Newton® Technology JOURNAL

Volume II, Number 3 June 1996

Inside This Issue

Communications Technology

Newton Internet Enabler

by Gary Hillerson, Hillysun Enterprises, Inc.

The Newton Internet Enabler makes it easy for you to develop applications that access the Internet. With Newton Internet Enabler, you establish a link to an Internet provider, configure your link with options, and use communications endpoint methods to send and receive data. The Newton Internet Enabler even automates status display and data conversion for you.

You can establish a link that uses one of two transport services: either TCP or UDP. Several applications can share the use of a link, and the Newton system software keeps the link open until all applications release the link. Newton Internet Enabler currently supports the use of two lower-level link protocols: SLIP and PPP.

Newton Internet Enabler provides an application programming interface and a setup application. The setup application, *Newton Internet Setup*, allows users to define the configuration for links to various Internet providers. For example, a user might set up a link configuration for Compuserve, another configuration for a local Internet provider, and a third configuration for checking email at work. Each link configuration includes information about the phone number to dial, the link-level protocol to use, and the initialization and login sequences for establishing the link.

The Newton Internet Enabler application programming interface (API) consists of about ten global functions that you can call to perform various net session-related tasks, a number of options that you can use to control the

New Technology

Newton Toolkit Does Windows

by Lee Dorsey, Apple Computer, Inc.

In an effort to expand the number of software developers who can develop for the Newton 2.0 platform, Apple is delivering the first release of Newton Toolkit for Windows, Version 1.6. This product provides complete Newton Toolkit functionality, including the compiler, profiler, and full support for Newton 2.0. It runs on Windows 95, Windows NT, and Windows 3.1 with Win32s with a fully native Windows look and feel.

"Shipping Newton Toolkit for Windows is a very important step in providing our developer community with the most flexible, powerful tools of any PDA platform," said Rick Fleischman, Newton Tools Product Line Manager. "As the Newton platform is maturing, Newton Toolkit for Windows allows us to reach out and address a much larger base of software developers who want to create Newton applications."

Newton Toolkit for Windows is only one example of the many improvements that continue to be made to developer tools for the Newton platform. In January, Apple shipped both Newton Toolkit for Mac OS, Version 1.6 and the Desktop Integration Libraries for Mac OS and Windows, Version 1.0. Newton Toolkit 1.6 is optimized for PowerPC, delivers improved debugging tools, and provides full support for Newton 2.0. The Desktop Integration Libraries (DILs) allow desktop application vendors for both Mac OS and Windows to directly synchronize data between their applications and data on a Newton PDA, without the use of any

continued on page 21

Newton Technology Journal

Volume II, Number 2 *June 1996*

Published by Apple Computer, Inc.

Lee DePalma Dorsey • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

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

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

Contributors
Greg Christie, Barney Dewey, Gary Hillerson, Peter Murray, Jeffrey C. Schlimmer, Eileen Tso

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

John Kawakami • *Editorial Assistant*

Matt Neuburg • *Editorial Assistant*

Judith Chaplin • *Art Director*

Editor's Note

Letter From the Editor

by Lee DePalma Dorsey

Same Time, Same Place, New Stuff

Many of you will be reading this issue of the *Newton Technology Journal* from your seats or hotel rooms in San Jose, CA during Apple's World Wide Developers Conference and some of you will be looking at it for the very first time. We hope that most of the Newton Developer Community has decided to attend Apple's annual developer event to learn more about Apple's technical directions and spend time learning about new technologies. As always, the Newton Systems Group will be on hand to introduce Macintosh and Windows developers to Newton technologies and the potential the platform holds as we grow the PDA product category and technology base. We'll also show attendees how Newton technology fits into the rest of Apple technologies and integrates into the company's business plan moving forward. And, for the experienced Newton developer, there will be new information and demonstrations of some of the latest Newton platform technologies that are covered in this issue.

Whether you're an experienced Newton programmer, a developer who's new to the Newton platform, or just evaluating the technology, you could not be involved in a more exciting technology at a more exciting time. As we have been demonstrating since its introduction in November, 1995, Newton 2.0 is a huge step forward in delivering the kind of functionality that users and developers alike expect from a PDA platform. This spring, we are seeing and hearing so much more about the internet, e-mail and communications. And Newton PDAs play right into this timely evolution. This issue of *NTJ* will run you through some of the most important aspects of the evolution: Barney Dewey, Product Manager for Newton Communications, takes you through our communications strategy and Gary Hillerson walks you through the latest tool you've all been waiting for – the Newton Internet Enabler. Finally, Eileen Tso will provide you with her take on how and why these are critical to the continued growth of the Newton platform and the solutions that make it cutting edge. On the tools front, Windows NTK is finally ready to go and attendees at the Intro to Newton Session will get first peeks at it. We talked about it last year, and now it's here. Same time, same place, but lots of new stuff.

While sessions at the WWDC are aimed at educating new and interested developers in the platform, there is always something challenging and new for the experienced developer. And, if you are already a platform convert, take this opportunity to convince a friend or two to stop by the Newton sessions to learn more about the possibilities of extending Mac applications to mobile Newton clients and getting involved in the first real "mobile internet" device on the market. As Guy Kawasaki says, "let a thousand flowers bloom". Each one of you is and can be an evangelist for this platform. WWDC is a great opportunity to convince your fellow developers to deliver their latest great ideas on this platform. The time is right, the market opportunities are there, and the tools are there. Show them what you already know about the premier PDA platform and the strides Apple has taken with Newton 2.0.

Lee

continued from page 1

Newton Internet Enabler

configuration of your links, and a communications tool that you use with your endpoints to perform communications on the Internet.

Figure 1 shows the relationship of the Newton Internet Enabler components.

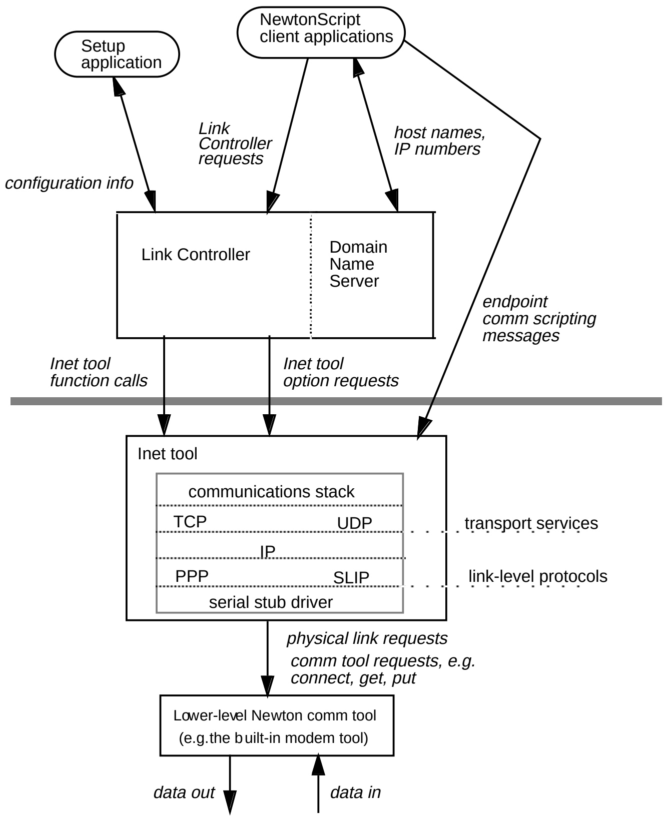

Figure 1 Newton Internet Enabler components

The Newton Internet Enabler application programming interface provides two kinds of functions: domain name service functions and link controller functions. You use the domain name service functions to translate between Internet domain names and their corresponding IP addresses. You use the link controller functions to establish, release, and find the status of your Internet links.

The domain name service and link controller functions are global functions in the Newton system software. These functions provide an interface between Newton communications endpoints and the Inet tool, which is the underlying communications tool that performs the actual work of establishing, maintaining, and communicating over the links.

