Newton Technology Journal

Volume III, Number 2

March 1997

Inside This Issue

Newton Technology

Behind Bars

by Michael S. Engber, Apple Computer Newton ToolBox Group

Introduction and Long Disclaimer

This article discusses various issues related to the screen orientation and the Button Bar in the most recent line of Newton devices. For example, on the MessagePad 2000 the Button Bar is drawn on the LCD screen. Therefore, it is possible to obscure it, reconfigure it, change its appearance, or entirely replace it.

This article describes how to do things that will put the Newton in a non-standard configuration that users may find confusing. Normal, well-written applications should have no need to do this. This information is intended for applications with special needs and should be thought of more as techniques for customizing a particular Newton device rather than general purpose APIs.

Do not give in to the temptation to add frivolous features to your applications. Every application does not need a preference checkbox to put its icon on the Button Bar.

Furthermore, most of the information presented here is not officially supported and is not guaranteed to work on future Newton Devices. Applications that use it run the risk of incompatibility with future systems. Of course, efforts will be made to maintain compatibility, but changes in design may make compatibility impossible. For example, the Button Bar could be radically redesigned making many of the options no longer applicable.

Despite this caveat, I'm sure there are still developers anxious to use this information. By following the suggestions in this article you will maximize your application's chances for compatibility with future Newton devices.

continued on page 3

Communications Technology

Binary Communications

by Ryan Robertson, Apple Computer, Inc.

Sending and receiving binary data using the Newton OS has always been something of a voodoo art. Unlike other computer systems, the Newton's communication architecture does not provide a GetByte or PutByte equivalent. The Newton OS instead implements a very flexible endpoint architecture.

Sending data is fairly straight-forward; you create your data, then pass it to the endpoint's Output method. You can specify to send data either synchronously or asynchronously.

Receiving data is much different than with a traditional computer operating system. To receive data, you must post an input specification which specifies the type of data you want to receive. Input specifications are — by nature — asynchronous.

There are advantages and disadvantages to the flexibility of this system.

Advantages include:

- You can post an input specification, then go do other processing until the termination conditions of the specification are met.
- The abstraction provided by the Newton's communication architecture makes it very easy to switch between different transport types. Its just as easy to open a serial connection as it is to open an ADSP connection.

Disadvantages include:

- You must do extra work to send and receive just a few bytes of data.
- The architecture doesn't lend itself well to stream based programming.
- It is very difficult to simulate synchronous input.

continued on page 6

Newton Technology Journal

Volume III, Number 2 *March 1997*

Published by Apple Computer, Inc.

Jennifer Dunvan • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

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

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

Contributors
Bob Ebert, Mike Engber, Ryan Robertson, Garth Lewis, Don Davis, Styephen Breit, Bent Schmidt-Nielson, David Austin, Debbie Carlton

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

Jessica Courtney • *Production Editor*

InfoGraphix • *Design/Production*

Editor's Note

Letter From the Editor

by Jennifer Dunvan

The Future of Newton

Given the uncertainty around Apple recently, the understandable question on everyone's mind has been, "What is the future of Newton?" I want to take some time now to reassure our developers and customers that the Newton platform is strong and moving ahead as planned. Amidst the chaos, the Newton Systems Group is completely intact. In fact, we have recently hired more engineers. At the time of this writing, the MessagePad 2000 and eMate 300 are off the production line and in the hands of the first new customers. By press time, you should all have your units in hand, and the Newton Systems Group will be well on their way developing what's next for Newton.

Clearly, the Newton effort does not end with shipping the eMate 300 and MessagePad 2000. New technology is being developed in the form of enhanced C++ and development tools, and developer releases of the Newton driver DDKs. New futures are being investigated in the form of Newton/Java research and development. Now, with the explosion of the world-wide-web, the shrinking costs of mobile communications hardware, and the publicly acknowledged need for low-cost high-tech educational tools, we're seeing the emergence of real-world problems for which Newton technology provides solutions. The new combination of greatly increased chip speeds, useful new form factors, a licensed OS, and modernized software make the answer obvious to the trained eye, and an exciting possibility to those new to Newton.

So let's get down to business. This issue of Newton Technology Journal is chock full of useful technical information. "Behind Bars" illustrates what you can do with the new soft button bar on the MessagePad 2000, and other button bar issues. You can find out more about sending and receiving binary data using Newton endpoints in "Binary Communications." As promised, part II of our data storage series goes into detail about how the OS caches entries and soups, and "Surviving the Grip" discusses what to do when Newton still needs the card you removed. Dragon Systems is our guest author this issue with an article outlining their efforts with speech recognition. See how WinCE stacks up against Newton in "Why Newton Beats Windows CE". We also introduce you to a key members of the Newton Systems Group, and let you in on the extraordinary experience of field-testing the eMate 300.

I recently had the pleasure of talking with many potential customers at the HIMMS conference (Healthcare Information Management Systems) in San Diego, CA where the MessagePad 2000 enjoyed a great reception. No less impressive than the hardware itself were the software solutions being demonstrated. Virtually every doctor, nurse, or healthcare administrator who visited the Newton pavilion left with a lasting impression of the promising possibilities that Newton solutions could bring to their organizations.

To all our Newton developers: Thank you for the solutions you contribute, thank you for your faith in Newton technology, and thanks for hanging in there through the uncertainty. It's now time to watch what the future will bring, and then continue to create an even better one!

Jen Dunvan
<dunvan@newton.apple.com>

continued from page 1

Behind Bars

Each section ends with a compatibility note to explicitly summarize these considerations.

Changing the Screen Orientation

There has always been an API for getting the current screen orientation, `GetOrientation`. Now there is an API for changing the screen orientation, `SetScreenOrientation`. It takes one argument, an integer specifying the desired orientation (See **Table 1**).

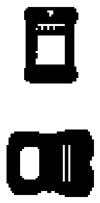
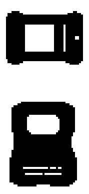
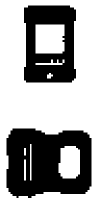
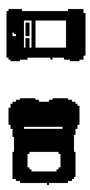
			
kPortrait 0	kLandscape 1	kPortraitFlip 2	kLandscapeFlip 3

Table 1. Screen Orientations for the MessagePad 2000 and eMate300

The return value of `SetScreenOrientation` is a `nil` or non-`nil` value indicating failure or success. For example, if the backdrop application is not compatible with the new orientation, the rotate will fail. Also note that `SetScreenOrientation` may present the user with a dialog giving the option of canceling the rotation because certain applications won't work in the new orientation. If the user cancels the operation, `SetScreenOrientation` will return `nil`.

Compatibility Notes

- `SetScreenOrientation` was added in 2.1. Check for its existence using `GlobalFnExists` if your code needs to run on earlier ROMs.
- Remember to use `LegalOrientations` to get an array of the available orientations.
- Check the return value (or call `GetAppParams`) instead of assuming the unit is in the new orientation.
- Do **not** use the seductively named function, `SetOrientation`. It is unsupported and does not do what its name implies.

Moving the Button Bar

There are hooks within the system to allow changing the position of the standard Button Bar and the position of the controls, scroll arrows and overview button. They are all controlled by userConfig values which are arrays of four elements, one element for each orientation. For example, `array[kPortrait]` specifies the value for the portrait orientation.

To restore any of these settings to their default value, set their entire userConfig value to `nil`. Individual orientations can also be set — use their default value by specifying `nil` for the corresponding array element.

`buttonBarPositions`
: `buttonBarPositions` is an array of four elements which specify the location of the Button Bar. The allowed values for the elements are the symbols `top`, `left`, `bottom`, & `right`; and `nil`.

`buttonBarControlsPositions`
: `buttonBarControlsPositions` is an array of four elements which specify the location of the scrolling and overview controls within the Button Bar. The allowed values for the elements are the symbols `top` & `bottom`, for when the Button Bar is on the right or left, the symbols `left` & `right`, for when the Button Bar is on the top or bottom, and `nil`.

`bellyButtonPositions`
: `bellyButtonPositions` is an array of four elements which specify the location of the overview button relative to the scroll arrows. The allowed values for the elements are the symbols `outside`, `inside`, `left`, & `right`; and `nil`.

Compatibility Notes

- Make sure there is a soft Button Bar (`if GetRoot().Buttons.soft` …) before setting these values.
- Make sure your positioning of the controls is consistent with the position of the Button Bar.
- Positioning the Button Bar so that the appArea becomes less than 320 high means that views without a `ReOrientToScreen` method will be unable to open – à la the behavior of MessagePad 120 and 130 units in landscape orientation.

Covering the Button Bar

Applications that wish to cover the entire screen need to ensure they work in all screen orientations and all Button Bar positions – especially on MessagePad 2000 units in landscape orientation with the Button Bar on the left – the key point being the appArea having non-zero top-left corner.

Remember, children of the root view open relative to the appArea. This means using a <*left*, *top*> coordinate of <0, 0> which will not cover the Button Bar unless it's on the right or bottom edge of the screen. In this situation the view's bounds need to be offset by the appArea's <*left*, *top*> global coordinates. GetAppParms now returns this information in the `appAreaGlobalLeft` and `appAreaGlobalTop` slots.

Depending on the reason for covering the entire screen, there are different approaches to use.

If the goal is simply to maximize the visible area of the base view, then the Button Bar should be obscured only if it's located on the LCD screen as it is on the MessagePad 2000. On a MessagePad 130, the root view encompasses a larger area than the LCD screen – the tablet. Obviously, drawing is limited to the screen, so applications don't increase their visible area by covering the Button Bar on a MessagePad 130.

Below is some sample code you can add to a base view's `viewSetupFormScript` to maximize visible area.

```
local params := GetAppParams();
if GetRoot().Buttons.soft then
  self.viewBounds :=
    OffsetRect(UnionRect(params.appAreaBounds,
params.buttonBarBounds),
               -params.appAreaGlobalLeft,
               -params.appAreaGlobalTop)
else
  self.viewBounds := params.appAreaBounds;
```

If the goal is to prevent users from accessing the buttons, then the Button Bar should be obscured regardless of whether or not it is on the LCD screen.

On units like the MessagePad 130, remember to take into account the fact that part of the base view will be off-screen. For example, it's important to ensure that the close box is visible. A simple way to accomplish this is by having a child view whose bounds are the appArea, and locate the rest of the application within that child.

Below is some sample code you can add to the base view's `viewSetupFormScript` to cover the entire tablet.

```
local params := GetAppParams();
self.viewBounds := GetRoot():LocalBox();
if params.appAreaGlobalLeft then
  self.viewBounds :=
    OffsetRect(self.viewBounds,
               -params.appAreaGlobalLeft,
               -params.appAreaGlobalTop)
```

Compatibility Notes

- On some units (e.g. the eMate 300), the buttons are not located in a view at all. Therefore, covering the entire tablet does not prevent the user from accessing the buttons (e.g. opening up the Extras Drawer).
- `appAreaGlobalLeft` and `appAreaGlobalTop` are new slots in `GetAppParams`. Check them before using if your code has to run on older devices.

CLOSING THE BUTTON BAR

Applications that wish to replace the Button Bar need to use `KillStdButtonBar` to close it. Closing the Button Bar view directly will leave a hole – the `appArea` will not be readjusted to include the unused Button Bar area.

`KillStdButtonBar` is intended to accommodate Button Bar replacements. If your application simply wants to cover the entire screen `KillStdButtonBar` is **not** the correct way to accomplish this. Instead, open a full screen view as previously described.

After closing the Button Bar, it is assumed that your application will provide the user with a replacement (e.g. a floater) that will provide a way to do the following:

- scroll up and down
- overview
- open the Extras Drawer (from which the user can access former Button Bar icons)

Replacing the Button Bar does **not** mean you should replace the contents of the `buttons` root view slot with a reference to your view. It is important that you do not do this. The system relies on the `buttons` slot containing the Button Bar – whether it's open or closed. In fact, there is no need at all for the root view to have a slot referencing your view – you can create an open your view using `BuildContext`.

Your replacement view is private, the system will not send it a message. It does not need to provide any of the Button Bar APIs described in this article. In order to be notified of changes (e.g. card yanking, package loading) you should register for changes on the "Packages" soup. This is an undocumented soup and you should definitely not rely on the format of its entries. When you are notified of a change, of any type, simply rebuild your list of icons.

