) reflects a strong user demand to make it more robust and easier to use. The interface has been standardized so that it more closely follows the UI norms elsewhere in the system. Entries are made in slips equipped with wider line spacing. Pickers allow access to key functionality. The user creates to-do tasks by tapping the ubiquitous New button and displays the list using Show. Once in the list, the to-do icon is used to access the slip itself; the priority popup is moved a level deeper. Deleting to-do items (a Herculean feat in Newton 1.x) is made trivial. Implementing in list view rather than edit view has reaped other advantages, including speed. Additional features including reminders, repeating tasks, and date selection are all accessed via picker at the slip level. The result is a more useful and usable to-do list which maintains its top-level simplicity.

System-wide Improvements

Local preferences (fig. 19) can now be accessed through the ubiquitous "i" button. An "always store new items internally" check box helps to mitigate the common problem of managing, or mismanaging, data between cards and the internal store.

Fig. 19 – screen shot of local prefs slip

Filing, too, has seen extensive revamping. Global and local filing capabilities give the user more flexibility in terms of organization. The filing slip (fig. 20) itself has been redesigned to preserve real estate. A "New" button accesses a slip with greater room for text entry, a delete button, and a global/local checkbox.

Fig. 20 – screen shot of new filing slip

Slips now have a dragging affordance, called a "picture hanger" to mitigate some of the problems associated with the previous implementation – lack of self-evidency, inadvertent movement of slips, etc.

Some of the system changes were made to provide the user with more feedback. The duplicate function has new animation and sound effects. Alerts have a neon-like border. A busy indicator (the Newton light bulb) lets the user know when the unit is working. A "comm" indicator (a five-pointed star) alerts the user to a variety of communication- and recognition-related events.

Local scrollers (fig. 23) have been added to several areas, primarily to allow a way to scroll within scrollable applications. In the calendar there was an urgent need for a "more" indicator, an indicator to show that more data exists off-screen. The solution was to provide local scrollers which allow the user to move from hour to hour while the universal (silk screen) scrollers move from day to day (as well as month to month.) The local scroller arrow is black if tapping it will bring more items into view.

Fig. 21 – screen shot of local scrollers with one arrow black

The keyboard icon opens a popup (fig. 22) on the second tap, rather than cycling through multiple keyboards. Advantages: faster access, fewer taps, extensibility.

Fig. 22 – screen shot of keyboard popup

A variety of new sound effects provide the user with feedback. Pen taps produce a random series of "plops" that one user likened to raindrops. "It makes navigating around much less monotonous." Other user favorites include the squeaking sound for attaching items to the clipboard and the calculator's "adding machine" feedback.

Alarms have been upgraded – they are persistent and contain a snooze button with a popup for snooze durations (fig. 23). Also notice the new border for alert slips.

Overviews (fig. 24) now support multiple selection, routing and filing. A gray line separates the check boxes from user data.

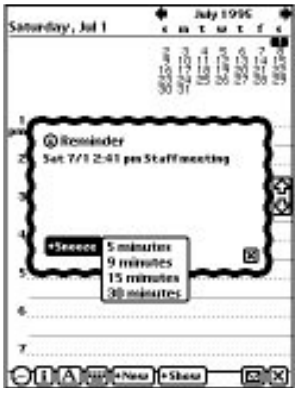

Fig. 23 – screen shot of new alarm slip with snooze popped up

Fig. 24 – screen shot of overview

The two-level undo of Newton 1.x has been replaced with a more standard undo/redo feature. As one user said: "This is really the best way to implement this!"

Routing slips (fig. 25) have a new graphical look, an envelope metaphor that extrapolates from the routing icon.

Fig. 25 – screen shot of new mail slip

Newton 2.0 introduces the ideas of multiple personae and worksites. Mimicking the Names interface, this feature (fig. 26) allows the user to switch personalities and locations painlessly. Users can have both a "corporate" and "freelance" persona, for example, by creating a second Owner card. Or, they can set up additional worksites for home or work with dialing prefixes, printer information, etc.

Fig. 26 – screen shot with worksite card view

EXTRAS! EXTRAS!

The Extras Drawer (fig. 27) is much more robust and easier to manage thanks to new functionality. Filing is now supported, as is multiple selection and deletion. With the addition of application-specific filing, users can configure their file folders in a way that makes sense to them. You can drag icons around, file them, move them to cards, and examine the data (in the "Storage" folder). The new bifurcated folder popup allows users to view internal and external stores, together or separately.

The UI for selecting Extras items is the same as selecting anything else in Newton – hold down the stylus until the ink blob appears. The user can drag through it, circle select or just tap and hold on the icon. Users who complained

of Extras Drawer overcrowding can now accumulate shareware apps to their heart's content. The extras drawer now supports overview and scrolling as well.