The Inet communications tool provides a configurable stack of protocols at and below the TCP/IP level. The Inet tool is a standard Newton communications tool, which means that it provides all of the endpoint services that are provided by other built-in communications tools, such as the built-in modem tool and the built-in serial tool. Like the other communications tools, you can control the configuration of the Inet tool with communications options.

The Inet tool can establish physical links using various low-level communications services. Each communications service is provided by a Newton communications tool such as the built-in modem tool. The Inet tool can run various link-level protocols that are provided with the Newton system software, including PPP and SLIP.

Your application can use several endpoints with the same Newton Internet Enabler link. Each endpoint, however, requires a significant amount of memory. The total number (for all applications) of endpoints that can be active is restricted by a combination of the user's hardware configuration and which software is currently in use on the device.

The remainder of this article describes the Newton Internet Enabler application programming interface. The November, 1995 issue of *Newton Technology Journal* (Volume 1, Number 5) provides an overview of Newton communications technology, including a discussion of endpoints and communications tools.

NEWTON INTERNET ENABLER AND CALLBACK FUNCTIONS

Many of the Newton Internet Enabler functions require you to provide a callback function, which is a function that the Inet tool calls during and/or after the performance of the operation that you requested. The callback function receives status and error information.

For example, the `InetCancelLink` function calls the callback function that you provide after it finishes its operation. Your callback function for `InetCancelLink` can determine if an error occurred and can determine the current status of the link that you wanted canceled.

Some operations call your callback function more than once. For example, the `InetGrabLink` function calls the callback function you provide many times during its operations. You can use your `InetGrabLink` callback function to monitor the progress of the grab, since each call to it provides you with the current status.

When a function requires that you specify a callback function, you do so by providing a context frame and the symbol of the function defined in that frame that you want to use as the callback function. For example, the `InetGrabLink` function takes three parameters and is declared as follows:

```
InetGrabLink(linkID, clientContext, clientCallback);
```

When you call `InetGrabLink`, you must specify a frame (or your application frame) as the value of `clientContext`, and you must specify a function defined in the frame as the value of `clientCallback`.

You might create a callback function for your `InetGrabLink` calls that looks like the following:

```
myApp.GrabLinkCallback := func(linkID, stat, err)
  begin
  if err=nil and stat.linkStatus <> 'connected then
    ;     // display status
  if err then
    ;     //handle the error
  // link established, so resolve the address
  end;
```

Then, when you call the `InetGrabLink` function in your application, you pass it the name (symbol) of your callback function. For example:

```
myApp.TestGrab := func()
  begin
  myStatusView := InetStatusDisplay(nil, nil, nil);
  InetGrabLink(nil, self, 'GrabLinkCallback);
  …
  end;
```

This function first calls the `InetStatusDisplay` function to create and display the status view. The call to `InetGrabLink` uses the default link ID and specifies self (the application frame) as the value of the `clientContext` parameter, and `'GrabLinkCallback` (the symbol for the callback function) as the value of the `clientCallback` parameter. The `GrabLinkCallback` function will be called repeatedly while the system is attempting to grab the link, until either the status is `'connected` or an error occurs.

Using the Link Controller Interface

You can use the Link Controller to create and manage a link between a Newton device and the Internet. The Link Controller can manage a single link for multiple applications simultaneously. This means that one application establishes the link and other applications use the same link.

To establish (grab) a link, you call the `InetGrabLink` function. The first grab of a link can be expensive in terms of time: typically, the Inet tool software dials the Newton modem and negotiates the connection to establish an Internet session. The Inet tool then performs whatever login and initialization procedures are required, which the user has configured with the Internet Setup application. All of this can take a substantial amount of time.

Since it can take so much time to grab a new link, Newton Internet Enabler makes it easy for another application to grab a link that has already been established by maintaining a reference count of users for each link. Whenever an application grabs a link, the link controller increments its count of users of that link. The physical link is dropped only after all users have released the link (when the count becomes 0).

Most of the link controller functions operate asynchronously. After the Inet tool initiates the operation, it returns control to your application. When the operation is complete, or in some cases while the operation is in progress, the Inet tool notifies you by calling a callback function that you have provided. You can use your callback functions to monitor the status of a link and to determine if an operation was successful or not.

While the grab of a link is in progress, the Inet frequently calls your callback function, providing it with information about the current status of the link. You can send this status information along to the `InetStatusDisplay` function to provide visual feedback to the user about the connection process.

The following is an example of a typical flow of operations that occur during an Internet session using Newton Internet Enabler:

1. An application (My_Application) issues a call to the `InetGrabLink` function. The link controller dials the modem and begins an Internet session with an Internet provider.
2. While the grab operation is in process, the Inet tool periodically calls the callback function that My_Application has defined for grab operations. This callback function calls the `InetStatusDisplay` function to update the on-screen display of the current status of the link.
3. When the grab operation completes, the callback function removes the status display from the screen.
4. My_Application instantiates and binds one or more endpoints to use over that link. Each endpoint can use either the TCP or UDP transport services, and each endpoint can be bound either to initiate an outgoing connection (connect) or to listen for an incoming connection (listen).
5. My_Application uses its endpoint(s) to perform communications operations, calling endpoint methods such as `Output` and `SetInputSpec`.
6. Another application (Your_Application) issues a call to the `InetGrabLink` function to use the same service provider as My_Application. The Inet tool returns the same link that it established in step 1.
7. Your_Application creates and uses endpoint(s) to perform communications operations.
8. Your_Application finishes its use of the link and calls the `InetReleaseLink` function. The link controller decrements its count of users of the link.
9. A third application (Their_Application) grabs the link, creates endpoints to use over the link, and releases the link.
10. My_Application finishes its use of the link and calls the `InetReleaseLink` function. The link controller decrements its count of link users. The count becomes 0, so the link is dropped: the Internet session ends, the modem is hung up, and any resources used for the link are released.

Grabbing a Link

To get started, you need to establish (grab) a link. To establish a link, you need to call the `InetGrabLink` function. You need to provide `InetGrabLink` with a link ID, a callback function, and a callback context frame:

```
InetGrabLink(linkID, clientContext, clientCallback)
```

For the link ID, you can tell `InetGrabLink` to use the default link by using `nil` or you can use an identifier returned by the `InetAddNewLinkEntry` function as the value of this parameter. When you specify `nil`, the system software uses the link ID that has been established as the default link ID, which is almost always what you want to do.

The `InetGrabLink` operation can take some time to complete. While it is in progress, the Inet tool repeatedly calls your callback function to report the current status of grabbing the link. The Inet tool calls your callback function until either an error occurs or until the status becomes `'connected`.

The status value in your callback is a status frame. This frame contains the current link status value and (possibly) other information. In your callback, you can use the `InetDisplayStatus` function to show the current status to the user. The next section, “Retrieving and Displaying Link Status Information,” describes how to display status to the user.

Here is an example of a callback function for the `InetGrabLink` function:

```
myApp.GrabLinkCallback := func(linkID, stat, err)
  begin
  myLinkID := linkID;                    // save the link ID
    // during grab processing, display status & return
  if err = nil and status.linkStatus <> 'connected then
    return InetDisplayStatus(linkID, myStatView, stat);

    // at this point, either we are connected or we
    //  got an error, so close the status display
  InetDisplayStatus(linkID, myStatView, nil);
```

```
if err then begin
   print("link failed");           // handle the error
   :endGrabLink(err);              // end connect attempt
   end

else        // status.linkStatus = 'connected, so resolve name
   DNSGetAddressFromName("apple.com", self, 'DNSCallback);
end;
```

The first statement, `myLinkID:=linkID`, saves the ID of the link that `InetGrabLink` is in the process of grabbing in one of your variables. You might want to store the link ID for use in other portions of your application.

If grabbing of the link is progressing without errors, your callback function gets called to report the progress. You can call the `InetDisplayStatus` function, as shown in the above example. The `myStatView` view used in the this example was created before the grab of the link was initiated.