To close the Button Bar you can use code like the following:

```
KillStdButtonBar(Array(4, '{buttonBarPosition: none}));
```

To restore the Button Bar use the following code:

```
KillStdButtonBar(nil);
```

It is also possible to reserve an edge of the screen for your replacement Button Bar so it can sit outside the appArea – like the standard Button Bar does. This would be useful if, for example, you create a replacement Button Bar that is thinner than the standard Button Bar.

The following code reserves the bottom twenty pixels in all four screen orientations.

```
KillStdButtonBar(Array(4, '{buttonBarPosition: bottom,
buttonBarThickness: 20}));
```

You can vary the position, and or thickness of the reserved area in each orientation by varying the position and thickness of the corresponding element of the array passed to `KillStdButtonBar`.

`KillStdButtonBar` makes no attempt to arbitrate conflicts between applications. If two applications try to use `KillStdButtonBar` there is no way for one application to quit and restore the previous state of the Button Bar. All you can do is put back the standard Button Bar.

It is assumed that conflicts will be rare. It seems unlikely (undesirable) for users to have two different Button Bar replacements installed at once. Nevertheless, program defensively. Check if the Button Bar is open (`call kViewIsOpenFunc with(GetRoot().buttons)`). If it's not open, inform the user that there is a conflict and do not call `KillStdButtonBar`.

Compatibility Notes

- Only use `KillStdButtonBar` if you're replacing the Button Bar, not if you want to cover it.
- `KillStdButtonBar` is a new API. Check for its existence using `GlobalFnExists` if your code needs to run on earlier ROMs.
- Configuring the Button Bar area so that the appArea becomes less than 320 high means that views without a `ReOrientToScreen` method will be unable to open – à la the behavior of MessagePad 120 and 130 units in landscape orientation.
- Stick to the APIs (`GetPartCursor`, `GetPartEntryData`, `GetPartEntries`) rather than accessing the entries in the "Packages" soup directly.

Configuring the Button Bar

We recommend letting users control what's in the Button Bar. It's simple for users to drag icons in and out of the Button Bar themselves. Changing the Button Bar behind user's backs can confuse them. For example, an application that automatically installs itself in the Button Bar will have to push some other icon off causing the user to wonder where that icon went.

The mechanism by which icons are marked and being located in the Button Bar is simply filing – specifically, being filed in the `_ButtonBar` folder. This is accomplished by changing an icon's `labels` slot using the Extras Drawer method, `SetExtrasInfo`. However, using `SetExtrasInfo` to move an icon to the Button Bar provides no control over its placement in the Button Bar.

There are two Button Bar methods that give you more control, `GetPartEntries` and `ReConfigure`. `GetPartEntries` takes no arguments and returns a frame with two slots, `fixed` and `mobile`. These slots contain arrays of part entries – à la the Extras Drawer method, `GetPartCursor`.

The fixed entries are "fixed" because they cannot be moved by dragging. By default, only the Extras Drawer icon is fixed. It's important that the Extras Drawer icon be fixed. Users should not be able to drag the Extras Drawer icon into the Extras Drawer. The mobile entries are "mobile" because they can be dragged in and out of the Extras Drawer by the user.

As was previously stated, users should be in control of the contents of the Button Bar. We do not recommend making your application's icon fixed. Fixed icons are intended for use by licensees and VARs creating Newtons that are only used for specific purposes.

The ordering of the entries in the fixed and mobile arrays corresponds to the order of the icons in the Button Bar. The icons for the fixed entries are first, followed by the icons for the mobile entries.

As mentioned before, the elements of the fixed and mobile arrays are part entries. You should not examine them directly – just as you should not directly examine the entries returned by `GetPartCursor`. There is an Extras Drawer method, `GetPartEntryData`, that returns a frame of information about the entry (icon, text, appSymbol, …).

The Button Bar's `ReConfigure` method takes one argument, a frame of fixed and mobile entries, and reconfigures the Button Bar. The order of the entries in the arrays controls the order of the icons in the Button Bar. For convenience, `ReConfigure`, also accepts appSymbols instead of part entries. This allows an icon to be added or removed from the Button Bar without having to lookup its actual part entry.

A related Button Bar method that you may want to use in conjunction with `ReConfigure` is `IconCapacity`. `IconCapacity` takes no argument and returns the number of icons (fixed plus mobile) the Button Bar can currently hold. This number varies depending on the orientation and location of the Button Bar. It will be zero if the Button Bar is closed.

Compatibility Notes

- Make sure there is a soft Button Bar (`if GetRoot().Buttons.soft` …) before using these methods.
- Make sure the Extras Drawer Icon is fixed – to prevent it from being dragged into the Extras Drawer.
- Remember that whether or not the Extras Drawer is the backdrop application affects whether or not it shows up in the Button Bar.
- Use `IconCapacity` to prevent overfilling (and those annoying notifications).
- Stick to the APIs (`GetPartCursor`, `GetPartEntryData`, `GetPartEntries`) rather than accessing the entries in the "Packages" soup directly.

Changing the Spacing and Font of Icons

There are hooks in the system to allow changing the font and spacing of the icons in the Extras Drawer and the Button Bar. They are all controlled by userConfig values.

```
buttonBarIconSpacingH
buttonBarIconSpacingV
```

These two userConfig values control the spacing of the icons in the Button Bar. They are integers specifying the spacing in pixels. They both default to 40 in the MessagePad 2000. The vertical spacing (`buttonBarIconSpacingV`) is only used when the Button Bar is laid out vertically – along the right or left edge of the screen. The horizontal spacing (`buttonBarIconSpacingH`) is only used when the Button Bar is laid out horizontally – across the top or bottom edge of the screen. To restore either of these settings to their default value set their userConfig value to `nil`.

```
extrasIconSpacingH
extrasIconSpacingV
```

These two userConfig values control the vertical and horizontal spacing of icons in the Extras Drawer. They are integers specifying the spacing in pixels. They default to 64 horizontally and 52 vertically in the MessagePad 2000. They have no effect when the Extras Drawer is in overview mode. To restore either of these settings to their default value, set their userConfig value to `nil`.

```
extraFont
```

This userConfig value controls the font used for the icon labels in both the Extras Drawer and the Button.Bar. You should stick to the integer font specifications (e.g. (`userFont9 + tsPlain`) or (`simpleFont9 + tsBold`)). Using the integer representation in this instance accomplishes two things; it reduces NS Heap usage (non-default userConfig values occupy NS heap space) and it restricts you to the set of built-in fonts. Using a font that is not in ROM is an extremely bad idea because the font could be removed. This information is stored in a soup. A user may be forced to do a cold-boot in order to remove a bogus font specification.

Compatibility Notes

- Stick to using built-in fonts for `extraFont` and specifying them in integer form.
- Remember to accommodate icons with two line titles when changing the Extras Drawer spacing.
- Make sure there is a soft Button Bar (`if GetRoot().Buttons.soft` …) before setting `buttonBarIconSpacingH` and `extrasIconSpacingV`.
- Setting `extraFont` and `extrasIconSpacing`X has no effect on earlier ROMs.

References and Suggested Reading

Dublin, Louis I., “Water Fluoridation: Facts, not Myths.” , Public Affairs Pamphlet Number 251B, New York, The Public Affairs Committee. 2nd edition, 1967.

“In many American cities, a technical debate – *whether to raise the fluoride content of public drinking water as a dental health measure*– is attracting nearly as much attention as juvenile delinquency, education, automobile accidents, or the hydrogen bomb.”

Sharp, Maurice, Extra Extra: Extras Drawer Features in Newton 2.0. Newton Technology Journal, February 1996, pp. 1,17-89

This article discuses using part entries in the extras drawer. These are the same entities that are used in the Button Bar.

NTJ

Newton Toolkit Version 1.6 for Windows Ships

Beginning March 17, Newton Toolkit for Windows is available for purchase through the Apple Developer Catalog and selected retail outlets. WinNTK provides the same rich object-oriented environment and NewtonScript language as MacNTK. WinNTKrequires a 486DX (or better) processor, CD-ROM, Windows 3.1 (with Win32s), Windows 95 or Windows NT. Memory requirements are 8MB RAM for Windows 3.1 users and 16MB RAM for Windows 95 and Windows NY users.

To order call:

U.S.	800/282-2732
International	716/871-6555

Online orders:
`<http://www.devcatalog.com>`
or e-mail:
`<adc.orders@apple.com>`.

continued from page 1

Binary Communications

In this article we will discuss how to send and receive binary data using the Newton’s communication architecture. We will also discuss some of the stumbling blocks to watch for as you write your endpoint code. This article assumes that you are familiar with the basics of setting up an endpoint, sending data, and using input specifications. For more information on any of these, check out the Newton Programmer’s Guide chapter titled “Endpoint Interface”.

Sending Binary Data

After you have setup and connected an endpoint, you send data by using the endpoint’s Output method. The Output method takes three arguments:

- The data to output. For our purposes, this data will be your binary object. Your binary object can either be allocated from the heap using the global function MakeBinary, or it can be allocated from a store using either the NewVBO or NewCompressedVBO store method. If you are outputting data that ranges in size from 1 byte to 2 KB, you are better off allocating the binary object from the NewtonScript heap. If you are outputting data that ranges from 2 KB on up, you should use a VBO. For more information about using VBOs, check out the Newton Programmer’s Guide chapter titled “Data Storage and Retrieval”.
- An array of output options. This argument is currently only used by the Newton Internet Enabler transport. If you are using UDP, you would specify the address and port of the machine to send the packet.
- An output specification. An output specification is a frame that encapsulates information about how to send the data. The output specification tells the endpoint what type of data you are sending and whether you want to send asynchronously or synchronously. For sending binary data, you also specify a target slot. The target slot holds a frame with an offset and a length slot. The offset slot specifies the offset into the binary object at which to start sending data. The length slot specifies how many bytes to send from the offset. If you are outputting packetized data, you will need to specify packet flags using the sendFlags slot.

There are two different ways you might choose to output your data. You might send the entire binary object with one Output call, or you may choose to send the data using consecutive Output calls.

Sending all of the binary data at once is very straight-forward. You pass the binary object as the first parameter to the endpoint’s Output method, then setup the target slot of the output specification so that the offset is zero and the length is the length of the binary data. Note that you do not need a target slot if you want to send all the binary data. If you output data synchronously, the output call will return when the entire binary object has been sent. If you output asynchronously, the output specification's CompletionScript will be called when the entire binary object has been sent. If you do output asynchronously, be sure that you do not modify the binary object until the CompletionScript has been called. Also, note that the calling context of the CompletionScript is the output specification itself, not the endpoint. This can make accessing your application's methods and data structures a little more difficult. For some tips on making this easier, see the "Tidbits" section below.

A drawback of sending the entire binary object at once is that you cannot use a deterministic progress indicator to let the user know how much of the data has been sent. You can, however, use a non-deterministic indicator — a barber pole for instance — to let the user know that some action is taking place. Note that you will not be able to use a barber pole if you output synchronously. It is highly recommended that you use the asynchronous form of all endpoint methods. It is much easier on the Newton OS, and will increase the performance of your data transfer. Below is a code example showing how to send all the binary data at once.

```
local myData := MakeBinary( 1024, 'binary );
local theCompletionScript := func( ep, options, result )
        begin
          // Handle completion here...
        end;

// A synchronous output example
fEndpoint:Output( myData, nil, {async: nil,
            form: 'binary} );

// An asynchronous output example
fEndpoint:Output( myData, nil,
         {async: true,
          form: 'binary,
          CompletionScript: theCompletionScript} );
```

The second way to output data is to use consecutive Output calls. We call this "chunking" the data. If you are chunking the data, you will set a different offset in the target slot of the output specification with each call to Output. Depending on your data, you may also set a different length in the target slot. An advantage of outputting your data in chunks is that you can use a deterministic progress indicator to let the user know exactly how much of the data has been sent. Note that you should limit the size of each chunk to under 2 KB.