Fig. 27 – screen shot of Extras Drawer with filing slip open

Newton 2.0 also allows the user to choose any application, built-in or otherwise, to be the backdrop for their Newton device. By selecting an item in the Extras Drawer and tapping the routing slip, the user can configure their device so that their favorite third-party software acts as the backdrop (fig. 28).

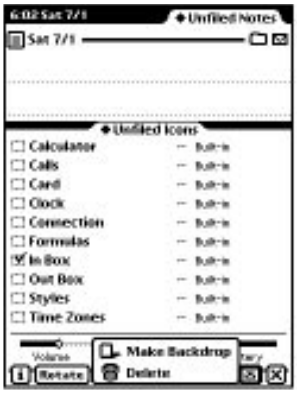

Fig. 28 – screen shot of "Make Backdrop" popup

You'll also notice the addition of a Rotate button in the Extras Drawer. This button will allow the user to display the notepad, the in and out boxes, and Extras in landscape orientation (fig. 29). Users have often requested this feature for reading e-mail, writing some notes, and for fax receive.

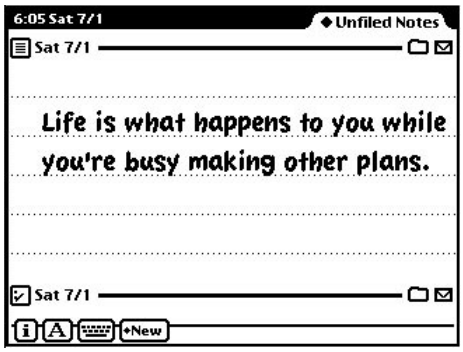

Fig. 29 – screen shot of Notepad rotated

Names Will Never Hurt Me

The Name File (fig. 30) has also undergone extensive redesign to accommodate increased functionality and to ease data entry. All the usual suspects are there: pickers, increased line spacing, New and Show buttons. Unique UI elements include an Add button which allows the user to configure and grow the card to his or her specifications. Input is made at the slip level (consistent with other apps) and editing is achieved by tapping the text to be changed. Expandos are minimized because users complained about dropping ink and their unnatural feel. (They are used in the new Custom fields but are larger and more forgiving than before.)

Fig. 30 – screen shot of Name Card with entry slip open

The A-Z tabs replace the A-Z picker that some users complained was difficult to target. The second tap on a tab takes you to the second letter – an interface that is quickly learned by users.

While some current Newton 1.x users miss the old UI where information was added in a single screen, supporting the increased functionality made the evolution to a multi-slip approach necessary. New users value the customizability of the new approach.

Another change was made in direct response to customer feedback. Beaming business cards was considered too cumbersome in Newton 1.x machines, so a "beam my card" option (fig. 31) has been added to the routing slip in Names. Users can now exchange business cards via beaming with much fewer pen taps than before.

Fig. 31 – screen shot of Beam My Card feature

Out With The Old, In With The New

The In/Out Box (fig. 32) has been upgraded considerably both in look and function. It has been promoted from a slip-based mini-app to a full-fledged application, based on NewtApp, with filing and routing support. Newton 2.0 UI elements are generalized here as well. Check boxes allow for multiple selection. Bullet points allow collapsing and expanding. (The Overview button provides a shortcut to do this too.) The information button allows access to help and local preferences. In Box / Out Box radio buttons at the top of the screen serve a dual purpose of identifying which box you are in and providing quick access to the other. Icons identify transport type and allow previewing of the document itself.

Fig. 32 – screen shot of I/O Box

Local preferences allow for multiple sort orders (by date, by transport, or by status) logging capability, and the ubiquitous internal store preference. Transport-specific preferences use pickers to provide extensive flexibility and user customizability. Print, fax, beam, and e-world preferences allow the user to configure everything from auto-filing to auto-delivery while avoiding recognition completely.

The Fax Viewer interface allows the user the ability to manipulate large binary objects and scale them in three dimensions.

The In/Out Box integrates three discrete architectural pieces – stationery (users can view and organize entries while in the I/O Box), the list manager (they can expand and collapse entries) and the underlying transports (they can activate items, readdress them, log them etc.) . In short, the I/O Box packs a significant amount of functionality into an interface that maintains a top-level simplicity. By utilizing mostly off-the-shelf Newton 2.0 UI components, the creators of the I/O Box have maintained a high level of usability.

Users reactions have been overwhelmingly positive. "This is a nice improvement for all communications functions." "Great. Super implementation." "This makes the Newton much easier to use." "This comes closest to a universal In/Out box than anything I've seen."