The grab of the link terminates when the connection is made or when an error occurs. In either case, you can remove the status display view at that point. To do so, call the `InetDisplayStatus` function with `nil` as the value of the status parameter.

If `InetGrabLink` encounters an error, the error code will be a non-zero value and your application has to do something with that error. In the example function, a message is displayed and the connection attempt is terminated.

If `InetGrabLink` succeeds, the callback receives `'connected` as the value of `linkStatus`. At that point, you can perform any operations that are appropriate. The example function takes this opportunity to convert its remote echo host name into an IP address, which is saved in a local variable by the `DNSCallback` function.

Retrieving and Displaying Link Status Information

Many applications want to display status to the user while a net connection is being established. Newton Internet Enabler makes this easy for you with the `InetDisplayStatus` function, which displays link status information on the Newton screen. Here is the declaration of the function:

```
statusView InetDisplayStatus(linkID, statusView, status)
```

You can use the `InetDisplayStatus` function in three ways, as follows:

- to create a new status view, pass `nil` as the value of each parameter:

  ```
  myStatusView := InetDisplayStatus(nil, nil, nil);
  ```

- to display status for a link in an existing status view, pass in the link ID, the status view, and the status frame that was sent to your callback function:

  ```
  InetDisplayStatus(myLinkID, myStatusView, myStatus);
  ```

- to remove and dispose of the status view, pass `nil` as the value of the status frame:

  ```
  InetDisplayStatus(myLinkID, myStatusView, nil);
  ```

The `InetStatusDisplay` function creates and uses a view that is based on `protoStatusTemplate`. For information about this proto, see the chapter "Additional System Services" in *Newton Programmer's Guide*.

To initiate the status display, you need to open the status view. The most convenient place to do this is just before your call to the `InetGrabLink` function. For example, the following function creates the status view, stores it in `myStatView` for subsequent use, and then calls the `InetGrabLink` function:

```
DoGrabLink := func()
   begin
   myStatView := InetDisplayStatus(nil, nil, nil);
   InetGrabLink(nil, self, 'GrabLinkCallback);
   end;
```

While the grab operation is in progress, you can update the status display whenever your callback function gets called. For example, the following code segment from a grab link callback function updates the status display if no errors have occurred and if the link status has not yet become `'connected`:

```
   if err = nil and status.linkStatus <> 'connected then
     InetDisplayStatus(linkID, myStatView, stat);
```

When the grab operation is done, you can remove the status display. The following code segment from a grab link callback function removes the status display when the link status becomes `'connected`:

```
if err = nil and status.linkStatus = 'connected then
   InetDisplayStatus(linkID, myStatView, nil);
```

The view displayed by the `InetDisplayStatus` function contains a button that the user can tap to call the `InetCancelLink` function, which cancels the grab operation that is currently in progress.

Configuring Newton Internet Enabler for Your Endpoint

After grabbing your Newton Internet Enabler link, you need to instantiate your endpoint. You send the `Instantiate` message to your endpoint with the options required to configure Newton Internet Enabler for your application.

You must set three options in your `Instantiate` message:

- The `'inet` service identifier option, which tells the Newton system software to use Newton Internet Enabler with your endpoint.
- The Inet tool physical link (`'ilid'`) option, which tells Newton Internet Enabler which link ID to use for your endpoint. Use the link ID that was returned by the `InetGrabLink` function.
- The Inet tool transport service type (`'itsv'`) option, which tells Newton Internet Enabler which transport type (for example, UDP or TCP) to use for your endpoint.

Binding Your Endpoint with Newton Internet Enabler

After you instantiate your endpoint, you need to bind it to an address. You either bind your endpoint to connect (initiate an outgoing connection), or to listen for an incoming connection. If you are binding an endpoint that is going to listen, you always need to pass the Inet local port (`'ilpt'`) option when you send the `Bind` message to your endpoint. If you are binding an endpoint that is going to connect, you need to pass the Inet local port option for UDP links, but not for TCP links.

The Inet local port option has two data slots that you specify: a short value, `InetPortNumber`, and a Boolean value, `useDefaultPort`. The `useDefaultPort` value only applies when you are binding an endpoint to connect over a UDP link. Assign the

`InetPortNumber` a value as shown in Table 1-1 when sending the local port option with a `Bind` request:

Table 1-1: Local port numbers for binding with Newton Internet Enabler

Bind type	Transport service type	Local port number
For connect	TCP	The system always selects the local port number, so don't set this option. You can, however, send a get (`opGetCurrent`) of this option with your `Bind` to retrieve the port number that the system assigned.
For connect	UDP	If you specify `true` for `useDefaultPort`, Newton Internet Enabler will select the local port to use and will return its value in the option. If you specify `nil` for `useDefaultPort`, you must supply a port number that is not in use or the `Bind` will fail.
For listen	TCP	Specify a port number to listen on as defined by the *IETF RFC 1700: Assigned Numbers* (October, 1994) document.
For listen	UDP	Specify a port number to listen on as defined by the *IETF RFC 1700: Assigned Numbers* document.

Connecting Your Endpoint with Newton Internet Enabler

After instantiating and binding your endpoint, you need to connect it. If you are using a TCP link, you need to pass the TCP remote socket (`'itrs'`) option when you send the `Connect` message to your endpoint. This option sets the host address with which TCP connects. You can use the domain name server to get this address.

If you are using a UDP link, you do not need to pass any options in your `Connect` message.

If you are using your endpoint to listen for an incoming connection, you do not need to send any options with the `Listen` message.

Sending Data

You use Newton Internet Enabler to send data just as you would with any Newton communications tool. You can set up an output specification frame and send the `Output` message to your endpoint after you have established a connection.

For UDP connections, you need to include the Inet UDP destination socket (`'iuds'`) option to establish the destination of the UDP datagram. Your UDP output specification must include two flags in the `sendFlags` slot: the `kPacket` and `kEOP` flags. For example, the following code segment sends the string "Hello World!" out over a UDP link.

```
local myUDPstreamOutputSpec := {
  form:        'string,
  sendFlags:   'kPacket+'kEOP,
}

local myUDPOptions :=
[{
  label:      "iuds",
  type:       'option,
  opCode:     opSetCurrent,
  result:   nil,
  form:       'template,
  data:
  {
    arglist:
  [
    130,          // byte 1 of host address
    43,           // byte 2 of host address
    2,            // byte 3 of host address
    2,            // byte 4 of host address
    7,            // destination port number
  ]
    typelist:
    [
      'struct,
      'byte,
      'byte,
      'byte,
      'byte,
      'short
    ]
  }
}];

try
  ep:Output("Hello World!", myUDPOptions,
                      myUDPstreamOutputSpec);
onexception |evt.ex.comm| do
  return :DoDisconnect();
```

For TCP links, you do not need to include any options in your `Output` message, nor do you need to specify any `sendFlags` values in the output specification frame. For example, the following code segment sends the string "Hello World!" out over a TCP link.

```
local myTCPstreamOutputSpec := {
  form:    'string,
}

try
  ep:Output("Hello World!", nil, myTCPstreamOutputSpec);
onexception |evt.ex.comm| do
  return :DoDisconnect();
```

The above example calls the application's `DoDisconnect` function if any communication exception occurs while sending the data.

You can also send expedited data over a TCP link. Expedited data is a single byte of data that gets sent immediately. The data byte gets inserted in front of any data on the remote end that has been received but not yet processed. For example, you might need to send out a break character in the middle of transmitting a large amount of data. To do so, you use the Inet expedited data option with your `Output` message.

See the chapter "Endpoint Interface" in *Newton Programmer's Guide* for detailed information about output specification frames and the `Output` method.

Receiving Data

You use Newton Internet Enabler to receive data just as you would with any Newton communications tool. Typically, this means that you set up an input specification frame and send the `SetInputSpec` message to your endpoint.