Outputting data in chunks is a place where you can potentially run into problems using synchronous outputs. Each time you call Output synchronously, the Newt task (the task that all NewtonScript runs in) will be forked. Each fork will require more system memory. If you output data in chunks synchronously inside of a loop, you run a risk of running out of system memory with repeated calls to Output. Forks are not "cleaned up" until execution returns to the main NewtonScript event loop. Below are some code examples showing the difference between outputting chunks synchronously and asynchronously.

```
// A code example showing synchronous output
local myData := GetDefaultStore():NewVBO( 'binary, 40960 );
// 40 KB
// Output in 1 KB chunks.
for i := 0 to 40960 -1024 by 1024 do
  begin
    fEndpoint:Output( myData, nil, {async: nil,
                form: 'binary,
                target: {offset: i, length: 1024} } );

    // Update progress indicator here...
  end;
```

At first glance, the above code example looks very simple, however the Newton will very likely reset halfway through the data transfer. There is just not enough system memory to fork the Newt task 40 times. Here is an example using asynchronous outputs.

```
// A code example showing asynchronous output.
fEndpoint.data := GetDefaultStore():NewVBO( 'binary, 4096 );
// 4 KB

// The offset slot is used in the output specification's CompletionScript
fEndpoint.offset := 0;

// The amountSent slot is only necessary for updating a progress bar
fEndpoint.amountSent := 0;

fEndpoint.completionFunc := func( ep, options, result )
    begin
      if NOT result then
        begin
          // Update the amount of data that has been sent
          ep.amountSent := ep.amountSent + 1024;

          // Update the progress bar here...

          // Make sure we have not sent all the data before
          // trying to send another chunk
          if ep.offset + 1024 < Length( ep.data ) then
            begin
              ep.offset := ep.offset + 1024;

              ep:Output( ep.data, nil,
                    {async: true,
                    completionScript: ep.completionFunc,
                    form: 'binary,
                    target: {length: 1024, offset:
ep.offset}} );
            end;
        end;
    end;

// Setup a progress bar here

fEndpoint:Output( fEndpoint.data, nil, {async: true,
          form: 'binary,
          completionScript: fEndpoint.completionFunc,
          target: {length: 1024, offset: fEndpoint.offset}
} );
```

This code is definitely more complex than the synchronous case. To output asynchronously, we must keep a few different variables around so that we know how much we have output, what our current offset is, what the data is, and what the completion function is. Notice that almost all of the work now takes place inside of the output specification's CompletionScript method. We first check to make sure that the result parameter of the CompletionScript was nil. If it was non-nil, then there was an error outputting data so we should probably abort the output and notify the user. Next, we update the amount sent variable. The amount sent variable is only used for updating a progress bar. We then check to make sure that we have not yet sent all the data. This code example is assuming that we are outputting in multiples of the length of the binary data. Finally, we output the next chunk of data. All of this adds a little more code over the synchronous case, however its is well worth it.

When do you want to send data in chunks versus sending it all at once? It depends on the size of the data. If you are outputting a small amount of data that requires one or two seconds to transmit, I would recommend sending all the data at once. If your data would require more than a couple of seconds to send, I would recommend sending it in chunks. That way you will be able to let the user know exactly how much of the data has been sent. Users are much happier when they have an idea of how long the data transfer will take.

Receiving Binary Data

You must first post an input specification to receive any type of data through an endpoint. An input specification is a frame that describes the characteristics of the data that you want to receive. An input specification tells the endpoint what kind of data you want to receive, and what the terminating conditions are for the data. Input specifications are, by definition, asynchronous. If necessary, you can emulate synchronous input by opening a modal view, handling input inside of that view, then closing the view. Doing this is cumbersome and not recommended, and suffers from the same forking issues as synchronous communications.

When you post an input specification, you provide a callback routine called the InputScript. The InputScript is called when a terminating condition of the input specification has been met. Terminating conditions can include:

- Byte counts. A byte count tells the endpoint how much data you want to receive for this particular input specification.
- End sequences. This is typically used when receiving strings. Examples include characters such as unicodeCR, $., etc. You can also specify a string such as “Login:” or “Password:”.
- End of packet. This is used for packetized transports such as ADSP and Sharp IR.

When receiving binary data, the only two terminating conditions you may use are byte count and end of packet. However, you may not even need these. When you post an input specification to receive binary data, you must add a target slot to that specification. The target slot is a frame with two slots: data and offset. Unlike any other input form, you must preallocate the binary object you want incoming data munged into. This means that you have to know the length of data you will be receiving ahead of time. The data slot in the target frame holds the binary object to write the incoming data to. The offset slot in the target frame tells the endpoint where to start placing data. For instance, if you have an offset of 10, all incoming data will be added to the binary object starting at byte 10.

A binary input specification is normally terminated when the target binary object is filled up. So normally, you don’t need to specify any type of terminating conditions. You can, however, specify a byte count if you want the input specification to terminate before the target binary object is filled.

As with outputting data, there are two different ways to input your data: You may receive an entire binary object with one input specification or you may choose to receive a “chunk” of the binary data with one input specification.

Receiving all of the binary data at once is very straight-forward. You will post an input specification with a target binary object and no byte count. When enough binary data has been received to fill up the target binary object, the InputScript of the input specification will be called, and will be passed the target binary object as its second parameter. As with outputting the entire binary object, you cannot use a deterministic progress indicator when receiving all of the data at once. Below is a code example of how to specify an input specification to receive a large binary object.

```
// Prepare the binary object to receive data into
local myData := GetDefaultStore():NewVBO( 'binary, 4096 );
// 4 KB

// Setup the input specification
local inputSpec :=  {
          form: 'binary,
          target: {data: myData, offset: 0},
          InputScript: func( ep, data, terminator, options)
            begin
              // Process the binary data here...
            end,
          CompletionScript: func( ep, options, result )
            begin
              // Handle errors here
            end,
          };

// Finally, post the input specification
fEndpoint:SetInputSpec( inputSpec );
```

When 4 kilobytes of data have been received, the input specification’s InputScript will be called. A common mistake is to assume that the calling context of the InputScript is within your application’s view hierarchy. The calling context of an InputScript is the input specification frame. See the “Tidbits” section below for information on how to access your application’s methods and data structures from within the InputScript .

Receiving the data in chunks requires re-posting an input specification with a different offset in the target slot. You will use the byte count terminator to determine the size of the chunk. Here is an example of what your input specification might look like:

```
// Prepare the binary object to receive data into
local myData := GetDefaultStore():NewVBO( 'binary, 4096 );
// 4 KB

// Setup the input specification
fEndpoint.offset := 0;
fEndpoint.amountReceived := 0;
fEndpoint.inputSpec :={
        form: 'binary,
        termination: {byteCount: 1024},
        InputScript: func( ep, data, terminator, options)
          begin
            ep.amountReceived := ep.amountReceived + 1024;

            // Update progress indicator here...

            if ep.offset + 1024 < Length( data ) then
              begin
                ep.offset := ep.offset + 1024;
                ep:SetInputSpec( {_proto: ep.inputSpec,
target: {data: data, offset: ep.offset} } );
              end;
          end,
        CompletionScript: func( ep, options, result )
          begin
            // Handle errors here
          end,
        };

  fEndpoint:SetInputSpec( {_proto: fEndpoint.inputSpec,
target: {data: myData, offset: fEndpoint.offset} } );
```

The InputScript will be called after 1024 bytes of data has been received. There are a couple of important things to point out in the above code example. First, we are explicitly setting the target slot of the next input specification inside of the InputScript. A common misconception is that you only need to post one input specification, and that the offset slot gets updated automatically for you as each chunk of data is received. In reality, you must update the offset slot, then re-post the input specification each time you want to receive a chunk of data. You will notice that I use proto inheritance for each input specification. This is done to reduce the RAM footprint of the code. The third important thing to notice is that in the InputScript, I am setting up the next input specification using the data parameter of the InputScript. The data parameter is simply a reference to the original binary object. Instead of storing a reference to the data as we did in the output case, we can just use that parameter when we post the next input specification.

Again, the same rules apply towards receiving the data in chunks and towards receiving all the data at once.

TIDBITS

The calling context of the Output method's CompletionScript is the output specification, and the calling context of the input specification's InputScript is the input specification. Because of this, accessing your application's methods and data structures requires a little more work.

Here are three possible ways to accomplish this:

- Create your endpoint as a child of your application by adding an _parent slot to the endpoint frame which refers to the application base view or some other view within your application. You will then be able to access your application through the first argument to the InputScript or the CompletionScript.
- Add an _parent slot to your input or output specification frame which refers to the application base view or some other view within your application.
- Reference your application's base view directly by using `GetRoot()`.(kAppSymbol).

BINARILY CHALLENGED

There are two bugs in binary communications that could cause you some trouble. The first deals with switching input forms.

There are two major forms of input types; stream-based which include the binary form and frame form, and byte-based which include the forms of bytes, string, and numbers. The difference between these forms is in how incoming data is buffered. The stream-based form writes directly into a destination object, whereas the byte-based form writes data into an intermediate NewtonScript buffer for endSequence and filter processing.

Buffered data can be lost when you switch between the two types: The data is not correctly copied between the different buffers.

The work around to this problem is to turn your communications protocol into a request-respond protocol. Do not send data to the Newton device until it signals it is ready to receive new data.

The second bug deals with allocating a binary object.

When you allocate your binary object, a temptation is to allocate it inline. By inline, I mean that you call MakeBinary, NewVBO or NewCompressedVBO inside of the target slot of the input specification. Here is an example:

```
local inputSpec := {
  form:  'binary,
  target: {data: MakeBinary( 1024, 'binary), offset: 0},
  InputScript: func( ep, data, terminator, options )
    begin
    // Handle input here
  end,
CompletionScript: func( ep, options, result )
  begin
    // Handle errors here
  end,
};
```

Because of a bug in the Newton OS, if you allocate a binary object inline, the unit will reset when it receives the first byte of data. To work around this bug, allocate the binary object as a local variable, then use that variable in the data slot of the target frame.

VIRTUALLY YOURS

Virtual binary objects (VBOs) can be very useful, although there are some important memory issues to keep in mind. A nice feature of VBOs is that you do not have to allocate the entire object before you start adding data to it. As you add data via BinaryMunger or StrMunger, the virtual binary object will grow, if necessary, to accommodate the data. Note that you do have to preallocate the entire object if you are using it in an input specification.

There is a downside to the flexibility of VBOs. Resizing a VBO can add a performance hit to your code. If possible, try to allocate the entire binary object ahead of time.

IN CONCLUSION

Here are some important things to remember:

- It is recommended that you use the asynchronous form of all the endpoint methods.
- If you receive data in chunks, the offset slot of the input specification's target frame is not updated automatically. You must update this slot yourself before you post another input specification.
- If you send or receive more than 2 KB of data, you should send it in chunks in order to give the user a better indication of the progress, and to reduce memory overhead.

Sending and receiving binary data on a Newton device can seem complex. Once you understand the quirks you will also realize the flexibility that is offered in the Newton's communication architecture. And always remember that the facts, although interesting, are irrelevant.

Ryan Robertson would like to thank Jim Schram for his help with this article.

NTJ

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

eMate 300 in the Classroom

by Garth Lewis, Manager, User Experience, Newton Systems Group

In the bright fluorescent light of a portable classroom, thirty sixth-graders huddle in small groups behind ten translucent green eMates. The lively chatter of their voices, and the steady hum of the air conditioning, create a noisy roar. For Rodney Palmer, social studies teacher and technology coordinator, the sound is music to his ears.

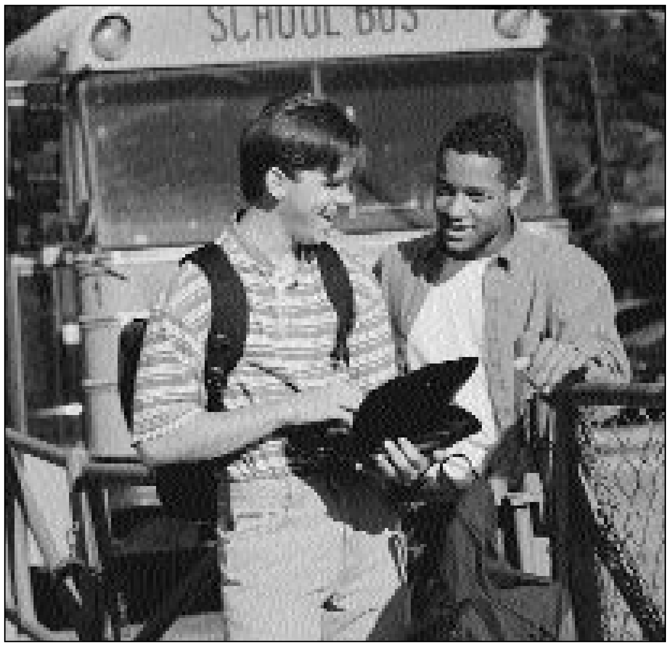
"This is great!" he says, beaming. "This is the sound of kids who are engaged in what they're doing!" Palmer had agreed to be part of an early seeding of eMate prototypes and was amazed by the reception they were getting. "In all my years of teaching, I've never seen a group of students get into a lesson so quickly." The seeding was one of several conducted last summer to get early feedback on the eMate. The reactions from educators and students were then plowed back into the design effort.