Designing for Newton 2.0

The changes made to the Newton interface reflect a maturing of the operating system based on real-world experience. Like a young person who grows up, inherent flaws have been smoothed over and positive qualities (we hope!) shine through. In terms of designing for the new OS, there are a few principles that we followed that developers may find useful:

1) Do yourself, and the user, a favor by easing data entry--provide more space for writing, more pickers, and less recognition where appropriate.
2) KISS – keep it simple, stupid. When it comes to designing for the MessagePad, less is definitely more. Build in breathing space in your apps.
3) Mimic the interface that's already there. Not that we think it's the only way to do it, but users are already familiar with it.
4) Finally, user test, user test, user test.

NTJ

continued from page 1

Newton 2.0: What's the Big Idea?

more complicated than ever. Busy schedules, increased travel, and leaner staff mean that mobile professionals are having to do significantly more – with less. Less resources. Less time. And tighter budgets. A look at the various market analyst reports from companies including BIS Strategic Decisions[1] and Forrester Research[2] outline the major trends about mobile professionals and how they work today:

- Professionals are more mobile. 71 percent of today's 44 million professionals in the United States spend more than 20 percent of their time away from their desk. Whether they're roaming a corporate campus or find themselves 35,000 feet in the air bound for a business meeting, they often find themselves without access to the resources they need to successfully conduct business – support staff, e-mail, key files and documents, and so on.
- Mobile professionals have less support than before. Corporate downsizing, decentralization, and a tight economy worldwide have resulted in leaner support staffs. Few professionals have the luxury of a dedicated support person; in today's working environment, it's much more likely that professionals share support personnel. The result is that professionals have to do more administrative work themselves – faxing, writing correspondence, sending messages, making reservations, and keeping their own calendar. Under extreme pressure, many professionals simply need assistance.
- Access to information is no longer a problem – keeping on top of that information is now a problem. For most of the 1980s and early 1990s, the quest was to give users full access to the information on the network. With client-server models implemented in many companies, access is rarely the issue – from their desktop systems, professionals are able to access many resources of their corporate network, intra networks, and the Internet. The challenge today is handling the deluge of information: sifting through it, pulling relevant information out, and using information intelligently. Today's mobile professionals need better tools to deal with all the information in their lives.
- Staying in touch is a way of life – you need to do be reachable to stay competitive. The proliferation of e-mail, paging systems, voice mail, and other electronic means of keeping in touch has created a world where professionals can be reached regardless of where they are. These new ways of communicating are dramatically changing the way people conduct business. Being "always reachable" has become an absolute necessity, and professionals need tools to help them deal with the barrage of messages they get.

A host of technologies is being deployed to gain a competitive edge

In response to the challenges of the new business environment, mobile professionals have adopted a wide range of technologies to help them deal with their daily flow of information:

- Cellular phones. Over 8 million professionals use cellular phones in the United States to keep in touch.
- Pagers. 8.7 million use pagers to stay in touch with colleagues and customers.
- Portable computers. 4.6 million use portable computers to take information on the road with them.

Each of these devices has enabled mobile professionals to use their time more efficiently. Obviously, each device offers a different way of communicating: the phone provides direct two-way contact; pagers, at least until very recently, offer one-way contact; and portable computers provide a way to actually do desktop computing work on the road. However, the multitude of professionals who take more than one device with them during the day – for example, a phone and a pager, or a phone and a portable PC – indicates that professionals need multiple ways of staying in touch.

Enter the personal digital assistants (PDAs). They combine many of the best aspects of the tools that mobile professionals have come to depend on – phones, dayrunners, personal computers. By offering a combination of these capabilities in a single device, PDAs are, in many ways, more powerful than these other devices.

PDA use is growing in the market, as companies and individuals are beginning to see the advantages of having a small, completely portable information and communications device. Forrester Research, Inc., an independent research firm, recently conducted a study of the PDA market and projected strong growth through the end of the decade. They predict that the installed base of PDAs will be over 8 million in 1999. This number would be be achieved by doubling the installed base annually.[3]

What PDAs can do to help

PDAs offer new capabilities – in an integrated package.

PDAs give users the best of all worlds: they offer the advantages of many of today's devices – without the drawbacks. They deliver the processing power of many personal computers, yet they're much lighter to carry around. They provide the communications capabilities of pagers and PCs, yet they also handle many other functions. They are intuitive to use, but much less intrusive than PCs in a meeting or business setting. Here are some of the advantages that PDA devices provide:

- Mobile data capture. Because of their low weight and small size, PDAs make ideal data capturing devices – for heavy-duty industrial uses, such as gathering inventory data, for taking notes in a meeting, and for jotting down expenses from a business trip. They're unobtrusive, and easy to use.
- Personal information management features, anytime, anywhere. For mobile professionals who need to stay on top of all the little bits of information in their life – appointments, telephone numbers, addresses, and notes – PDAs offer an ideal way to take a personal information management application with them everywhere they go.
- Strong communications capabilities. By combining the best features of pagers, cellular phones, and computers, PDAs deliver robust communications capabilities including paging, faxing, and access to e-mail and on-line and information services either wired or wirelessly. Integrating communications capabilities with other PDA applications such as call and contact management features allows for more efficient communications – and a better use of the user's time.
- Instant access to functionality. Unlike computers, there is no boot-up

time for PDAs. Users have instant access to information and applications, enabling the rapid recording of notes, ideas, facts, figures – even while walking about on the factory floor or between meetings in a corporate setting.
- Smaller, lighter, cheaper – and fewer moving parts. PDAs offer a significant size advantage for mobile professionals. That is, they're lighter than PCs, less expensive, smaller – and contain fewer parts that require service or can break on the road. Though it may seem like a rather small point, the fact that a PDA can be up to 5 pounds lighter than an average portable PC starts to make a significant difference on that long trek to your boarding gate.
- All this – and it can be integrated into your computing world. Of course, all these capabilities aren't really worth having, unless they can be integrated into your world. PDAs, such as the Apple MessagePad and the Motorola Marco, can be effortlessly integrated into computer networks and used as companions to desktop PC systems. Newton PDA products offer superior connectivity, data-sharing capabilities, and synchronization capabilities.

Newton 2.0 Platform Overview: "Organize. Communicate. Integrate."

The Newton platform is the only PDA platform that offers strong capabilities in the three areas that mobile professionals require in a mobile device: organization, communication, and integration with desktop systems.

Organization capabilities

The Newton 2.0 OS offers the most complete set of organizational functions of any PDA, both built-in and from third parties. These "best of class" capabilities include:
- Names – Enables you to easily record and recall names, addresses, telephone numbers, and important contact and personal information.
- Dates – Enables you to keep your calendar, to-do lists, and reminders organized.
- Notes – Enables you to take notes, including freeform notes in digital ink, graphics, and notes that are converted to typed text. Additionally, an API is now supported that enables developers to add different kinds of stationery in addition to plain notes.
- Other organizational tools. Newton 2.0 includes a worldwide map with time zones, conversion tables, common formulas, a call and contact tracker as well as an alarm clock.
- The broadest availability of third party applications from traditional ISVs, and more importantly, hundreds of new developers.

While many of the capabilities enumerated above were also available with Newton 1.0, their power, scope, and ease-of-use have been greatly improved in Newton 2.0. These improvements are highlighted later in this article.

Communication capabilities

The Newton 2.0 offers improved communications capabilities over Newton 1.0. The Newton platform makes it easy for users to:
- Manage all their communications in a universal in and out box
- Send and receive faxes
- Access e-mail
- Access the Internet with a forthcoming TCP/IP stack
- Share information wirelessly with other Newton users by taking advantage of the "beaming" capabilities of the platform
- Add future communications solutions. The Newton communications architecture is a platform for additional hardware, services, and protocols, allowing in-house developers and ISVs to easily build in the communications capabilities that users want. Motorola, for instance, has created a Newton PDA that has a radio built in which supports Ardis and the RadioMail service. Key to Newton 2.0 has been a significant overhaul to the communications APIs and tools.

The Newton platform has been designed to let users accomplish these communication tasks via wired and wireless means such as:
- Connection to computer networks via a built-in serial port. Newton can be easily integrated into AppleTalk and TCP/IP networks
- Wireless connection to networks and on-line services via PC card modems and external modems.
- Wireless access to on-line services through a cellular phone – for instance, users can connect their Newton device to a standard cellular phone and access e-mail services.
- Integrated wireless access through products like the Motorola Marco.

Personal Computer Integration capabilities

The Newton platform is designed to serve as an ideal companion to desktop PCs. Newton devices can be used in the field to gather information for use later on a desktop system back at the office. Of course, the reverse is true, too – while users are in the field, Newton devices can be used to access information that's stored on remote PCs. Newton 2.0 platform offers unprecedented connectivity to PCs:
- Desktop Integration Libraries (DILs). This Apple technology makes it easy for developers to create end-user solutions that allow direct synchronization of data between Newton PDAs and applications on PCs. Ultimately, DILs will make it possible for data in hundreds of PC applications to be synchronized with data on Newton PDAs – so you could use the same personal information management applications or a database on your PC and PDA and keep the two pieces of information in perfect synchronization.
- Newton Backup Utility. This free product gives customers the most critical PC integration features they need. It keeps their data safe by making it easy to back up their Newton PDA information onto their PC – and then restore it when they're ready to use it again. The Newton Backup Utility also allows them to download packages from their PC to their Newton PDA, so they can install software and other files by just tapping a button.
- Newton Connection Utilities. Newton Connection Utilities gives you everything you need to integrate data on your Newton PDA and your PC – even when you're away from your desk. It offers the same backup, restore, and installation features in the Newton Backup Utility. Plus, it lets you synchronize desktop files with corresponding data residing on your Newton PDA. So, for example, your file of names and phone numbers on your PC can be synchronized with the ones on your PDA. You can also import and export text files and information between your PC and PDA. Newton Connection Utilities even allows you to use a PC keyboard to enter data directly into your Newton PDA. And by using a modem, you can perform these tasks even when you're on the road.
- Newton Press. Newton Press provides mobile professionals with a convenient drag-and-drop method of publishing electronic books that