For UDP links, your input specification frame must include the `kPacket` receive flag and must include `useEOP:true` in the termination slot. In addition, you can include two options in the `rcvOptions` slot if you want to: include the UDP source socket option to retrieve the address of the datagram sender, and include the UDP destination socket option if you want to retrieve the exact address to which the packet you received was sent. The destination address might be other than your local address if the packet was sent to a broadcast address.

The following code segment receives a datagram packet over a UDP link.

```
local streamInputSpec := {
  form:           'string,
  termination:    {useEOP: true},
  discardAfter:   256,
  rcvFlags:     kPacket,
  rcvOptions:     {
                label:    "iuss",
                type:       'option,
                opCode:     opGetCurrent,
                result:     nil,
                form:       'template,
                data: {
                    arglist:
                    [
                      0, // host addr - byte 1
                      0, // host addr - byte 2
                      0, // host addr - byte 3
                      0, // host addr - byte 4
                      0, // host port number
                    ]
                    typelist: kPortAddrStruct,
                    [
                      'struct,
                      'byte,
                      'byte,
                      'byte,
                      'byte,
                      'short
                    ]
                }
              }

  inputScript: func(ep, data, terminator, options)
    begin
    // do something with data
    end,

  completionScript: func(ep, options, result)
    begin
      // skip error handling for canceled requests
    if result <> kCommAbortErr then
      begin
      print("Error: " && result);
      ep:DoDisconnect();
      end;
    end,
}

try
  ep:SetInputSpec(streamInputSpec);
onexception |evt.ex.comm| do
  return :DoDisconnect();
```

The example input specification frame above tells Newton Internet Enabler to receive a packet of data from the UDP link and provides two scripts: the `inputScript` function to process normal completion of data reception and the `completionScript` function to process unexpected termination of data reception. In addition, this input spec includes a "get" of the UDP source socket address, which will be filled in with the IP address of the host that sent the datagram to your application.

For TCP links, you do not need to include any options or specify any receive flags in your input specification frame. For example, the following code segment receives a carriage return-terminated string from a TCP connection.

```
local streamInputSpec := {
  form:          'string,
  termination:   {endSequence: UnicodeCR},
  discardAfter:  256,

  inputScript: func(ep, data, terminator, options)
    begin
    // do something with data
    end,

  completionScript: func(ep, options, result)
    begin
      // skip error handling for canceled requests
    if result <> kCommAbortErr then
      begin
      print("Error: " && result);
      ep:DoDisconnect();
      end;
    end,
}

try
  ep:SetInputSpec(streamInputSpec);
onexception |evt.ex.comm| do
  return :DoDisconnect();
```

The example input specification frame above tells Newton Internet Enabler to terminate input upon receiving a Unicode carriage return character and provides two scripts: the `inputScript` function to process normal completion of data reception and the `completionScript` function to process unexpected termination of data reception.

You can also receive expedited data over a TCP link. When expedited data arrives, your application is immediately notified: the link controller sends an application event frame. The `eventCode` slot of this event frame has the value `kEventToolSpecific` and the `data` slot is the byte that was received.

See the chapter "Endpoint Interface" in *Newton Programmer's Guide* for detailed information about input specifications, the `SetInputSpec` method, handling communications events, and other styles of receiving data with an endpoint.

Disconnecting Your Endpoint

When you have finished using your endpoint, you need to disconnect, unbind, and dispose of it. The following function shows you an example of finishing your use of an endpoint.

```
MyApp.DoDisconnect := func()
  begin
  if ep then begin      // ignore all disconnect errors
    try
        ep:Disconnect(true, nil);
    onexception |evt.ex.comm|  do
        nil;

    try
        ep:UnBind(nil)
    onexception |evt.ex.comm| do
        nil;

    try
        ep:Dispose()
    onexception |evt.ex.comm| do
        nil;
    end;
  end;
```

Releasing Your Link

After your application is completely done with the link, or whenever you will not be using the link for a long period of time (approximately 15 minutes or longer), you need to release it by calling the `InetReleaseLink` function. If no other applications are using the link, the Newton system software shuts it down.

You need to provide `InetReleaseLink` with a link ID, a callback function, and a callback context frame:

```
  InetReleaseLink(linkID, clientContext, clientCallback)
```

USING THE DOMAIN NAME SERVICE INTERFACE

You can use the Newton Internet Enabler domain name service functions to translate between host name and Internet address representations. Newton Internet Enabler provides the following domain name service global functions:

- the `DNSCancelRequests` function cancels any pending DNS requests.
- the `DNSGetAddressFromName` function translates a domain name into its corresponding Internet address.
- the `DNSGetMailAddress` function translates a domain name into the Internet address for a mail server that serves that domain.

- the `DNSGetMailServerNameFromDomainName` function translates a domain name into the domain name for a mail server that serves that domain.
- the `DNSGetNameFromAddress` function translates an Internet address into its corresponding domain name.

You must supply a `clientContext` and `clientCallback` parameter to each of the DNS functions, just as you do for the link controller functions. However, the DNS callback functions are called with different parameters than are the link controller functions.

The callback function for `DNSCancelRequests` receives no parameters.

The callback function for all of the other DNS functions receives two parameters: an array of DNS results frames and a result code. Each results frame contains a number of slots that describe the DNS operation that was performed. For example, the `DNSGetAddressFromName` function is declared as follows:

```
DNSGetAddressFromName(addr, clientContext, clientCallback)
```

An example of a callback for this function is shown here:

```
myApp.DNSGetAddrcallback := func(results, error)
  begin
  if error or length(results) < 1 then
    begin
    print("DNS error: " && error);
    // do something with the error
    return;
    end;

    // save the resolved address
  myRemoteIpAddr := results[0].resultIPAddress;
  end;
```

Each results frame contains a type slot and at least one result slot. Most results frames contain the `targetDomainName` slot; however, this is not guaranteed. Table 1-2 shows which slot is guaranteed to be valid for each DNS operation.

Table 1-2: Result slots for each DNS operation

DNS operation	Results frame slot
`DNSGetAddressFromName`	`resultIPAddress`
`DNSGetNameFromAddress`	`resultDomainName`
`DNSGetMailServerNameFromDomainName`	`resultDomainName`
`DNSGetMailAddressFromName`	`resultIPAddress`

For example, the `DNSGetAddressFromName` function returns a results array that looks something like this:

```
[{
type: kDNSAddressType,
targetDomainName: "newton.apple.com.",
resultIPAddress: [155,227,54,3]
}]
```

In contrast, the `DNSGetNameFromAddress` function returns a results array that looks something like this

```
[{
type: kDNSDomainNameType,
targetDomainName: "newton.apple.com.",
resultIPAddress: [155,227,54,3]
}]
```

Some DNS operations return a results array that contains more than one results frame. For example, a mail exchange operation can generate multiple mail exchange results frames.

Using the Newton Internet Enabler Options

You configure the Newton Internet Enabler links with communications options. Send these options down with your endpoint method calls, as you do for other communications tools. Table 1-3 describes the Newton Internet Enabler options, including which options to send with which endpoint methods.

Table 1-3: Newton Internet Enabler options

Option name	Description	When to use
Expedited data transfer (`'iexp'`)	For expedited transmission of data over a TCP link.	Set this option with an `Output` call to transfer data on a TCP endpoint.
Physical link identifier (`'ilid'`)	To identify the link ID to use.	Set this option at endpoint instantiation time.
Local port (`'ilpt'`)	To set the local port number for TCP binds. (at endpoint instantiation or bind time). You don't need to set this option for a `Connect`.	Set this option if you are binding to do a `Listen`
	To set the local port number for UDP binds.	Set this option at endpoint instantiation or bind time.
	To retrieve the local port number used for TCP or UDP.	Retrieve the value of this option when you are connecting, sending, or receiving data.
TCP remote socket (`'itrs'`)	To set the socket to which TCP connects.	Set the value of this option at before using the connection (at endpoint instantiation, bind, or connect time).
	To retrieve the sender address for data received over a TCP link.	Get the value of this option when listening for data on a TCP connection.
Transport service type (`'itsv'`)	To set the transport service type (TCP or UDP).	Set this option at endpoint instantiation time.
UDP destination socket (`'iuds'`)	To set the destination address for data being sent over a UDP link.	Set this option when sending data with a UDP connection.
	To retrieve the destination address for data received over a UDP link.	Get this option when listening for data on a UDP connection.
UDP source socket (`'iuss'`)	To retrieve the source address for data received over a UDP link.	Get the value of this option when listening for data on a UDP connection.