The eMate concept was born out of requests from educators for an affordable, portable, computing tool geared toward education, that would address the equity and access issues most schools face. They wanted a computer that provided a core set of functionality to complement the multimedia machines that they had, or perhaps didn't have, in their schools.

Between July and December 1996, NSG's User Experience group placed ten Apple eMate 300 prototypes into elementary, middle, and high school classrooms in a variety of subject areas. Teachers incorporated the eMates into their existing curriculum while a team of researchers observed, videotaped, and interviewed everyone involved. Now, after more than a year of development, the first EVT units were in the hands of their toughest critics—students. "It's so cool!" exclaimed one fifth-grader as she opened the lid of the eMate for the first time. "I gotta have one," said another. "Please make it cheap!"

Students from all grade levels reacted enthusiastically to the eMate's futuristic look, its translucency and clamshell design. Thomas Meyerhoffer, eMate's industrial designer, worked to create a product that "ventures into an emotional space." In addition to providing solutions to ergonomic challenges (the handle, armrests, pen holders, sloping keyboard, and rugged backpack-friendly shape), the eMate design has a rare emotional appeal that attracts both children and adults. Said one East Coast teacher: "When I think of a laptop, I think of a box...but that's not a box. Is it a laptop? (She opens the lid.) Oooooh. I think I just fell in love."

The personal connection students make with the eMate translates into real learning advantages. Students who previously exhibited motivation problems have shown measurable improvements in productivity. Palmer points out that one assignment, a unit on note-taking, usually takes days to accomplish. With the eMates, the students did it in one day. "The eMates gave them a way to focus their attention." The combination of the easy-to-use interface, the direct manipulation of the pen, and the novelty of the eMate, fuel a sense of excitement and empowerment. The students master the eMate quickly, feel good about this accomplishment, and transfer those feelings onto their work.



In many ways the Newton OS is ideally suited to students. The "instant on" feature appeals to young people with limited attention spans (and to teachers who struggle to engage them.) Students prefer the pen to the mouse for drawing and manipulating objects. The graphical interface, icons, and casual font have a simplicity that seems almost childlike. The sound effects and animation provide a sense of play without encroaching on the learning process.

Many teachers prefer the eMate's simplicity, even its gray-scale display, because it's less distracting to students than conventional computers. "I see a lot of the kids, particularly the elementary students, they have color and all of that, and all they want to do is play games with it," says one elementary school teacher. In contrast, the eMate emphasis is on organizing and communicating ideas in written form. And when a teacher needs to get students' attention, he or she can simply ask them to "close your eMates."

The eMate's long battery life quickly grabs teachers' attention. With twenty-four hours per charge, and one-hour recharge time, there's a better chance that the eMate will be available when students need it. In a third grade class using both eMates and AlphaSmart keyboards, a teacher gave the class a writing assignment. After a few minutes one of the students returned wearing a hapless expression. "All the AlphaSmarts are dead."

The eMate's portability is also viewed as a major asset by teachers.

"Now technology can follow the students around, instead of the students following the technology," said one teacher, enthusiastically. Assignments begun in the classroom can be finished on the eMate as homework. This is especially important for students who don't have computers at home.

In a fifth-grade social studies class, the teacher had groups of 3-4 students, each armed with an eMate, move between ten stations within the classroom. At each station, the groups analyzed a political cartoon, recorded their interpretation in the word processor, and found modern corollaries. By the end of the exercise, they were able to beam their work to each other. "Students need to be able to move and be hands-on," explained one educator. "It's a great way to learn."

In another exercise, students used the drawing program to design personalized stationery. They quickly learned to flatten the screen to facilitate drawing and to drag out shapes using the pen. They used stamps, sometimes resizing them, to create borders and backgrounds. Finally, they printed their work. In the course of one class period, every student in the class had mastered the drawing program and printing, and completed their assignment.

It is this benefit, the ability to put more computers in the hands of more students, that educators see as significant. Pam Fox, an elementary school principal, says, "If they have to wait for a turn on the big machine with all the bells and whistles, they may not get a turn. But with this small machine which is less expensive, they might get some hands-on experience two or three times a week which makes a huge difference."

The core application, Newton Works, allows students to do the vast majority of daily computing tasks on the eMates. With only one Macintosh in her classroom, "there's always a huge traffic jam at the computer" says Amy Bloom, a seventh grade teacher. Educators are frustrated that students are using $2000 multimedia machines to do basic word-processing. June Schiller, an elementary school principal, sees the eMate as a way out of that quandary. "The thing that takes the most time… is the writing. I don't see this as taking the place of doing that other stuff, of CD-ROM and the other, but I see it as a cheap way of doing a lot of the work that takes the most time in a classroom setting."

The eMate's back-to-basics approach makes it simple for teachers to incorporate it into existing curriculum. For example, an elementary school teacher gave her students the assignment to write a poem. A third of the students used the ten available eMates while the remaining twenty students used pen and paper. At the end of class, the ten students using eMates printed out their poems and turned them in along with the twenty handwritten ones. No modification of the teacher's lesson plan was needed. The only practical difference was the poems written on the eMates had been word-processed, spell-checked, and could be easily read by the teacher.

The concept of sharing took on a whole new meaning when using the eMate. In a high school class, students from two classes used the eMates during consecutive 45-minute periods. The teacher used the multiple user setting to allow the students' data to be stored separately within the device. The students were able to work the entire class period without taking time out to copy their data to the desktop computer. Another teacher used the multi-user classroom mode to allow groups of students within one class to share an eMate while maintaining discrete workspaces.

In classrooms where teachers used the eMate Classroom Connection software to copy and retrieve data to a desktop, students easily navigated the software. The "Dock" button on the keyboard was a useful shortcut for students. The simplified interface made the direction of data flow immediately clear. The single document transfer lasted just a few seconds, making the total interaction quick and efficient. With the option to "move" data to the desktop, the eMates can be "wiped clean" for students to use in the next class period.

Collaboration was an important element in each of the classrooms we visited. The sharing of work via infrared transmission was fun for students ("it's like magic!") and opened up new opportunities for collaborative work. One teacher had groups of students work on separate pieces of a project on different eMates and then beam them to one machine for assembly into a whole. In another class, students wrote one sentence of a story, beamed it to their neighbor who added another sentence, and they beamed it to another student, and so on. By the end of the exercise, there were ten communally-written stories of ten sentences, and a group of very excited children.

A consensus of teachers agree that the possibilities presented by eMates in the classroom are endless. Teacher requests for sample activities to serve as a "jumping-off point" led to an Apple-supplied Teacher's Guide. After a few weeks, they predict, there will be as many different ideas and approaches as there are teachers who use them. If the initial feedback from students and teachers is any indication, the reaction to eMates in the classroom will be enthusiastic indeed.

NTJ

"Teachers' eMate Wish List"

Teachers who participated in the early seeding of eMates offered their ideas for future applications and enhancements. Here are a few examples:

database software

foreign language modules (using text to speech)

customized stamps

journal application

utility to allow teachers to manage student information

assessment and grading programs

simultaneous broadcasting of assignments to multiple eMates

Ethernet support

curriculum modules

literacy programs

reference materials

textbook supplements

Hyperstudio equivalent

wireless connection to desktops and printers

diffuse infrared

If you are interested in registering your application, send e-mail to Joe Bishop at `<bishop@apple.com>`.

Advanced Techniques

Data Structures, part 2 of 3

by Bob Ebert, Newton Developer Advocate, ebert@newton.apple.com

INTRODUCTION

This series of articles focuses on information useful to those who have mastered the basics of the Newton data storage APIs. It assumes the reader is already familiar with NewtonScript and the Newton data storage concepts in the Newton Programmer's Guide (NPG) and has some experience writing Newton applications.

In part 1 of this series we talked about how soups index their entries, and how to create efficient searches. There we asserted that reading in entries was slow. This article gives the details behind what happens when entries are read and written. Knowing how this works can help you write the most efficient applications.

Correction

In part 1 of the series, the first paragraph in the first section, "Under the Hood," showed incorrect calls to `soup:AddXmit` and soup:`AddToDefaultStoreXmit`. Both calls were missing a second argument, which is the change symbol for the notification.

Caveat Coder

Warning: This article contains undocumented, unsupported details regarding how the current release of the Newton OS caches data storage objects. The information is presented because I believe that understanding what's going on helps. Knowledge like this makes it easier to design efficient code, easier to debug problem code, and may give you ideas for your own designs. This information is not presented so that you can write code that relies on the current design. Do not do this.

At least one developer made this mistake, creating a 1.x application that used an undocumented slot in a cursor. That application subsequently failed because of changes made for the 2.0 release of the OS. What was worse was that the operation in question could have easily been done in a supported way.

There is almost always a supported way to do what you want. Look for it!

Entries

As explained in part 1, entry data isn't kept in the user store in a form that is directly accessible to NewtonScript. There isn't anything like a frame on the user store. Instead, strings are written one place, the rest of the frame is serialized and written to another place, and tagged and indexed slots are stored still elsewhere for efficient searching.

When you call the `Entry` method of a cursor, you get a frame built up from the data on the user store. We generally blur the distinction between the data on the store and the frame that temporarily exists in memory, calling both things an **entry.** When a distinction needs to be made, the frame in the NewtonScript heap is called the **cached entry frame** or simply **cached entry**.

At some point the OS must read data from the store and create the cached entry frame. Obviously, this must be done before any data in the entry can be accessed. However, it's often the case that an entry needs to be referenced but no data from that entry is needed. When you move a cursor using a method like `Next`, `Prev`, `Move`, or `Reset` the cursor "points at" an entry, but at this point nothing may need data from the entry. It would be a waste of time and memory to do the work of creating the cached entry frame as soon as a cursor pointed to an entry.

Entries are Fault Blocks

A soup entry is really a special object called a **fault block**, which is a special class of object composed of two parts. One part is a simple NewtonScript frame often called the cached frame. The other part is a handler which knows how to create and save the cached frame.

Most of the OS treats a fault block as if it were a simple frame. When a slot is accessed the OS checks to see if the cached frame exists, and if so the slot is simply looked up in that frame. When a slot is added or changed it works the same way, if the cached frame exists the slot is set in that frame.

When the cached frame doesn't exist, a fault occurs and a message is sent to the handler, which creates or **faults in** the cached frame. Once this is complete and the cached frame exists, slot access continues as described above.

In the case of soup entries, the cached frame is the cached entry frame. It's what the rest of your code reads and modifies when you work with soup entries. When a cursor accesses a new entry, a fault block gets created with a handler that knows how to retrieve the data, but the cached frame does not yet exist. It's not until the first time some code looks at the slots in the entry that the cached entry frame gets created. This explains why validTests are potentially slow. The `validTest` function typically looks at slots in the soup entry, which reads in the entry data from the user store (deserializing the entry's elements) and creates the cached frame (allocating space from the NewtonScript heap.)

The overhead of reading in the soup entry is incurred the very first time a slot in the entry is touched.

There are functions which work with a entry that do not cause the entry to be faulted in. Most of the global functions that work with entries, such as `EntryUniqueID`, `EntryModTime`, or `EntryRemoveFromSoupXmit` don't cause the cached entry frame to be created.

You can read more about entry caching in the Newton Programmer's Guide section on data storage. Fault blocks (and how to create your own versions) are covered in the section on Mock Entries.

It's now easy to understand how some of the other entry management functions work. It is the cached entry frame that holds all modifications made to a soup entry. These modifications don't become permanent until `EntryChangeXmit` is called. `EntryChangeXmit` causes the fault block's handler to write the data in the cached frame to the user store. `IsSoupEntry` checks to see if the object passed is a fault block for soups, rather than a regular frame.

`EntryUndoChanges` works by throwing away the cached entry frame. The next time someone needs to access data in the entry, the entry handler faults again, creating a new cached frame from the (unmodified) data on the user store.

`AddXmit` and the other entry adding methods are an unusual case. When you call an add method, you pass a regular NewtonScript frame that will be turned into a soup entry. The add method does the necessary work to write the data in the frame to the user store, but it then has to somehow turn that frame into a fault block, so that any references to that frame now refer to the soup entry (the fault block.)