can be read on a Newton PDA. Using your Windows or Macintosh computer, all you have to do is drag the text files, graphics, e-mail, and reference information you want and drop them onto the Newton Press icon; an electronic book is automatically created that you can format and import into your Newton PDA. You can view, fax, annotate, and print the book. And you can even distribute it to other Newton PDA users.

A superior and open platform

Unlike other PDA platforms that are essentially proprietary and closed systems, Newton is a completely wide open system. Apple's goal is to enable both commercial developers and in-house development teams, to quickly create applications. The Newton 2.0 platform offers the most mature set of development tools in the industry:

- Newton Toolkit. This robust development environment offers rapid prototyping, a library of extensible components, and an interactive development cycle that quite literally lets developers see results in minutes. We have added support for a 2.0 platform file and most importantly brought these tools to Windows. (We like to joke that even though people may have made the wrong choice in personal computers, they can still make the right choice with PDAs.)
  - Newton Book Maker. This application enables you to effort1lessly create electronic books out of documents on a PC for use on Newton PDAs. Book Maker is similar to Newton Press, but is a more powerful development tool that allows integrating custom functionality using NewtonScript.
  - C++ tools. Apple will be offering low-level programming tools that enable calling C++ routines from within Newton applications, letting developers capitalize on previous development efforts to get their applications to perform computation intensive tasks.

These tools have enabled corporate development teams to prototype custom Newton applications that are fully integrated into the fabric of their existing network, database, and PC environment in matter of a few weeks or months as opposed to years.

The largest number of third-party applications

The Newton platform also has the advantage of offering users more than 1,000 off-the-shelf applications, shareware, freeware and utilities which do everything from organizing your day, to tracking expenses, to highlighting places to visit in foreign cities. These applications have been created by familiar names in the computer industry – Claris, Intuit, and a host of other innovative companies. It is this creative community of developers that has truly helped position the Newton platform as the premier PDA platform for customers.

Newton 2.0: Improvements on all fronts

Based on extensive customers testing, the Newton 2.0 OS makes significant improvements over Newton 1.0 software. Every aspect of the software has been updated, optimized, or rearchitected to allow for higher performance and enhanced ease-of-use:

Improved input methods

Getting information into a Newton PDA has never been easier. Apple has made improvements on three main fronts:

- Improved handwriting recognition. Newton 2.0 now offers more accurate handwriting recognition as compared to 1.0. Early on during the development of 1.0, every user study indicated that customers would not modify their handwriting style to achieve better recognition results. This led us to a very dictionary centric approach to recognition. In Newton 1.0, when a word was written that was included in the dictionary, it usually would get recognized. However, for words not in the dictionary such as names and places the poor results were worthy of great laughs and parodies. Fortunately, Palm Computing developed Graffiti, which made recognition work if you trained yourself to write letters in a very precise, if somewhat unusual way. The popularity of Graffiti forced us to rethink our original assumptions that users were not willing to adapt to recognition. With Newton 2.0, the printed recognizer now allows customers to write non dictionary words with excellent results as long as they lift the pen between each letter. This improved printed recognition is no longer based primarily on a dictionary lookups, but instead looks at individual letters first, then comparing to a dictionary. Of course, Apple, along with Paragraph, Inc. has also made significant improvements to the cursive recognizer by improving the low level algorithms and increasing the number of words in the dictionary.
- Ink Text. In addition to improved recognition, the new OS also enables users to record ideas and text in "ink text" which allows for rapid note taking without any waiting for text to be recognized before moving on to other words. Ink text flows words into sentences in the same manner as recognized or typed text, but does not perform recognition. Each word is moved into place after the previous, as opposed to leaving the ink as a graphic where it was originally written. Ink text can also be mixed with sentences containing regular words as well, but this is more than ink word processing. Ink text can be used in any application and recognized as regular information, without having to be recognized or typed. For example, a customer can enter a meeting title in ink, set an alarm, and when the alarm goes off the alert displays the meeting title in ink.
- Keyboard entry of data. When you connect your Newton PDA to a Windows or Mac OS computer you can use that PC's keyboard to enter data directly into your Newton PDA. The text you type instantly appears on your PDA's screen. In addition, Apple will be offering a small portable hardware keyboard for when you need to respond to volumes of e-mail while on the road.
- Data export and import. Newton Connection Utilities and the Desktop Integration Libraries enable you to easily import and export data from a Newton PDA to a PC and vice versa. This makes transferring data and files simple and easy. With Newton Press, you can easily publish Newton books of information to take with you as well.
- Ease of use. The Newton interface has been made more intuitive and consistent across all areas:
  - Consistency. One of the most important goals for the Newton 2.0 user interface was to insure greater consistency across the built-in applications. You'll notice that the familiar folder and envelope buttons, as well as a keyboard icon, are available everywhere. Some new features we have added are a "New" button to all built-in applications and a "Show" button for those applications with multiple views of information such as the calendar. These buttons are becoming analogous to the Mac OS "File and Edit" present in all applications.
  - Predictability. In 1.0 we tried to intuit what a user was trying to do and more often than not, we were wrong. In Newton 2.0 we have made the system more explicit in various areas including text entry. Users now have a caret (^) displayed on the screen for directing the exact placement of text. You can still place the caret anywhere