Newton Internet Enabler: Providing Convenient Connectivity

To summarize, the Newton Internet Enabler makes it easy for you to provide Internet connectivity in your NewtonScript applications. If you know how to use Newton communications endpoints to send and receive data, then you already know how to use the Inet tool to send and receive data over the Internet.

The only additional concept that you need to understand to use the Newton Internet Enabler is that of links. You establish a link to establish an Internet session with a certain Internet services provider. Each link is configured to work with a certain transport service (TCP or UDP) and to use a certain link-level protocol (PPP or SLIP). Although only one link can NTJ

Communications Technology

Enabler Kits

by Eileen Tso, Apple Computer, Inc.

With the introduction of the Newton 2.0 OS, communications capabilities are more important than ever. The Newton Systems Group relies on support and feedback from developers to make sure that effective communications applications are created for the Newton platform. "Communications Strategies for Newton 2.0" alludes to a number of tools and "kits" which we are in the process of putting together. Developers can use these aids to help us carry out the vision we're planning for the platform and associated solutions.

eMail Enabler

The first of these kits is the Newton eMail Enabler. As most of you are aware, a significant accomplishment of Newton 2.0 OS was that it made e-mail choices and related complications transparent to users by implementing transports and a Universal In/Out Box. However, with our built-in eWorld solution on MessagePads, we perhaps made it too easy for developers to simply take that as "sample code" and replicate the client for other e-mail solutions (for example, Internet POP/SMTP, LAN-based e-mail access, dial-in services, and so on). Now that eWorld no longer exists and that client is essentially unnecessary, it becomes apparent that the eMail Enabler is what we should have provided originally, with or without eWorld.

As mentioned in the Communications Strategies article, the eMail Enabler includes an NTK streams file, elements of an e-mail transport, stationery, views, and so on. Some of you may recall a query made by DTS last year asking how many of you within our existing developer base would be in need of such a tool. The response was a resounding plea for an alternative to the eWorld code. Stay tuned: this summer should bring your e-mail solution to completion, without the need for the built-in client or "sample code."

Messaging Enabler

The second kit described in the article is the NewtonMessaging Enabler. Those of you who have heard rumors or snippets from previous conversations may know this tool as the "Paging Enabler." Early on, we realized there was a need to transport data more seamlessly with page cards and similar devices. Now that the industry is starting to cross barriers, with pagers functioning more like messaging devices and packet radio proliferating, what started as the Paging Enabler has become a much stronger and more useful tool as the Messaging Enabler. As with the eMail enabler, not all developers will require the Messaging Enabler. It isn't a magical black box/middleware combination of code which will allow you to take an existing application and get it to send pages. But, not far from that scenario, it will allow you to speak to those messaging devices if you so choose. From there, the 2.0 transport would take over, and then your application.

Internet Enabler

The last enabler described in the Communications Strategies article – and also discussed at length in Gary Hillerson's accompanying article – is the Newton Internet Enabler. Briefly, the NIE is Newton TCP/IP. Like the other enablers, NIE will not be a necessity for every communications developer. But also like the other pieces, it will be made freely available to developers this summer.

If you'd like to obtain more information on any of these enablers, including detailed schedules, timelines, or functionality, please send your inquiries to NewtonDev@AppleLink.apple.com, and your message will be forwarded accordingly. As you can imagine, these three efforts are being managed from distinct subgroups within our organization, and although DRG and your respective evangelists are always here to provide you with information, sending an e-mail message to NewtonDev will get your issues, questions, and comments directly into the hands of those managing the enablers.

Once again, we are extremely appreciative of your continued support and your contributions to our platform. And, as always, we look forward to seeing the results of our work together.

NTJ

Communications Strategies for Newton 2.0

by Barney Dewey, Apple Computer, Inc.

Today, the definition of "work environment" is rapidly changing. People are roaming farther and farther from their base of operations, whether that base is a corporate desk or a home office. Job-related information now resides not just on a local hard drive, but on servers maintained by in-house MIS staff or remote information service providers. Although it's easier than ever to work where you want to, keeping in touch and getting to the information you need can still be complicated, involving multiple communications methods with different protocols, interfaces, and software.

Newton 2.0 OS supports a broad range of communications technologies. Whether mobile users require wired or wireless means to exchange information, Newton 2.0 OS supports numerous communications protocols and standards for each. The Newton 2.0 OS architecture supports the following communications technologies:

- Fax send and receive (built-in)
- E-mail (built-in and third-party)
- One- and two-way paging and messaging
- Networking (built-in AppleTalk)
- Internet protocols (TCP/IP, UDP, PPP)
- Packet radio networks (ARDIS, CDPD, RAM)
- Cellular fax and data (AMPS, digital, GSM)
- PCS fax and data (CDMA, PCS-1900, TDMA)
- Wireless LAN support

Newton Platform Communications Strategy

The basic communications strategy for the Newton platform is threefold: core communications, wireless communications, and Internet connectivity. Newton Systems Group has a number of initiatives to fulfill this strategy. They are:

- Build in the most essential capabilities (e.g., serial connection, fax)
- Encourage developers to enhance the built-in capabilities and to provide new capabilities
- Assist developers by providing communications APIs and enablers (enablers are described below)

The Newton 2.0 OS has a full set of communications APIs such as Transport, Routing, In/Out Box, and communications scripting. Because most types of communications applications have one or more common functions, we believe that providing a higher-level program interface will simplify the development of many communications applications. The developer functions that provide this higher level of interface are called enablers. The first enabler we provided was the Modem Enabler. In this article we will introduce you to three new enablers: Newton eMail Enabler, Newton Messaging Enabler, and Newton Internet Enabler.

Some enablers – like the eMail Enabler and the Messaging Enabler – provide higher-level APIs and code that developers can use to streamline the development process. Others – like the Internet enabler – provide new functionality. (For more information on these enablers, see "Enabler Kits.")

We have five groups in Newton Systems Group working on communications: Developer Relations, Developer Technical Support, Communications OS Engineering, Communications Solutions Engineering, and Product Marketing. Two of the groups – Developer Relations and Developer Technical Support – are entirely devoted to supporting third-party developers, who continue to be the most important part of our communications strategy. The Enabler Kits discussed here are one method of taking key enabling technology developed by our Communications OS Engineering and Communications Solutions Engineering groups and making it available to developers.

Core Communications

Our first strategic goal is to have the best core communications of any PDA. When we talk about core communications, we mean much more than basic communications; core communications includes all of communications capabilities one expects on a laptop or desktop computer. Today, Newton PDAs include more built-in communications capabilities than many laptops and desktop machines. We intend to expand this leadership in future Newton PDAs.

Let's review the core communications elements, as well as a new e-mail initiative.

Universal In/Out Box. This capability provides common access and control for most communications, including e-mail, faxing (send and receive), infrared transmission, wireless messaging, and printing.

Modem support. Newton 2.0 OS allows Newton PDA users to connect a modem via the serial port or through a PCMCIA-standard PC Card slot. Newton 2.0 technology supports data transfer over a modem at up to 28.8K bps and beyond. Our Modem Enabler allows developers and device manufacturers to easily write scripts (we call them "setups") for most modems to operate with the Newton 2.0 platform.