These add methods are very unusual functions which effectively modify one of their arguments. Note the distinction: lots of functions modify the contents of a passed frame, array, string, or other binary object, but all function calls in NewtonScript are call-by-value, so actually modifying an argument isn't otherwise possible. `ReplaceObject` is what actually accomplishes this trick. The add methods uses `ReplaceObject` to change any and all references to the passed frame into references to the fault block.

Entry Management

With the 2.0 release of the Newton OS, some new functions were added to allow you to more closely manage the entry's cached frame.

`AddFlushedXmit` does exactly the same thing that `AddXmit` does with respect to creating the fault block, writing the data to the store, updating indexes, etc. The difference is the fault block that's created will not have its cached frame set. There are also flushed versions of the add methods for union soups.

As part of adding a soup entry with `AddXmit`, the OS does two important things. It walks the frame being added, writing the data to the store, and it uses `EnsureInternal` to create the cached frame, so that the requirement that data in soup entries is safe from card ejection is met. This `EnsureInternal` step can be expensive. Since `AddFlushedXmit` doesn't create the cached frame, it can skip the `EnsureInternal` step. The process of reading in an entry from the user store (faulting it in) guarantees the cached frame will be safe from card ejection.

`AddFlushedXmit` saves both time and memory, and can be a real win if you know that nothing is going to cause the entry to be faulted in right away. On the other hand, if things are set up so that an entry is used or modified right after it's created, `AddFlushedXmit` doesn't help, since the first access will cause the cached frame to be created. You should experiment with `AddXmit` and `AddFlushedXmit` when creating entries. If you're building up a soup from static data, `AddFlushedXmit` may be a lot faster. If you're creating entries one at a time as the user enters data, the difference may not be as noticeable.

`EntryFlushXmit` and `EntryChangeXmit` are similar in the same way. `EntryChangeXmit` does an `EnsureInternal` on the cached frame, then writes the data to the soup. `EntryFlushXmit` skips the `EnsureInternal` step, writes the data to the soup, then discards the cached frame so that the entry must be read in again next time it's needed.

Use `EntryFlushXmit` if you need to keep a reference to the entry around for some reason, but have no plans to touch data in the entry for a while. The big win comes from avoiding `EnsureInternal`. Keep in mind that `EntryFlushXmit` doesn't actually reclaim the NewtonScript heap space used by the cached entry frame, it just removes the only reference to it. The garbage collector still has to do cleanup, the same cleanup that it would normally do if you no longer referenced the entry itself (that is, the fault block.) Making sure you don't keep references to unneeded entries or cursors may pay off more than trying to be tricky with `EntryFlushXmit`.

Other Caches

You may have noticed that each time you call `GetStores`, you get an array containing the same objects, one per store. That is, the OS doesn't create new store objects each time you make this call, but rather appears to return an existing object. This is hardly surprising; there are a lot of objects that exist even when your application isn't using them, like global variables, other applications, or the root view.

However, you may not have noticed that each time you call `store:GetSoup` or `GetUnionSoup` to get a particular soup, you also get the same object. Once again, the OS appears to return an existing object rather than create one each time you call the function. This also isn't very surprising, because clearly there is only one instance of a soup on a given store, and naturally you expect to get that one each time.

When you perform a `Query`, you get a cursor object. In this case, if you call the `Query` method a second time with the same arguments, you don't get the same object, but rather a new different cursor. Again, this makes perfect sense, each cursor needs to have its own "pointer" into the soup, and if you always got the same cursor for the same query, there would be no way to have them reference different entries.

When you call `cursor:Entry`, you get a soup entry. If you navigate some other cursor to the same place in the soup, and call `cursor2:Entry`, you get the identical entry object—the same fault block. This makes sense too, since clearly there's only one copy of the entry on the store. The OS gives the illusion that there is some entry object in memory, just waiting for someone to ask for it, and which will be given to anyone who asks.

As an aside, this is occasionally a problem for programming. If one application modifies an entry's cached frame, any other application that happens to be using that entry is suddenly working with a modified object, even though `EntryChangeXmit` or `EntryFlushXmit` wasn't called and no notification has yet been sent. The OS designers had to make a tradeoff between living with this behavior and the alternative, which would be to give each application a separate version of the entry and add a more complex database-like locking scheme to prevent multiple applications from modifying the same entry at the same time.

In a handheld single user device, it's unlikely that two applications will need to be modifying the same data at the same time, and so you can adopt a programming technique to avoid the problem. The technique is to make sure your application calls `EntryChangeXmit` or `EntryFlushXmit` relatively soon after modifying an entry. This is a good idea anyway, since a reset between when you change an entry and when you save it back to the soup would cause data loss. Built-in applications and applications based on the NewtApp framework typically save changes within a few seconds of modifying an entry's cached frame and when closing an editor.

Back to the mystery of identical objects. Clearly there isn't enough memory in the NewtonScript heap for the OS to really keep all the stores, soups, cursors, and entries around just waiting for someone to need them. Something special is going on behind the curtain to maintain this illusion.

The OS maintains independent lists of stores, union soups, soups, cursors, and entries that are in use. When someone asks for one of these objects, the OS first looks in its list to see if the needed object is present, and if it is, it returns that object. If the needed object isn't there, a new object of the proper type is created, tucked away in the list, and returned. These lists, or caches, are NewtonScript arrays.

The caches are more than just standard arrays, however. If they were normal arrays, then the references to the objects in the array would be enough to keep the objects themselves from being garbage collected. The OS would have to know when no other application needed the soup, store, cursor, or whatever and explicitly remove the reference from the cache.

The caches are implemented using a special NewtonScript object called a **weak array**. Weak arrays are just like normal arrays in most respects, with one important distinction. During garbage collection, if the only references to an object are in weak arrays, then that object is disposed of and the corresponding elements of the weak arrays are set to `NIL`. Weak arrays are documented in the Newton Programmer's Reference, and can be created using the global function `NewWeakArray`.

Here's a quick demonstration, from the NTK inspector. Note the contents of the array weenie change after garbage collection. The otherwise unreferenced string `"Atlas"` disappears, but the location string remains.

```
weenie := NewWeakArray(2);

weenie[0] := "Atlas";
weenie[1] := GetUserConfig('location).name;
weenie
#4418B25 [_weakarray: "Atlas", "Cupertino"]
GC();
weenie
#44146D1 [_weakarray: NIL, "Cupertino"]
```

Where the OS Caches Objects

Stores are not cached in a weak array, there is a real array of store objects maintained by the OS. The `GetStores` function simply returns this array. When a memory card is inserted a new store object is created and stored in the array. When the card is removed the stores in unmounted and the object removed from the array.

Inside each store object is a slot called `'soups` which contains a weak array of soups in use on that store. When the OS needs to use a soup on the store, a soup object is created and stored in this weak array in an empty position, or in a newly created element if there are no empty positions. Since it's a weak array, there is no worrying about when the soup is no longer needed, garbage collection takes care of the cleanup. The store methods `GetSoupNames` and `GetSoup` should be used to access soups in a supported way.

The OS maintains a separate weak array of union soup objects. There is no direct access available to this weak array from scripting. Again, garbage collection takes care of the cleanup.

Each union soup maintains a list of member soups in a slot called `'soupList`. This is not a weak array, since the member soups are a finite set and will be needed for as long as the union soup is needed. Note that this list may not contain a soup for each store, since member soups are not created until needed. The union soup method `GetSoupList` should be used to access this array in a supported way.

Each soup or unionSoup maintains a weak array of cursors that use that soup in a slot called `'cursors`. This cache isn't necessary for implementing the `Query` method, but it's part of how the OS manages to keep cursors up to date when the soup contents change. Again, since the cursors are kept in a weak array, garbage collection takes care of the cleanup. The soup method `Query` should be used to get a cursor in a supported way.

Soups maintain a reference to the store which contains them, in a slot called `'storeObj`, for use in various soup methods such as `RemoveFromStore`. The supported way to get at this is the soup method `GetStore`.

Each soup (but not union soup) also maintains a weak array of entry frames from that soup that are currently in use, in a slot called `'cache`. This cache is used to ensure that different cursors or different applications all read and write to the same cached entry frame. The only supported way to get a soup entry is via a cursor.

It's harder to tell what's in a cursor, since the implementation is all done in a C++ class and NewtonScript slots aren't used, but each cursor holds a reference to the soup or union soup which it's searching.

It's also harder to tell what's in an entry fault block, since that's also implemented in a C++ class and doesn't use NewtonScript structures,. However, each fault block also maintains a reference to the soup object that the entry is contained in. That soup together with indexing information about the entry is sufficient to allow the cursor to locate the real data when an entry needs to be faulted in. That same data allows the `EntrySoup` and `EntryStore` functions to be easily implemented.

Don't write production code that uses undocumented slots in stores, soup, union soup, or cursor objects.

As you can see, there is a lot of cross-referencing going on with the data storage model. All this cross referencing means that lots of data can be kept in memory by just a single reference. By forgetting to clean up a single reference to a single entry or cursor, you can force the soup and unionSoup objects to remain in memory.

If you've ever used `TrueSize` to try to get the space used by a given entry's frame, you'd have been surprised. It typically returns results that are much larger than expected. A quick test I did showed my card in the `"Names"` soup takes over 110K! Clearly that's not right. All this cross-referencing explains why.

```
e := GetUnionSoup("Names"):Query(
      {indexpath: 'sorton,
      startKey: "Ebert Bob"}
   ):Entry();
TrueSize(e, nil);
objects        431       121254
...
```

`TrueSize` follows references and knows how to look inside some kinds of C++ objects, like entry fault blocks and cursors, for contained references. This means that calling `TrueSize` on a soup entry actually counts the size of everything soup or store related that's currently in memory! The links are followed from the entry fault block to the soup, to the store to other soups on the store and from there to cursors and entries for those otherwise unrelated soups. If you want to know how big a cached entry frame is for a particular entry, just clone it before passing it to `TrueSize`. Note that the size of the cached frame in the NewtonScript heap is different from the size of the entry on the store. The global function `EntrySize` will tell you how much store space an entry requires.

```
TrueSize(Clone(e), nil);
objects        26        1002
...
EntrySize(e);
#6B4    429
```

You can make use of the various cross-reference lists to help track down unneeded references. By forcing a garbage collection then looking in the various lists you can easily tell if something in your application is referencing a soup, cursor, or entry that it shouldn't, because the item will appear in a list where you don't expect it to.

To easily track down where an unneeded reference exists, you can use the other feature of `TrueSize`, which is searching (nearly) everywhere for an object. If I wanted to find out what was hanging on to my names soup entry, for example:

```
TrueSize(nil, e);
...
 person           undo[0][0].receiver._proto
                 .realData.faultSoup.storeObj
                 .soups[4].cache[2]
 person           vars.e
```

This tells me that there's two places holding a reference to this entry, one is in a global variable e, which I expect since I created it. The other is in some weird place that I've never heard of before, but appears to be in one of the caches (in this case a weak array) in a soup that's reference by a store that happens to be referenced from some other soup needed for some undo action. (See, those cross-references are pervasive!)

My first thought on seeing this was that I should have forced a garbage collection first to get rid of the reference in the cache, which shows that even experienced programmers can have wrong thoughts. Forcing a GC wouldn't clear out the reference in the cache weak array, because the global variable has a "strong" reference to the entry. To really get rid of it, I'd clear out the global variable first, then force a garbage collect. Once that's done, there's no way to verify that the entry is really gone using only `TrueSize`, since there's no longer anything to pass to that function! To verify that nothing else is holding a reference I'd have to go poke around in those undocumented lists in the stores and soups, or carefully check free memory with `GC` and `Stats`.

Conclusion

Entries in NewtonScript are special objects and there is significant overhead involved in both reading and writing them from the user store. Careful thought while designing your applications will pay off by minimizing the NewtonScript heap space used and the time needed to access your data. You have control over when an entry's cached frame is or is not faulted in, and you can take advantage of this to improve performance.

Knowing how stores, soups, union soups, cursors, and entries relate to each other helps when creating efficient applications, and helps even more when tracking down performance or space problems. If you're ever unsure about how to optimize your application, experiment with different alternatives, measure the space and speed differences, and choose accordingly.

NTJ

Marketing Strategy

Why Newton Beats WinCE

by Don Davis, Apple computer

It remains to be seen whether Windows CE will *displace* or *validate* Newton appliances. Our view is that the vertical business needs bursts of infrastructure and the horizontal business needs definition and focus. We welcome the stimulation and energy that only someone like Microsoft can provide to these areas. As a result of their entrance to the marketplace, we see new opportunities for hand-helds in retail, corporate horizontal and vertical markets.