on screen to insure free form note taking, unlike a word processor, but now you can be sure the words you enter go where you want.
  - Intelligence. People are creatures of habit and machines should be smarter about watching what people do and learning from their actions. One example of this is the familiar diamond for a pop up list of choices. In 2.0, lists can be made smarter and keep track of the information you enter such as the people you send mail to, the companies you work with, and the things that you do.
- Increased performance. Newton 2.0 has many performance improvements in the areas of NewtonScript code and also soup and data management. In general, NewtonScript performance for compute-intensive functions is almost as fast as if the code were compiled, but the code size is still significantly smaller as expected in an interpreted language. In addition, with support for indexing, tags and general improvements in soups, customers have the ability to manage large amounts of information in database-oriented applications or more simply through the use of folders.

SUMMARY

Competition intensifies, but Apple continues to innovate and develop the market.

The Newton OS was developed to be a scalable architecture to support a wide variety of small, portable PDA devices. Since Apple coined the term PDA in 1992, dozens of companies have entered the PDA market.

Today, Newton PDA products, that is, the Apple MessagePad and Motorola Marco, remain the dominant platform – in terms of market share, performance, breadth of solutions, and developer flexibility. The platform is a generation – or two – ahead of competitors.

The Newton platform offers mobile professionals real solutions for organization, communications, and personal computer integration. In addition, Apple is providing a complete set of developer tools and programs to make the platform more open to ISVs and custom in-house development. The fact that Newton 2.0 offers all of this makes it the premier PDA platform which will continue to grow and develop the market.

No other platform offers this today and most will not be able to offer this for years to come. Newton 2.0 is a major step for the Newton platform and we believe the future holds great opportunities for Apple, our partners, developers and most importantly our customers.

[1]BIS Strategic Decisions, Mobile Professional Segmentation Study, January 1995
[2]Forrester Research, PDA's: Time Will Tell, August 1994
[3]Forrester Research, Inc. PDA's: Time Will Tell, August 1994, page 9.

NTJ

continued from page 1

Technical Overview of Newton 2.0: The Developer Perspective

data to the user in views, and allows the user to edit some or all of the data, then it is a potential candidate for using the NewtApp framework. NewtApp is well suited to "classic" form-based applications. Some of the built-in applications constructed using the NewtApp framework include the Notepad and the Names file.

STATIONERY

Stationery is a new capability of Newton 2.0 that allows applications to be extended by other developers. If your application supports stationery, then it can be extended by others. Similarly, you can extend another developer's application that supports stationery. You should also note that the printing architecture now uses stationery, so all application print formats are registered as a kind of stationery.

The word "stationery" refers to the capability of having different kinds of data within a single application (such as plain notes and outlines in the Notepad) and/or to the capability of having different ways of viewing the same data (such as the Card and All Info views in the Names file). An application that supports stationery can be extended either by adding a new type of data to it (for example, adding recipe cards to the Notepad), or by adding a new type of viewer for existing data (a new way of viewing Names file entries or a new print format, for example).

To support stationery, an application must register with the system a frame, called a data definition, that describes the data with which it works. The different data definitions available to an application are listed on the pop-up menu attached to the New button. In addition, an application must register one or more view definitions, which describe how the data is to be viewed or printed. View definitions can include simple read-only views, editor-type views, or print formats. The different view definitions available in an application (not including print formats) are listed on the pop-up menu attached to the Show button.

Stationery is a powerful capability that makes applications much more extensible than in the past. Stationery is also well integrated into the NewtApp framework, so if you use that framework for your application, using stationery is easy.