Network Connectivity. One of Apple's strategic directions for the Newton platform is to provide network connectivity to the most popular PC networks. Newton 2.0 OS uses AppleTalk for this purpose. AppleTalk support provides connectivity to most Macintosh and Windows NT networks. Future versions of the Newton Internet Enabler (see "Enabler Kits") will allow TCP/IP over Ethernet networks.

Access to corporate LANs. Newton 2.0 OS provides an extended AppleTalk stack, so users can access a local AppleTalk network from any node just by connecting to that node's AppleTalk connector. AppleTalk is also supported as a standard network (AppleTalk ADSP) by Windows NT, which allows connectivity to most Windows networks.

Access to the Ethernet LANs. Our strategy for network access is to expand from AppleTalk to support TCP/IP over Ethernet. We expect this capability to be available through future Apple enabler capabilities and the use of third-party Ethernet drivers and Ethernet access cards within the next year.

Fax send and receive. Full fax support is part of our strategy to provide more core communications capabilities in the Newton platform. Newton 2.0 OS not only provides the capability to send faxes, but the capability to receive faxes, with the option to annotate them before forwarding them to someone else. As a part of our strategy to continually update communications capabilities, we are adding Class 2 fax support, especially important in markets outside of North America.

The Name and Fax Number information stored in the Name files is closely integrated with the sending fax feature in Newton 2.0 OS. Newton 2.0 OS supports complex dialing codes and multiple phone numbers.

Printing. The Newton PDA strategy is to support printing via networks and serial-connected printers. Printing over AppleTalk is available. Printer drivers are built in for most Apple printers; support for many other popular printers is available with the Newton Print Pack. We plan to support printing over TCP/IP Ethernet networks over the next year.

E-mail. Apple's e-mail strategy for the MessagePad continues to be one of providing an architecture and development environment to support a wide variety of e-mail, including an online e-mail client for services like CompuServe and America Online; LAN e-mail, such as Lotus cc:Mail, CE QuickMail, and Microsoft Mail, through remote-access mail clients; wireless e-mail from RadioMail and WyndMail; and Internet mail access through applications like Qualcomm's Eudora.

To support this strategic direction, we are developing the Newton eMail Enabler. The eMail Enabler consists of the following elements:

- A documented and preferred user interface for eMail and messaging clients for the Newton platform
- Documented APIs
- An NTK streams file that provides the common elements of an e-mail application. Example elements are: routing slip, connection slips, In/Out Box headers, eMail stationery, and configuration slips.

You can make inquires concerning the eMail Enabler at NewtonDev@applelink.apple.com. Put "eMail Enabler" in the subject line of the message.

Wireless Communications

Having the best wireless communications is important to the future of the Newton platform. As workers depend more on electronic services (e.g., e-mail, corporate database access, and Internet access) and tend to spend more time away from their desks, wireless access will become essential for a number of industries and occupations.

We are working closely with licensees to provide integrated wireless solutions. We are also providing expansion paths in Apple products (including the MessagePad), such as PC Card Type II support for various wireless networks.

Discussed below are key wireless capabilities in the Newton platform and a new initiative for one- and two-way paging and messaging.

Cellular support. We will continue to provide connectivity for popular cellular phones with cellular/fax modems. The Newton Modem Enabler provides a facility to support most cellular modems. The need for professionals to access dial-up services (e.g., corporate LAN e-mail or the Internet) continues to grow. Many customers like to leverage their cellular phone by using it with the Newton platform.

We are currently watching the development of CDPD (Cellular Digital Data Packet). This cellular technology allows data transmission over cellular voice networks, and will allow high throughput of wireless data. So far, CDPD has been adopted mainly for vertical applications, and does not look like it will become a nationwide standard.

GSM. GSM is a popular digitally-based cellular standard outside North America. GSM could also become popular in North America if it is widely adopted as PCS-1900 in the PCS band. The Newton platform strategy for GSM continues to include built-in data adapter support for most phone brands and Modem Enabler and device driver support for other devices.

PCS (Personal Communication Services). Newton 2.0 has the communications architecture to support the future PCS-1900, TDMA, and CDMA data systems. Some of these technologies will be implemented in North America over the next few years. They promise to bring lower-cost wireless communications and advanced services like personal telephone numbers (telephone numbers that are not tied to a specific device but follow a person around) and high-speed data transmission (64 kbs or more).

Wireless LANs. Some customers need to stay connected to a local-area network – such as in a warehouse or on a large campus – even though they must roam the area continually. Newton 2.0 lets such mobile users stay connected to their LAN through wireless means using their Newton PDA, while maintaining a very high data transmission rate. We support "wireless Ethernet"-type capability that provides true mobility for many vertical customers with our device driver kit. Both 2.4 GHz and infrared solutions are shipping from third-party device providers.

Packet radio. Today, PC Card solutions exist that allow Newton 2.0 PDA users to access ARDIS and RAM two-way packet radio services. These services provide nationwide wireless data transmission in the U.S. In Europe, Mobitex networks (Ericsson) and the DataTAC networks (Motorola) are widely available. Our strategy is to work closely with developers and device manufacturers to develop support for these networks and devices, as well as future networks and devices.

Infrared (IR). Newton 2.0 has built-in infrared technology, so Newton PDAs can exchange information with one another by "beaming" Newton data, such as notes, name and address cards, and call records. Our strategy for IR is to support standardized IR transports as they become widely used. We plan on supporting IrDA in future products within the year. There are many opportunities for developers to provide driver and connectivity applications for the Newton platform to transfer data via IrDA to desktop applications and other devices (e.g., printers and specialized devices).

One- and two-way paging and messaging. Through the Newton 2.0 communications architecture, mobile professionals can use a Newton PDA to send and receive wireless pages (both alphanumeric and two-way pages). Newton 2.0 OS supports multiple paging cards, and all messages are received into the Universal In/Out Box for centralized, efficient messaging. Through broader paging technology support and developer applications, the Newton platform can support non-messaging uses such as automatic wireless updates to databases, trading of stocks and bonds, and monitoring of remote systems.

To support this strategic direction in paging and messaging, we are developing the Newton Messaging Enabler. The Messaging Enabler is both a developers' tool and an end-user application. It consists of the following elements:

- A documented API to allow easy integration with developer applications through the "routing slip" and "Put away" functions
- A simple end-user application for reading and sending text and data (this application can be expanded by developers through the documented API)
- Device drivers for one-way pagers (the Socket PageCard and the

Motorola NewsCard) and two-way pagers (the Motorola Tango). Device developers can write additional drivers for new paging devices as they become available.

You can make inquires concerning the Messaging Enabler at NewtonDev@applelink.apple.com. Put "Messaging Enabler" in the subject line of the message.

Internet Communications

Internet technologies are essential to the Newton communications strategy. They will provide better networking capabilities when coupled with Ethernet and AppleTalk networks. Perhaps the most exciting part of the Internet is the trend for Internet technologies (e.g., PPP, TCP/IP, HTML, GIF, POP/SMTP) to replace proprietary clients.

An example of this trend is Lotus cc:Mail For the World Wide Web, Release 1.0. It allows full cc:Mail support without cc:Mail client software. Another example is the use of Lotus InterNotes or Oracle WebServer to provide access to corporate processes and databases without using Lotus Notes or Oracle SQL database client software on the Newton platform.

The advantages to corporations of using Internet technologies for mobile platforms are:

- No change is required to the corporate database

- The mobile platform, free from the restrictions of proprietary applications and protocols, becomes a "universal client" that offers complete access to corporate data without providing mobile software

To bring Internet technologies to the Newton platform, we have four strategic initiatives underway: third-party Internet applications development, improved hardware capabilities, enabling software, and Apple-branded solutions.

Internet applications development. We are relying on and working closely with developers to provide key Internet applications such as HTML, e-mail, newsgroups, Telnet, and FTP. There are many opportunities for developers in these areas, as well as in other Internet areas.