Our expectation is that all of these market slices will pick up and that CE will serve as a stimulus for all devices across several hand-held categories. We also believe that if we include the MessagePad 2000 and CE devices in a unified category (HPC, for lack of a better term), we may all stand to make it onto some corporate buying lists in the sub-laptop space, providing basic functionality at a much lower price point.

We have the strongest device. The industry media want to populate the HPC (handheld personal computers) category with only CE devices - seemingly because any machine made by Apple cannot be a "PC". But is a Windows CE device, really a PC? We think not. The MessagePad 2000 is at least an HPC in functionality and is only distantly related to the PDA category populated by PIM devices such as the US Robotics Pilot device. At a certain point these categories lose all meaning and something different like a "laptop replacement" or "companion device" delineation will likely evolve. But, with dual com slots, 162 MHz processing power, 24 hour battery life, 16 gray scale screen, TCP/IP, speaker and microphone, 8MB of ROM and 5MB of RAM, it doesn't matter what hand-held category we are in - we rule it.

So is CE better than Newton in other ways? Well, they do have better connectivity to the Windows desktop and they do have a shorter learning curve for Windows users. But what else? Easier to write drivers and software to Windows APIs? Yes, and we agree, so we are working toward that in Newton. But better back office connectivity? Don't jump to Microsoft Back Office yet, we think they will have trouble powering up LAN cards. Better Web browsing? How fast can it be? Black and white screens, little keyboard, one com slot . . . different vendors, different problems. Do they print yet?

With the MessagePad 2000 you can store and communicate at the same time. You can have a peripheral piece of hardware - like a laser scanner - in while you are sending data back to a wireless access point. You can receive pages and respond right back to them...on and on. We are giving you opportunities to truly enhance the capabilities of your applications not just offer them "in Windows". And they will be faster. Way.

Microsoft needs your help now, but when they get horizontal, that means bundled applications - Microsoft ones. They do like their licensing money. That is their thing you know. The vertical dollars are not big enough for them in the long run.

Microsoft is betting that people need a scaled back version of Word, Excel, Schedule Plus and Explorer for the road. If they can get the power management issue licked, they will probably do ethernet to push their network advantages. They want to stretch the desktop. That is where they have market share. If they placed the right bet they can sell large quantities into the channels of the hardware vendors and get on some of those corporate buy lists through resellers and major accounts managers with vendors like Compaq. This will lead, they believe, to a game that they know very well - stimulating market demand to expand the OS in order to drive hardware manufacturers to upgrades. Resell the market over and over again with expanded product offerings.

But does this model work for hand-helds? We wonder. Aren't people used to more from Windows than less? How much can you deliver to seasoned Win 95 laptop users with 25 - 40MHz and 2/4MB of RAM? That's a pretty little 4 gray scale screen and keyboard they have there. And there is another problem. The CE OS is already ahead of the hardware. This is the tradeoff for that magic $500 price point and it means there already needs to be a rev of the hardware just to get it up to the current CE capabilities.

We believe the early sales numbers from CE will dictate which hardware vendors stay in the game. If the horizontal market doesn't show up, hardware vendors will be far less inclined to reinvest in further hardware changes. To add more com slots, handwriting recognition, better power management, better screens and higher processing speeds will cost a lot of money. Some have probably not yet recovered their startup costs let alone one or two more revs of the hardware. Will the market run rate be high enough to get them all into the game? If it isn't where does that leave you?

To bridge the gap and provide interim wins, Microsoft will be encouraging you to partner with them on vertical CE activity. They know, after trying this with Windows for Pen Computing, that the market can fragment into vertical market niches with the lack of large corporate or consumer buys. Example, Pen Tablets.

Maybe you think this is a little of the pot calling the kettle black. After all we partner with you on vertical opportunities and are trying to bridge the gap to horizontal ourselves. And hey, we did some bundling of our own on the MP2000 and there is a price list we put together for the eMate release into education. So what gives?

Just one thing: those bundled applications and co-marketed solutions on Newton appliances are yours, not ours. And if you weren't one of the bundled applications for the MessagePad 2000 release or the eMate 300 price list, chances are we are trying to find you other opportunities to take advantage of. When horizontal takes off, you are part of the solution - - not just a means to an end. All of our future plans depend on you.

It is a hard time and simultaneously a great time for Newton development. This article was written to those of you who feel you must decide on supporting either Newton or Windows CE. If you are making that decision, we understand, but want to leave you with one final thought... We don't see you as our bridge over troubled waters. We see you as the land on either side.

Don Davis, Solutions Marketing Manager
Newton Systems Group

NTJ

New Product Technology

Speech Recognition for the MessagePad 2000

by Stephen Breit and Bent Schmidt-Nielsen, Dragon Systems, Inc. 320 Nevada St., Newton, MA 02160

INTRODUCTION

The MessagePad 2000 has two key ingredients which make Newton, for the first time, a viable platform for portable speech recognition. One is the powerful 160 MHz StrongARM microprocessor, and the other is high quality, 16-bit audio input. With the MessagePad 2000 as an enabler, the potential benefits of providing speech recognition on Newton devices are clear. First, in mobile situations, a speech user interface (SUI) can free one or both hands for other uses, such as handling material or operating equipment. Second, navigating a complex application can be much faster with speech than with a pen because speech input need not be constrained by the tree-like structure of a GUI. And third, as an input modality, speech is much faster than handwriting. The use of speech input on small devices is particularly compelling because they may either lack keyboards, or have keyboards that are too small to use efficiently. Or putting it more succinctly "Computers keep getting smaller, but our fingers stay the same size".

By now you are conjuring up visions of holding a Newton device and having it recognize and understand whatever you say. Providing such a capability is, of course, our ultimate aim, but we can safely say that this is at least a few years away. In the mean time, we can offer constrained speech recognition capabilities which will be a valuable addition to many applications. Users will be able to say phrases and sentences in a natural way, i.e. without pausing between words, but the vocabulary and word order will be constrained to a pre-defined grammar. For example, for an inventory application, the user will be able to say "Part number one five three seven", or, more generally, "<qualifier> <digit_string>", where <qualifier> might be "Quantity", "Part number", "UPC", etc. and <digit_string> is a series of 1 to 12 digits. Examples of other applications which might benefit from this type of speech recognition capability include medical record keeping, insurance appraisal, meter reading, rental car returns, sports data acquisition, and law enforcement. And there will be many others.

In this article, we describe our efforts to port one of Dragon's speech recognition engines to the Newton platform, and the capabilities which we expect to offer to Newton platform software developers. Since this is the first time that anyone has done speech recognition on the Newton, and speech recognition will undoubtedly be new to many readers, we begin with an overview of a typical speech recognition system. This provides a basis for understanding the capabilities of the speech recognition system that we have ported to the Newton. Next, we describe some of the challenges to doing the port, and how we have overcome them. Then, we describe a grammar for an inventory application. And finally, we provide some code snippets which illustrate how you include speech recognition in an application.

Overview of a Speech Recognition System

The majority of commercially available speech recognition systems rely on Hidden Markov Models (HMMs) and have the key components shown in **Figure 1**. To explain how this system works, we begin at the lower left with the microphone. The analog signal from a microphone is converted to a digitized wave form by the audio system hardware. This *audio input* is processed by the software *audio analysis* module which converts it to a form suitable for speech recognition. To do this, the audio analysis module applies a series of transforms, starting with an FFT, and outputs a *frame* of parameters every 20 ms. Typically, each frame contains 12 to 36 *parameters.* The audio analysis module also may apply a the speech detection algorithm to detect transitions from silence to speech, and from speech to silence. A collection of consecutive frames which begins and ends with silence is known as an *utterance.*

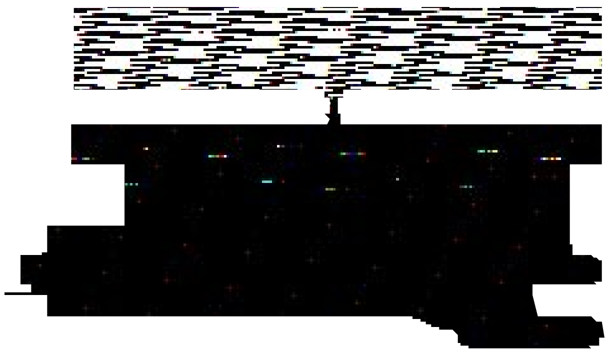


Figure 1. The components of a typical speech recognition system.

Before describing what the *recognizer* does with the utterance, we must describe the other key inputs. The *acoustic models* are built by collecting a set of utterances of similar sounds and assembling them into what amounts to a prototypical utterance for each sound. The individual sounds can be either elemental sounds called *phones* (hence *phonetic* modeling) or entire words (hence *whole-word* modeling). The accuracy of a speech recognition system is largely dependent on the quality of the acoustic models. Models that are intended for use by speakers who have not trained the recognizer are said to be *speaker independent*. Models that are intended for a specific speaker, and have generally been trained by that speaker, are said to be *speaker dependent*. Both types of models can be further *adapted* to a particular speaker, and this generally improves recognition accuracy.

If phonetic models are used, the *dictionary* provides a translation between word spellings and their pronunciations in terms of phones. If whole-word models are used, the dictionary simply provides a mapping between a word and its acoustic model. The dictionary and acoustic models are usually supplied with the speech recognition system, but it is up to the application developer to supply the *grammar.* The grammar defines the sequences of words that can be recognized. Some examples of grammars are given later in this article.

Simply put, the *recognizer* processes each utterance and returns the sequence of words that was most likely to have produced the utterance. Going into a bit more detail, the application specifies an active grammar, either from a predefined data structure, or by building it "on the fly". Then the application tells the audio analysis module to begin processing audio input. When the audio analysis module detects the start of an utterance, the recognizer begins its job. To start with, the recognizer hypothesizes all words that could have started the utterance based on the active grammar. It then scores each frame in the utterance against these hypotheses, and the score decreases with each successive frame. The score is the log probability that the acoustic model could have produced the observed utterance; thus, the hypothesis with the highest score had the highest probability of producing the observed audio input. As the recognizer scores successive frames against all active hypotheses, it prunes hypotheses whose scores are much lower than the best-scoring hypothesis in order to save computations.

The Speech API provides a well defined interface by which the application interacts with the recognition engine. The complexity of the Speech API depends on the recognition capabilities being offered. It may have as few as 20 entry points for simple command and control capabilities, or more than 200 calls to support dictation.

Speech Recognition Capabilities for Newton

With some background information in hand, we can now describe the capabilities of the speech recognition engine that we have ported to the Newton platform. Dragon has a number of recognition engines in its stable; we chose one that is known internally at Dragon as C-REC. C-REC was originally developed to run on a digital signal processor (DSP), so it has a small memory footprint, and the code is relatively small and portable.

C-REC can recognize phrases and sentences that are spoken continuously, *i.e.* in a natural way without pausing between words. Though C-REC imposes no restrictions on the size of the vocabulary or the grammar, it is best suited to "small-vocabulary" recognition tasks with simple context-free grammars. The vocabulary and grammar can be context sensitive. "Small vocabulary" in this case means that the grammar *perplexity,* or average branching factor, should be approximately 50 or lower in order to achieve real-time performance. The actual vocabulary size may be quite large (thousands of words). Higher perplexity grammars can be used if a response time of a second or more is acceptable.

C-REC works with either whole-word or phonetic models. To build speaker-independent whole-word models, we need one sample of each word from many (100 or more) different male and female speakers. This is a significant disadvantage relative to phonetic models where, once the models are built, any word for which we have a phonetic pronunciation can be recognized. On the other hand, experiments with C-REC on limited vocabularies have shown that whole-word models are more accurate than phonetic models. And whole-word models require less computation, at least when the active vocabulary is relatively small. Due to these considerations, plus some additional factors which are discussed in the next section, we chose speaker-independent, whole-word models for the initial port of C-REC to the Newton platform. We envision offering a software developer's kit which includes whole-word, speaker-independent models for a "standard" vocabulary. With a judicious choice of words, this standard vocabulary should be sufficient for a number of applications. If a particular application requires words that are not in the standard vocabulary, we will record samples of the additional words and build a set of custom acoustic models for that application.

We are inevitably asked about recognition accuracy. The answer to this question depends on many factors, including the size of the active vocabulary, the grammar, the quality of the microphone, the amount of background noise, and the quality of the acoustic models. Just to give some idea, we have estimated from test results that, in quiet conditions at least, continuously spoken strings of 5 digits should be recognized completely correctly 98% of the time.