VIEWS

New features for the view system include a drag-and-drop interface that allows you to provide users with a drag-and-drop capability between views. There are hooks to provide for custom feedback to the user during the drag process and to handle copying or moving the item.

The system now includes the capability for the user to view the display in portrait or landscape orientation, so the screen orientation can be changed

(rotated) at any time. Applications can support this new capability by supporting the new ReorientToScreen message, which the system uses to alert all applications to re-layout their views.

Several new view methods provide features such as bringing a view to the front or sending it to the back, automatically sizing buttons, finding the view bounds including the view frame, and displaying modal dialogs to the user.

There is a new message, ViewPostQuitScript, that is sent to a view on request when it is closing, after all of the view's child views have been destroyed. This allows you to do additional clean-up, if necessary. And, you'll be pleased to know that the order in which child views receive the ViewQuitScript message is now well-defined: it is top-down.

Additionally, there are some new viewJustify constants that allow you to specify that a view is sized proportionally to its sibling or parent view, horizontally and/or vertically.

Protos

There are many new protos supplied in the new system ROM. There are new pop-up button pickers, map-type pickers, and several new time, date, and duration pickers. There are new protos that support the display of overviews and lists based on soup entries. There are new protos that support the input of rich strings (strings that contain either recognized characters or ink text). There are a variety of new scroller protos. There is an integrated set of protos designed to make it easy for you to display status messages to the user during lengthy or complex operations.

Generic list pickers, available in system 1.0, have been extended to support bitmap items that can be hit-tested as two-dimensional grids. For example, a phone keypad can be included as a single item in a picker. Additionally, list pickers can now scroll if all the items can't fit on the screen.

Data Storage

There are many enhancements to the data storage system for system software 2.0. General soup performance is significantly improved. A tagging mechanism for soup entries makes changing folders 7000% faster for the user. You can use the tagging mechanism to greatly speed access to subsets of entries in a soup. Queries support more features, including the use of multiple slot indexes, and the query interface is cleaner. Entry aliases make it easy to save unique pointers to soup entries for fast access later without holding onto the actual entry.

A new construct, the virtual binary object, supports the creation and manipulation of very large objects that could not be accommodated in the NewtonScript heap. There is a new, improved soup change notification mechanism that gives applications more control over the notification and how they respond to soup changes. More precise information about exactly what changed is communicated to applications. Soup data can now be built directly into packages. Additionally, packages can contain protos and other objects that can be exported through magic pointer references, and applications can import such objects from available packages.

Text Input

The main change to text input involves the use of ink text. The user can choose to leave written text unrecognized and still manipulate the text by inserting, deleting, reformatting, and moving the words around, just like with recognized text. Ink words and recognized words can be intermixed within a single paragraph. A new string format, called a rich string, handles both ink and recognized text in the same string.

There are new protos, protoRichInputLine and protoRichLabelInputLine, that you can use in your application to allow users to enter ink text in fields. In addition, the view classes clEditView and clParagraphView now support ink text. There are several new functions that allow you to manipulate and convert between regular strings and rich strings. Other functions provide access to ink and stroke data, allow conversion between strokes, points, and ink, and allow certain kinds of ink and stroke manipulations.

There are several new functions that allow you to access and manipulate the attributes of font specifications, making changing the font attributes of text much easier. A new font called the handwriting font is built in. This font looks similar to handwritten characters and is used throughout the system for all entered text. You should use it for displaying all text the user enters.

The use of on-screen keyboards for text input is also improved. There are new proto buttons that your application can use to give users access to the available keyboards. It's easier to include custom keyboards for your application. Several new methods allow you to track and manage the insertion caret, which the system displays when a keyboard is open. Note also that a real hardware keyboard is available for the Newton system, and users may use it anywhere to enter text. The system automatically supports its use in all text fields.

System Services

System-supplied filing services have been extended; applications can now filter the display of items according to the store on which they reside, route items directly to a specified store from the filing slip, and provide their own unique folders. In addition, registration for notification of changes to folder names has been simplified.

Two new global functions can be used to register or unregister an application with the Find service. In addition, Find now maintains its state between uses, performs "date equal" finds, and returns to the user more quickly.

Applications can now register callback functions to be executed when the Newton powers on or off. Applications can register a view to be added to the user preferences roll. Similarly, applications can register a view to be added to the formulas roll.

The implementation of undo has changed to an undo/redo model instead of two levels of undo, so applications will need to support this new model.

Graphics and Drawing

Style frames for drawing shapes can now include a custom clipping region other than the whole destination view, and can specify a scaling or offset transformation to apply to the shape being drawn.

Several new functions allow you to create, flip, rotate, and draw into bitmap shapes. Also, you can capture all or part of a view into a bitmap. There are new protos that support the display, manipulation, and annotation of large bitmaps such as received faxes. A new function, InvertRect, inverts a rectangle in a view.