New hardware capabilities. To enhance the Newton platform for Internet technologies, we are making changes in three hardware areas over the next year:

- *Memory.* The trade-off for memory is always cost. We are providing more system memory in Apple products (e.g., the MessagePad 130) to fill the needs of running TCP/IP communications and Internet applications at the same time. We are also looking at ways of providing memory in a modular fashion.

- *Graphics.* Graphics are fast becoming an important part of the Internet experience. We are evaluating ways to provide improved screen capabilities so the Newton platform will have a graphics display.

- *Speed.* The need to support graphics, run multiple applications simultaneously, and offer TCP/IP communications push the need for better platform performance. We are evaluating the use of the StrongARM technology for use with the Newton platform.

Enabling technologies. A set of common Internet technologies like TCP, UDP, IP DNR, PPP, and SLIP are required to run Internet applications. Therefore, we are developing the Newton Internet Enabler (NIE). Version 1.0 of the NIE will provide a complete kit for developers, including the protocols mentioned above plus a scripting engine for establishing connections to ISPs. Future versions of NIE will build on this set of Internet technologies.

(The NIE is a strictly a developers' tool, and will be provided to developers to include in their applications. It will be licensed and handled in a manner similar to DILs. NIE 1.0 will be available this summer.)

Apple-branded solutions. To make it easier for customers to obtain the necessary elements for an Internet solution on the Newton platform, we are considering developing an Apple-branded product, like the Apple Internet Connection Kit, for the Newton. As the rest of the pieces discussed here come together, we will evaluate this opportunity for both Apple and our developers.

In Conclusion

Advanced communications capabilities are extremely important for the Newton platform, and the members of the Newton Systems Group are excited about the Newton platform communications strategy. The combination of built-in capabilities (e.g., serial connection and faxing) [illegible] development aids such as enablers will help Newton developers add new communications capabilities to their products, keeping Newton the premiere PDA platform. NTJ

Newton

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

NewtonScript Techniques

Using the NewtApp Framework

by Greg Christie, Apple Computer, Inc.

The NewtApp framework is a collection of system-supplied protos which can be combined to form a complete application. In the past, creating an application for Newton PDAs usually required assembling many protos to form a complete application. What sets NewtApp apart is the amount of functionality built into the NewtApp framework. Support for the basic system services – filing, finding, routing, overview management, stationery, and so on, is provided in the NewtApp protos. The process of creating an application with NewtApp consists primarily of describing your application's data and how to view it. It is then a rather simple task of assembling a shell of an application around your data. You can use DTS sample code as the basis for that shell, or you can create your own shell. Either way, once you've created your first NewtApp, it is easy to reuse that NewtApp to contain other types of data for other applications. A future DTS sample may contain a pre-built, reusable application shell to further accelerate development of your own applications.

Not all types of application lend themselves to the NewtApp treatment. A successful NewtApp is typically an application which deals with a single soup entry at a time and whose entries all come from the same soup. When used with the stationery mechanism, you can provide many different views of an entry. In addition to providing multiple views of a single type of data, stationery can be used to extend your application to use different types of data. For example, the single application Notes can handle lined notes, checklists and outlines. The same soup can contain items created by different stationery, and a NewtApp can display them all.

An application can be built with NewtApp whether it uses stationery or not. Since routing in Newton 2.0 OS is handled through stationery, and it is often very desirable to have multiple views of individual data items, stationery can be very useful. However, the choice is yours. If you do not need to print or fax items, provide multiple views, or extend your application, you can still use NewtApp. The DTS "Checkbook" sample demonstrates the use of the NewtApp framework without stationery. In addition, stationery can be used in an application which does not use NewtApp. This article, however, only covers the use of NewtApp. It can serve as a guide to both stationery- and non-stationery-based NewtApps.

The current NewtApp design has two chief limitations. First, a NewtApp can only be used with soup-based data. Secondly, a NewtApp can only work with entries from a single soup at a time. Therefore, it would not be appropriate for an application like a calculator or general utility. In addition, an application which must deal with data from several soups simultaneously, like the built-in Calendar, would be a bad candidate for NewtApp. However, if your application is like most Newton applications, or if you want to quickly prototype an idea for an application, then NewtApp is a good way to go.

Why NewtApp?

When NewtApp was designed it was intended as a way for the Newton Systems Group to create the applications in the ROM as well as being a supported framework for third-party applications. The Names, Notes, Calls and In&Out Box applications were all built using NewtApp. As the built-in applications for Newton 2.0 evolved, the NewtApp framework evolved along with it. Once the framework could be used, a codification of the procedures needed to implement an application was documented. These procedures for using the application framework are detailed and documented completely in the *Newton Programmer's Guide* and are demonstrated in the DTS sample code.

It is important to note that although you may be able to get different combinations of the NewtApp protos to function as an application, alternate construction techniques and arrangements are not supported, and may break in the future as the NewtApp framework evolves.

There were many design goals for NewtApp. Among these were:

- Increase speed and ease of deploying common Newton applications.
- Eliminate common code in Newton applications.
- Provide system services for "free."
- Encourage visual commonality amongst Newton applications.
- Provide an easy container for stationery.
- Allow third party applications to "Upgrade" along with the system.

Aside from the soup-based limitations, the NewtApp framework meets these goals and is a good way to start if you're new to Newton programming. It is also straightforward to convert your 1.x application to 2.0 using NewtApp. I mention the steps needed for this at the end of this article.

Using NewtApp

Using NewtApp to create an application may be a little different than what you're used to. The framework has several layers, and it may be confusing at first to decide what should go at each layer. This section first discusses the general technique of creating a NewtApp application, and then discusses each of the layers. Since stationery can figure heavily in the use of NewtApp and is a powerful mechanism for extending an application, this discussion will also cover stationery-specific features. Chapter 5 of the *Newton Programmer's Guide* covers the use of stationery in NewtApp in detail. For a demonstration of a complete, stationery-based NewtApp, see the DTS Sample code called "Who-Owes-Whom."

Before You Start

Three main flavors of Newton applications are supported by NewtApp. These three are card, page and roll. A card application, like Names, shows one entry at a time, and all entries are the same height as each other. A page application is one where only one entry is shown at a time, but the entries are of arbitrary height. The Calls application is an example of this flavor of application. The roll flavor is one which, like the Notes application, can show an arbitrary number of entries of arbitrary height. By supporting all three flavors, NewtApp gives you the flexibility to experiment with different styles for your application.

Another step before you begin is to sketch out your data structure and how you want that data to be represented visually. If using stationery, plan it carefully, and try to encapsulate the data-specific code into your stationery. Avoid presuming what data your application is displaying, and when designing stationery, remember that the stationery may be used outside of your container application. Don't hard-code references between your application and your stationery. Instead, use framework-supplied slots and inheritance to reference one layer from another.

The final thing to remember before building your NewtApp is that most of the work consists of assembling the protos and setting values in slots. The framework does the rest. This frees you to create a solid data design which is represented visually in an easy-to-use manner. To get this functionality, be sure to call inherited methods in the system-supplied protos when you write your own.

As a final note, be sure to include the required statements in your install and remove scripts. These magic incantations ensure that your application will register its stationery and itself for all needed system services.

Layers

The NewtApp framework is a collection of layers. From the top down, they are Application, Layout, Entry, and Slot. If using stationery, it is an additional layer between Entry and Slot. The Application layer is topmost and is where the various pieces are plugged together. The Layout layer provides the overview and default view of your data. The Entry layer is where your individual soup entries are displayed. The slot views allow the user to view and edit individual slots in your soup entry. When using stationery, the stationery layer contains the `viewDefs` for your data, and these `viewDefs` will contain your slot views.