Implementation on the MessagePad 2000

Next, we describe some details of how we ported C-REC to the MessagePad 2000. Our first step was to measure the quality of the audio system, since we have found a wide variation in the quality of the audio hardware on notebook computers. As we expected, the signal from the built-in microphone is not of sufficient quality to do accurate speech recognition. But we were pleased to discover that we got a high quality audio signal when we connected a head-mounted, noise-canceling microphone to the MessagePad via a small, custom-built amplifier and the line-in pins on the Newton connector.

C-REC is written in C, so we have been using a beta version of the Newton C/C++ Toolkit to port it to the Newton. The Newton C/C++ Toolkit does not allow any global or static variables. Therefore, we had to remove all such variables from C-REC, and store pointers to persistent data in a *binary object*. This proved to be a laborious process because the Newton C/C++ compiler only indicates that there are global symbols, but does not give their names.

Before porting the audio analysis module, we had to convert from floating-point to fixed-point operations because the StrongARM does not have floating-point instructions. We verified that there was no loss in recognition accuracy from this change. When we ran the fixed-point audio analysis on the Newton device, we found that it required only 6% of the CPU cycles. This was very encouraging because the audio analysis module must run continuously whenever the microphone is "turned on", whereas the recognizer runs only when there is speech input to be processed.

In order to port the recognizer itself, we had to find a way to store and load the acoustic models. One approach is to store the models in a binary object, and pass this binary object to the recognizer. Due to the NewtonScript garbage collection, the binary object would periodically move around in the memory system. This would require patching up pointers to data in the binary object each time the recognizer is called. An alternative approach, which was expedient and gave better performance, was to compile the acoustic models into the code. The disadvantages of this approach are that it takes a lot of memory to do the compilation, and that the code must be recompiled each time we change the acoustic models. As of this writing, we are compiling the models into the code. The memory required for the code alone is approximately 100 kB. For most applications, the additional memory required for the acoustic models will be between 250 kB and 1MB. During recognition, the recognizer requires an additional 250 kB of dynamically allocated memory.

A Speech Interface for an Inventory Application

Suppose you want to develop a speech-enabled application for taking inventory in a store or warehouse. Let's assume that four pieces of information are needed to inventory an item: 1) its location in the warehouse, expressed as a row number and a section number, 2) its 6-digit number, 3) its color, and 4) the quantity of identical items. You design a view which has fields for entering this information. You would like the user to be able to open a new inventory view by voice and enter data in the fields in any order.

To illustrate the design of the speech interface, we need to introduce a pseudo code for expressing a grammar. The grammar has three basic entities: *words, rules,* and *groups.* A word may be either a single word or a phrase consisting of two or more words. We denote a word simply be spelling out the word. We denote a phrase by putting underscores between the component words. If we want the recognizer to return text that is different than the spelling of the word, we enclose the return text in parentheses following the word. For example, "two(2)" indicates that the recognizer should return the character "2" when it hears a word that sounds like "two".

A group is a collection of words or rules. We denote a group by a comma-separated list of words or rules enclosed in curly brackets. We denote the name of a group by text enclosed in angle brackets. For example,

```
<global> = {next_item, previous_item, go_back, start_over,
enter_data};
```

defines a group named "global" in which any of the phrases listed on the right-hand side can be recognized. Each phrase counts as one word. We purposely called this rule "global" because these words will always be active in the inventory application. For example, the user will always be able to return to the previous field on the view by saying "go back".

To enter numbers between 1 and 99, it is useful define the following groups (we use ellipses to fill out an obvious sequence of words):

```
<one2nine> = { one(1), two(2), three(3),..., nine(9)};
<digit> = { zero(0), oh(0), <one2nine>};
<ten2nineteen> = {ten(10), eleven(11),
twelve(12),...,nineteen(19)};
<twenty2ninety> = {twenty(20), thirty(30),
forty(40),...,ninety(90)};
```

Now we need to introduce rules. A rule is a sequence of words and/or states. We denote the name of a rule by text enclosed in square brackets. For example, the rule

```
[twentyone2ninetynine] = <twenty2ninety><one2nine>;
```

allows users to say numbers between "twenty-one" and "ninety-nine", excluding multiples of 10. Finally, the group

```
<number> = {<one2nine>, <ten2nineteen>, <twenty2ninety>,
<twentyone2ninetynine>};
```

allows users to say, in a natural way, any number between one and 99. Note that the <number> group is defined in terms of previously defined groups.

For the inventory application, it will also be useful to define the group

```
<color> = {black, white, red, blue, green, yellow, orange,
purple}
```

and the following rules:

```
[location] = row <number> section <number>;
[part_number] = part_number
<digit><digit><digit><digit><digit><digit>;
[color] = color <color_name>;
[quantity] = quantity <number>;
```

Finally, we define a group which encompasses all of the rules for the inventory application:

```
<inventory_view> = {[location], [part_number],
        [color], [quantity], <global>};
```

When this rule is active, the user may enter data in any field on the form at any time, return to the preceding field if there is an error in it, clear the entire form, return to the previous item, or move on to the next item.

This example gives some idea of how you define a grammar. There are a total of 47 words in this example (each phrase counts as one word). We expect that all of these words will be part of our standard vocabulary.

PROGRAMMING EXAMPLE

This example shows how you incorporate speech recognition in an application:

Initializing the speech recognition system

The first step is to instantiate the speech recognition system.

```
SpeechRecog := {
// put additional variables or overrides here
 _proto: protoDragonSpeechRecognizer;
}
local theView := self;
SpeechRecog:setUp( theView, inputSource, errorCallBack );
```

The inputSource argument determines whether the system takes audio input from the built-in microphone, or line-input. You must provide a function for error callbacks. It is desirable to execute this code when your application starts up, otherwise the user may experience a brief delay while the systems allocates memory for the audio input buffers.

Defining the grammar

At this point, you need to define the grammar from which you want to recognize. We have not implemented the calls for defining a grammar in NewtonScript yet, so we cannot offer any code fragments here. Suffice it to say that there will be a set of calls for defining groups and rules as they are described in the previous section. And there will be calls for iterating the words in the dictionary, and checking whether a particular word is in the dictionary.

Starting the audio analysis module

The next step is to start the audio analysis module:

```
SpeechRecog:startAudio();
```

This could have been done transparently by `setUp()`, but there is a good reason for giving you control over when it starts. Once started, the audio analysis module must run continuously. This draws power and, more importantly, overrides the power-down features of the MessagePad 2000. You need to start the audio analysis module well before you expect speech input, because it takes almost a second for the MessagePad 2000 to charge up a capacitor in the audio system hardware.

Starting and stopping the recognizer

You define one or more callback functions for processing the results from the speech recognizer. For example, you may have a different callback function for each view or context. The transcription parameter is a text transcription of what was said (it must be permitted by the grammar, of course). The grammarInfo parameter is a reference to a NewtonScript frame which allows you to determine which rule was recognized.

```
resultCallBack := func( transcription, grammarInfo )
 begin;
 setValue(resultView, 'text, transcription);
 end;
```

You start the recognizer:

```
SpeechRecog:startRecog( resultView, resultCallBack,
grammarGroup );
```

Once started, the recognizer starts recognizing whenever a new utterance is available and does a callback when it is finished recognizing the utterance. You must stop the recognizer when you want to change the grammar or as the first step in the process of shutting down the speech recognition system:

```
SpeechRecog:stopRecog();
```

Stopping the audio analysis module

After shutting down the recognizer, you shut down the audio analysis module:

```
SpeechRecog:stopAudio();
```

Shutting down the speech recognition system

Finally, unlike an ordinary NewtonScript program, you must free the resources used by the recognition system before assigning nil to the Recog variable:

```
SpeechRecog:shutDown();
SpeechRecog := nil;
```

CONCLUSIONS

As a result of work by Bent Schmidt-Nielsen, Maha Kadirkamanathan, and Shaun Keller, a small-vocabulary, continuous-speech recognizer is now running on the Newton platform. As of this writing (early February '97), it recognizes strings of continuously spoken digits and returns a text transcription within one-half second after the user has finished speaking. It is very exciting to get this responsiveness from a computer that is powered by a few AA cells! In the immediate future, we plan to supplement the digits vocabulary with a set of standard words and phrases such as those listed in the inventory application example. Further down the road, we will consider adding other features such as phonetic models, and the ability to adapt the speaker-independent models to an individual speaker.

Judging from the enthusiastic response we have already received from Newton developers who have heard about our efforts, there is a lot of pent-up demand for speech recognition on the Newton. We are planning to offer a tool kit which will enable Newton developers to speech enable their applications. The tool kit will include an AutoPart, a user "proto", documentation of the NewtonScript API, a microphone, and a preamplifier. Built into the AutoPart will be acoustic models for a standard vocabulary. If the standard vocabulary does not meet the needs of a particular application, we are prepared to develop custom acoustic models for that application. Beyond the inventory application that is suggested in this article, we believe that the type of capability we have described will be useful for many other applications. We look forward to hearing your requirements for speech recognition on the Newton platform.

NTJ

Surviving the Grip

by Bob Ebert, Newton Developer Advocate, ebert@newton.apple.com

Note: This is a follow-up article *to"Newton still needs the card you removed"* by Mike Engber, first published in February 1994 in Double-Tap magazine.

Introduction and Terminology

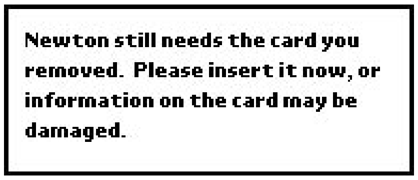

Newton still needs the card you removed. Please insert it now, or information on the card may be damaged.

Figure 1. (you blew it)

If you see this slip, you have encountered the **Grip of Death** (aka **G.O.D.** or **the grip**). This happened because something has **touched** the card while it was being **unmounted**, causing the unit to require the card back to resolve some reference. If the reference isn't caught when the card is unmounted, it becomes a **bad pkg ref** which can cause throws at a later time.

Briefly, the grip happens when the OS follows a reference to an array, frame, binary object, symbol, frame map, VBO, or soup data on a card while the card is being unmounted.

Grips happen when you touch data on a card while the card is being unmounted.

That isn't enough of a description for you to understand the problem or why it occurs, so read the*"Newton still needs the card you removed"* article. (You may want to read it two or three times.) This article assumes the reader is familiar with the common causes and workarounds for the grip. The original article is currently available on-line at
`<ftp://ftp.apple.com/pub/engber/newt/articles/NewtonStillNeedsTheCard.rtf>`
and on the Newton Developer CDs in Technical Information:Newton 1.x-related ARCHIVES: ARCHIVE - 1.x Articles:NewtonStillNeeds article.

Since that article was written, we've undergone one major and several minor revisions to the Newton operating system. With the 2.0 (and later) OS, many of the things that used to be common causes of the grip are now easily avoided. For example, most of the OS registries now ensure that the necessary elements of what's being registered are stored internally so that the corresponding unregistration can be done without touching the card or leaving bad references.

There have been many other changes and enhancements added to making working with PC cards easier. Smarter registries, soup definition frames, better package handling, atomic store operations, better cleanup, and debugging hooks make tracking down and avoiding the grip easy and almost convenient. This article discusses those changes and enhancements.

The most recent devices from Apple now have multiple PC card slots. The support has been in the OS since 2.0, but never before has hardware with multiple slots been available. This enhancement does not complicate the card removal process as far as your applications are concerned. The only thing to note is that removing one card can cause the OS to touch the remaining card, which isn't a problem unless you happen to eject both cards at the same time, in which case the OS itself will give you the grip on one of the cards.

Better Notification

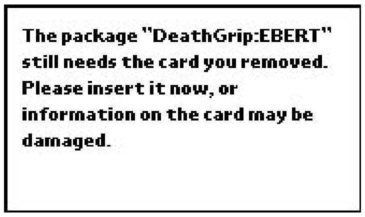

The package "DeathGrip:EBERT" still needs the card you removed. Please insert it now, or information on the card may be damaged.

Figure 2. (I blew it)

In the older versions of the OS, only the slip from **Figure 1** would appear, and this slip still appears when there is no information available about the package that caused the problem. The slip in **Figure 2** is displayed on 2.0 and later units when the OS can determine which package caused the problem. It's intended to provide a clue to the user about which package must be removed to fix the card. Don't let it be yours!