Views of the class clPictureView can now contain graphic shapes in addition to bitmap or picture objects.

Sound

The interface for playing sounds is enhanced in Newton 2.0. In addition to the existing sound functions, there is a new function to play a sound at a particular volume and there is a new protoSoundChannel object. The protoSoundChannel object encapsulates sounds and methods that operate on them. Using a sound channel object, sound playback is much more

flexible – the interface supports starting, stopping, pausing, and playing sounds simultaneously through multiple sound channels.

Built-in Applications

Unlike in previous versions, the built-in applications are all more extensible in version 2.0. The Notepad supports stationery, so you can easily extend it by adding new "paper" types to the New pop-up menu. The Names file also supports stationery, so it's easy to add new card types, new card layout styles, and new data items to existing cards by registering new data definitions and view definitions for the Names application. There's also a method that adds a new card to the Names soup.

The Dates application includes a comprehensive interface that gives you the ability to add, find, move, and delete meetings and events. You can get and set various kinds of information related to meetings, and you can create new meeting types for the Dates application. You can programmatically control what day is displayed as the first day of the week, and you can control the display of a week number in the Calendar view.

The To Do List application also includes a new interface that supports creating new to do items, retrieving items for a particular date or range, removing old items, and other operations.

Recognition

Recognition enhancements include the addition of an alternate high-quality recognizer for printed text and significant improvements in the cursive recognizer. While this doesn't directly affect applications, it does significantly improve recognition performance in the system, leading to a better user experience. Other enhancements that make the recognition system much easier to use include a new correction picker, a new punctuation picker, and the remote writing feature (new writing anywhere is inserted at the caret position).

Specific enhancements of interest to developers include the addition of a recConfig frame, which allows more flexible and precise control over recognition in individual input views. A new proto, protoCharEdit, provides a comb-style entry view in which you can precisely control recognition and restrict entries to match a predefined character template.

Additionally, there are new functions that allow you to pass ink text, strokes, and shapes to the recognizer to implement your own deferred recognition. Detailed recognition corrector information (alternate words and scores) is now available to applications.

Routing and Transports

The Routing interface is significantly changed in Newton 2.0. The system builds the list of routing actions dynamically, when the user taps the Action button. This allows all applications to take advantage of new transports that are added to the system at any time. Many hooks are provided for your application to perform custom operations at every point during the routing operation. You register routing formats with the system as view definitions. A new function allows you to send items programmatically.

Your application has much more flexibility with incoming items. You can choose to automatically put away items and to receive foreign data (items from different applications or from a non-Newton source).

The Transport interface is entirely new. This interface provides several new protos and functions that allow you to build a custom communication service and make it available to all applications through the Action button and the In/Out Box. Features include a logging capability, a system for displaying progress and status information to the user, support for custom routing slips, and support for transport preferences.

Endpoint Communication

The Endpoint communication interface is new. There is a new proto, protoBasicEndpoint, that encapsulates the connection and provides methods to manage the connection and send and receive data. Additionally, a derivative endpoint, protoStreamingEndpoint, provides the capability to send and receive very large frame objects.

Specific enhancements introduced by the new endpoint protos include the ability to handle and identify many more types of data by tagging the data using data forms specified in the form slot of an endpoint option. Most endpoint methods can now be called asynchronously, and asynchronous operation is the recommended way to do endpoint-based communication. Support is also included for time-outs and multiple termination sequences. Error handling is improved.

There have been significant changes in the handling of binary (raw) data. For input, you can now target a direct data input object, resulting in significantly faster performance. For output, you can specify offsets and lengths, allowing you to send the data in chunks.

Additionally, there is now support for multiple simultaneous communication sessions.

Utilities

Many new utility functions are available in Newton 2.0. There are several new deferred, delayed, and conditional message-sending functions. New array functions provide ways to insert elements, search for elements, and sort arrays. Additionally, there's a new set of functions that operate on sorted arrays using binary search algorithms. New and enhanced string functions support rich strings, perform conditional substring substitution, translate a number of minutes to duration strings ("16 hours", "40 minutes"), and support easier formatting of numbers.

Books

New Book Reader features include better browser behavior (configurable auto-closing), expanded off-line bookkeeping abilities, persistent bookmarks, the ability to remove bookmarks, and more efficient use of memory.

New interfaces provide additional ways to navigate in books, customize Find behavior, customize bookmarks, and add help books. Book Reader also supports interaction with new system messages related to scrolling, turning pages, installing books, and removing books. Additional interfaces are provided for adding items to the status bar and the Action menu.

NTJ

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 ujnlimited 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.

Newton Associates Plus Program

This 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