Removable/Remountable Cards

In 1.x, if you got the grip you were stuck in a classic "lather, rinse, repeat" loop. The Newton device would not proceed until you reinserted the card. On reinsertion, the OS would finish the unmounting process, then immediately remount the card. Remounting the card reintroduced the grip. The only way out was to reset the unit. In 2.0, the OS stops after unmounting the offending card and gives you a chance to remove it safely, a significant improvement.

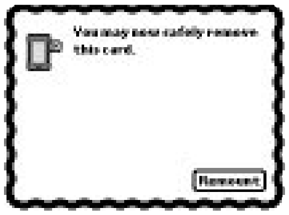

You may now safely remove this card.

Figure 3. (Newton saved you)

Bad Package References

After a package is deactivated, the 2.0 OS replaces any references to data in the package with a special object, called a bad package reference. This is done by searching the NewtonScript heap, and doesn't take very long. (It's roughly the same as garbage collection.)

This is an important change. In the 1.x OS the invalid reference would remain unchanged, and could be used later. If the user were lucky, they'd get a -10401 "Bad package" error when this occurred. If they were unlucky, some other data would happen to be at that location in memory and the result would be unpredictable.

On 2.0 and later, using a bad pkg ref typically causes a throw of an `evt.ex.fr` exception with the error code -48221: "Reference to deactivated package." Depending on what operation is being done with the bad pkg ref, other errors are possible. For example, trying to find the length of a bad pkg ref will generate an `evt.ex.fr.type;type.ref.frame` exception with error code -48418: "unexpected immediate." The string `<bad pkg ref>` will be displayed in the `'value` slot of the exception frame, which provides the needed clue that it's a card problem.

The change makes it much more likely that you'll notice immediately when using a bad pkg ref, so it's now much easier to find out what your application has done to cause the problem. Typically simply setting `breakOnThrows` to `TRUE` and doing a stack trace when the error occurs will tell you enough to find and fix the problem.

-48221 errors or any error involving a <bad pkg ref> means you have a grip related problem where the OS simply didn't notice the bad reference until later.

Suppressing Package Activation

Occasionally during development you'll create a package so horrible it resets or locks up the unit during installation. With 1.x, you'd either have to have had the foresight to put the package on a card during debugging and then erase the card by having the prefs application open when it was mounted, or you'd have to do a hard reset, erasing all data on the internal store.

With 2.0 there's a way out. Hold down the pen along the left edge of the mostly-black splash screen during a reset and you'll get a chance to prevent packages on the internal store from being installed, or hold the pen there while inserting a card to mount the card without activating any packages on that card. Once this is done, you can simply delete the problem package and preserve the rest of your data.

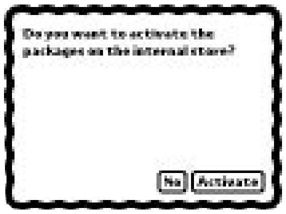

Do you want to activate the packages on the internal store?

Figure 4. (phew)

By "along the left edge" we don't mean all the way against the edge of tablet, but rather over the display area, within about 1/4 inch from the edge. Holding down the pen along the top edge of the screen not only allows you to avoid mounting packages on the internal store, but also resets the backdrop application to Notes.

Hold down the pen along the left edge of the screen during reset or card mounting to prevent activation of packages.

Open prefs then insert a card to erase the card without touching any data on it.

Hold down the power key while resetting to erase the internal store.

Locking a Package

When an application on a card is open or in use, ejecting the card is guaranteed to cause the grip. During card unmounting, packages on the card are deactivated. Deactivating a package executes the `RemoveScript` for each part in the package, and also closes the base view for any form parts in the package. If nothing else, the system searches your base view's `_proto` chain for the `Close` method, and this will cause the grip.

This is fairly safe, and users quickly learn to close applications before ejecting a card. It's not practical to work around this, so don't try. Any workaround would require trying to `EnsureInternal` enough of your application so that it could be closed without causing the grip, and this would be a total waste of precious NewtonScript heap space.

However, this process does mean your application may be closed in unusual circumstances, even when it's the backdrop or doesn't provide a close box. For certain operations this may be a problem; live connections may be have to be aborted or gathered data may need to be discarded. To help avoid this, the 2.0 release of the OS has support for reserving a package, and the store on which the package exists.

Reserving a package is done by calling `MarkPackageBusy`, giving a reference to the package along with some text information to be shown to the user if necessary. `MarkPackageNotBusy` cancels the reservation. When a package is reserved, the user gets an extra warning if they attempt to delete it, freeze it, or move it to another store. These functions are documented in the Newton Programmer's Reference.

Reserving a package effectively reserves the card that contains the package. Attempting to eject such a card does something new. The "Newton still needs the card you removed" slip still appears, but with the reserving application's name as provided to `MarkPackageBusy`. Unlike during normal unmounting, this slip appears immediately when the card is unlocked, and processing is suspended until the card is reinserted. On reinsertion, processing picks up right where it left off. The card is not unmounted, the applications are not closed, and as far as most operations in NewtonScript are concerned, nothing happened. The card is effectively "locked" into the unit.

The OS provides a way to lock a store briefly so that it may be used without fear of being ejected. The store method `BusyAction` will execute a call-back function with the store reserved until the call-back completes. Ejecting the card while the call-back is executing produces the same "locked" result.

If you are designing robust applications, you might also check out the store method `AtomicAction`. This executes all the store-related operations in a call-back function as an atomic unit, so that if any failure (such as a store full error) occurs during the call-back the store is left in the same state it was in before the call-back started.

You may wonder why all apps don't simply reserve their packages when they open and unreserve them when they close. When a user ejects a card, it's generally their goal to get the card out of the unit. If the card is locked by your application, the user would reinsert the card, close the offending application, and then eject the card again.

Not reserving the packages is cleaner. Consider what happens if two applications are open. Only one produces a grip message when the card is unlocked. The user reinserts the card, closes that app, ejects the card again, and then the second application produces a grip message. Repeat as necessary, and it quickly gets annoying. Not locking the packages means the user is notified only once. After reinserting the card, package deactivation continues grip-free for all remaining packages.

A rule of thumb is to mark packages busy for as brief a time as possible, to minimize the inconvenience to the user. Only lock a package if the inconvenience caused by removing the package or closing the application is greater than that caused by having to remount the card and manually abort some operation.

Mark packages or stores busy only when necessary to avoid user inconvenience.

Tracking Down the Grip

When the grip is encountered while unmounting a store, the 2.0 OS tucks away a reference to the offending data. With the help of a debugging package written by Mike Engber, you can get this reference, and find out exactly what it was that caused the problem.

This debugging package is called FindGOD, and the package is available on-line at `<ftp://ftp.apple.com/pub/engber/newt/FindGOD.sit>` and on the Newton Developers CD (#12) in Tools:The UN Files:FindGOD.

Using it is fairly simple: install the FindGOD package on the internal store, yank the card to reproduce the grip, reinsert and then remount the card, then call `FindGOD()` from the NTK inspector. The object that caused the grip will be displayed, which typically will tell you exactly what you need to fix to avoid the problem. The README that comes with FindGOD gives more details, so read it.

Use FindGOD to quickly track down the cause of the grip.

Conclusion

The 2.0 OS vastly improves the situation with respect to the grip of death conditions. You still need to be careful when writing your applications, but at least now if you encounter the problem, tracking it down is easier.

Read the original "Newton still needs the card you removed" article.

NTJ

Newton Systems Group Profiles

Meet Debbie Carlton

Director of Marketing, Newton Systems Group

Since I accepted the position of Director of Marketing for the Newton Systems Group in the fall of 1996, I have been focused on rebuilding the marketing organization, putting all of the marketing mechanisms in place to ensure a successful launch of the Message Pad 2000 and eMate 300, developing a product road map for future Newton products, and creating program plans to support our partners. My vision for Newton-based products is to create a family of hardware products and a single, robust operating system from which licensees and Apple can build complete customer solutions. One of the key strategies I am implementing to reach this vision is to understand via focused customer research, the unique problems that our customers experience related to mobile computing. With the customer at the center of our design targets, we will apply the most innovative designs and state-of-the-art technologies to develop winning products.

A key element in understanding our customers' needs and experiences is gathering data on the solutions they want to employ. I believe that our developers are critical to our products' success in the market. Without the availability of top notch applications and solutions, the MessagePad 130, the MessagePad 2000, the eMate 300 and all future Newton products will not provide complete solutions to customer problems. For this reason I have made it a priority in Newton Marketing to create robust developer tools as well as supportive developer and VAR/SI programs.

With a customer driven product road map coupled with robust solutions, the last leg of the marketing equation is promotion. To this end we are designing new advertising campaigns for the MessagePad 2000 and eMate 300 aimed at the mobile business professional and K-12 institutions, respectively. We've had our PR engine working hard since October creating interest in the marketplace and communicating the key customer benefits of our products over the competition. PR will continue to be central to our outbound marketing efforts given the cost effectiveness of this medium. We are constantly looking for newsworthy items related to customer adoption of the Newton technology, innovative implementations of our products, new software applications and solutions, and awards and prizes earned based on our own engineering efforts as well as those of partners and licensees. We are also enhancing our web presence and want to develop links between the Newton web site and sites of our developers, VARs/SIs and licensees. Finally, we are redesigning our merchandising and collateral materials and participating in a variety of trade shows around the world to communicate the messages about our exciting new products.

We look forward to a timely and successful launch of the MessagePad 2000 and eMate 300 into our distribution channels. We plan to maintain interest and generate demand for our new products by a combination of all outbound marketing elements. Finally, we plan to add other exciting Newton products to our portfolio later in 1997 and1998.

NTJ

Developer Training Update

Newton Developer Training has expanded to offer you new options. In addition to Newton Essentials, we are offering Newton Communications and web-based training - and we're taking our courses on the road. With the exciting upcoming release of the Apple eMate 300 and the Newton MessagePad 2000, now is the time to take advantage of the Newton 2.0 Developer Training offered through Apple Developer University, Arroyo Software, and Calliope Enterprises. This training will cut your learning and development time while providing you the expertise needed to write useful and powerful Newton applications. See schedule and registration info below to sign up! Also, be sure to check out our new free online training courses at our Newton Developer website:
`<http://devworld.apple.com/dev/newton/devservices/nsgtraining.html>`

Newton Programming: Essentials

Learn how to develop applications for Newton devices using the state-of-the-art development environment, Newton Toolkit for Macintosh and Windows, in combination with a very powerful, small, robust, object-oriented language, Newtonscript. The use of Newton Toolkit lets you interactively develop your applications without having to execute sequential edit, compile and link cycles. In addition, human interface guidelines for developing on PDAs are discussed. Duration: 5 days

Newton Programming: Communications

Learn the fundamentals of Newton Communications. Using a self-paced mentored approach, we cover information for Newton programmers who want to add communications code to their applications. Students will have access to code, articles, references and labs for use as they desire. An qualified instructor is available to work one-on-one with anyone having specific questions or problems. In addition, a module on the Newton Internet Enabler will be presented. Duration: 5 days

Schedule

Class	Location	Date	Cost/ Student	Registration
Comms + Essentials	Ann Arbor, MI	4/21/97	$1500	313-439-3828
Comms + Essentials	Ann Arbor, MI	5/19/97	$1500	313-439-3828
2.1 Essentials	Cupertino, CA	5/12/97	$1500	408-974-4897
2.1 Essentials	Cupertino, CA	7/7/97	$1500	408-974-4897
2.1 Essentials	Cupertino, CA	9/8/97	$1500	408-974-4897

Arroyo Software and Calliope Enterprises are both long-time providers of Newton programming training for Apple. Both companies provide instructors who are experienced Newton programmers as well as expert trainers. Arroyo Software authored the Newton Communications Course and Calliope Enterprises authored the Newton Essentials Course, both for Apple Computer, Inc. Educational and group discounts are available.

Onsite training can be arranged both within the United States and worldwide. Further information about both companies is available upon request.

Arroyo Software	Calliope Enterprises
214 West Main Street	700 East Redlands Boulevard, Suite 154
Milan, MI 48160	Redlands, CA 92373
(313) 439-3828	(909) 793-5995

```
sobel@arroyosoft.com
nrhodes@pobox.com
http://www.arroyosoft.com/
http://www.pobox.com/~neil/training.html
```

NTJ

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

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.

New: Newton Associates Plus Program

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

Newton Associates Plus Program Features:

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

Annual fees are $500.

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

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

AppleLink: DEVSUPPORT

Internet: devsupport@applelink.apple.com

Newton