The Application Layer

The NewtApplication proto serves as the base view of your NewtApp. If you want filing in your application, add a folder tab proto – either `newtFolderTab` or `newtClockFolderTab`. You can also put the status bar for your application in this base view. The `newtStatusBar` proto contains two slots – `menuLeftButtons` and `menuRightButtons` – where you put the buttons for your status bar. The buttons will automatically be laid out for you at runtime. If you want different buttons to be displayed for your overview versus your default view, NewtApp can handle this too. Create a slot called `statusBarSlot` in your base view, and in that slot put the declared name of your `newtStatusBar`. In each of your layouts (see Layouts, below) put `menuLeftButtons` and `menuRightButtons` slots. As your application switches layouts, the appropriate buttons will appear on the status bar.

Many of the buttons you see in NewtApps are supplied by the system. With stationery there is a New button which is a `newtNewStationeryButton`, and a Show button which is a `newtShowStationeryButton`. For any NewtApp application, routing and filing are provided by the `newtActionButton` and the `newtFilingButton`. All of the required behavior is built into these buttons; you only have to include them in the `menuLeftButtons` and `menuRightButtons` slots where appropriate.

In the `newtApplication` proto, you set a few slots which determine the application's behavior for scrolling, finding, and filing. In addition, the application layer also contains a few key slots to make your application function. These are `allSoups`, `allLayouts`, `allDataDefs` and `allViewDefs`. Use these slots to customize your NewtApp for the presentation of your application-specific data.

The allSoups Slot

The `allSoups` slot in the application layer is a frame of frames which defines the application soup. Each subframe of `allSoups` protos off of the system's `newtSoup` proto, and specifies a soup you want to work with in your application. A simple `allSoups` slot would look like this:

```
{ appSoup:{_proto: newtSoup,
        …}}
```

When your application is opened, these subframes contain the actual soups for your application. The required slots of a `newtSoup` are listed in the *Newton Programmer's Guide*, but a few advanced ideas are worth discussing here.

If you want to provide different soups for your application to work with, they can be specified here. At runtime (see Layout Level, below) you can determine which soup you are viewing. You can also use this same technique to provide different query specifications or sort orders for your application's soup. Just put each different one in its own subframe of allSoups. You should also place any methods which work on your soup in these frames. If your application requires other soups, for the values in pickers, or other related information, specify each soup here. These secondary soups will be created, registered and maintained whenever the application is open.

Finally, be aware that at runtime these frames are soups and you can add or remove entries, as well as perform any other soup-related task directly from these frames. Don't forget that the application base view cannot be inherited by methods defined in the `allSoups` frame. Though these slots are located in the base view, they proto to `newtSoup`, and they are not children of the base view. If you need to reference the base view from a method in a `newtSoup`, use `GetRoot().(kAppSymbol)` to get back to the application base view.

The allLayouts Slot

The two layouts for the application are set up in the `allLayouts` frame. They are built and instantiated as child views of the base view at runtime. This is a little different than building an application for Newton 1.x. In some applications, the basic view of the data and the overview were two different child views of the application base view, and you would show and hide each one as needed. Since overview management is a service handled entirely by NewtApp, you don't actually declare these layouts to the base view. Just create a frame with two slots – default and overview – and use `GetLayout`(filename) to specify the NTK layout file to use for each. Everything else is done for you.

The allDataDefs Slot

If your application uses stationery, this slot is where you specify the `dataDefs` for your application, as well as any other `dataDefs` you may be registering with the system. These `dataDefs` are installed and registered at package install time, and are unregistered and removed at package removal time. This is accomplished through the required call to `NewtInstallScript` in your package's `InstallScript`. As with allLayouts, this is a frame of frames. The slot name for each `dataDef` should match the symbol slot in that `dataDef`. See the *Newton Programmer's Guide* for more information on the required slots for `dataDefs`. The `allDataDefs` frame can explicitly list a set of frames and/or it can reference NTK layouts using the `GetLayout(filename)` function.

The allViewDefs Slot

The `allViewDefs` slot is where the `viewDefs` are linked to the `dataDefs` in your application. This includes any print or other routing formats you wish to use. The structure of this frame parallels the `allDataDefs` slot. For each `dataDef` slot in `allDataDefs` you need a frame which contains the `viewDefs` for that `dataDef`. Aside from the stationery requirement that one of the `viewDefs` must be named default, you are free to associate as many `viewDefs` with a single `dataDef` as you like. For example, if an `allDataDefs` slot contains:

```
{ dataDef1: GetLayout("dataDef1"),
  dataDef2: GetLayout("dataDef2")},
```

then a matching `allViewDefs` slot might look like:

```
{ dataDef1:{default:GetLayout("dd1DefaultViewDef"),
       notes: GetLayout("notesViewDef"),
       frameFormat: protoFrameFormat},
  dataDef2:{default: GetLayout("dd2DefaultViewDef"),
       frameFormat: protoFrameFormat}}
```

Note that for beaming and mailing, you can use the system-supplied `protoFrameFormat` which provides routing for free. Also, notice that you don't have to specify the same number of `viewDefs` for each `dataDef`. Even if the application is currently displaying an item with a `viewDef`, but that `viewDef` isn't available for another item, the NewtApp framework will switch to the default `viewDef` when needed. In addition, the `newtShowStationeryButton` will only list the `viewDefs` available for the currently displayed item.

The Layout Level

When we refer to layouts in NewtApp, we don't mean layouts in the NTK sense. Yes, the layouts in a NewtApp are NTK layout files, but so are the other levels of a NewtApp. In NewtApp the layout layer controls the general appearance of your application, so it may help to think of them as the "visual layout" of the application. NewtApp supports two layouts in an application. These are the layouts enumerated in the allLayouts slot of the application layer, and are called default and overview. The default layout is used to show the individual entries in your application soup, and the overview layout provides the familiar Newton overview of your soup. Currently there are three protos for the default layout, and two protos for the overview layout. Which protos are used depends on the flavor of application you are building – card, page or roll.

The most important slot to set in the layout proto is the `masterSoupSlot`. This slot contains the name of one of the slots from the `allSoups` frame in the application base view. In the example from in the `allSoups` section, the layouts would have a `masterSoupSlot` of `'appSoup`. In addition, if you use the `statusBarSlot` mechanism of the application base layer, then you can create `menuLeftButtons` and `menuRightButtons` at this level to replace the status bar buttons. Other key slots and methods are outlined in the *Newton Programmer's Guide* and demonstrated in the DTS sample code. One handy method to remember is `DoRetarget()`. This method will update the layout from the soup data.

So, if you need to advance to a particular entry, you can go to the entry in soup referenced by a layout's `masterSoupSlot` and call `DoRetarget()` to update the views.

The Default Layout

For the default layout in the application, use `newtLayout`, `newtPageLayout` or `newtRollLayout`. For card-flavor applications, use `newtLayout`. Page and roll applications should use `newtPageLayout` and `newtRollLayout` respectively. This layout level is where the user will view and edit soup entries. Therefore, the views which will contain your soup entries will be added to the default layout. For a `newtLayout` proto, use NTK to draw a `newtEntryView` (see Entry Views, below) as a child view of this template. For `newtPageLayout` or `newtRollLayout` use a `GetLayout(filename)-style` reference to an entry view in the layout-level `protoChild` slot, and the system will create the entry views as needed. If you want some fancy graphics as backdrop for your entries, or some controls which operate on a soup-wide basis, the layout layer is an appropriate place for them.

The Overview Layout

There are two different layouts which can be used as overviews: `newtOverLayout` is meant for card-flavor applications, and `newtRollOverLayout` is used for page- or roll-flavor applications. These overview protos give you a lot for your money. They provide the standard Newton 2.0 OS overview for the application, complete with scrolling, check boxes, icons based on `dataDefs`, and the one- or two-line summary of each entry. In addition, tapping on an item displays the corresponding entry in the default layout. You can suppress the check boxes, change the summary, or modify the behavior when an item is tapped. The details of the `newtOverLayout` and `newtRollOverLayout` protos are in the *Newton Programmer's Guide*. But to get standard overview behavior, just set the `masterSoupSlot` and go.

The Entry Layer

The entry layer is where an actual soup entry will be represented and displayed in your application. This is where you can work with individual soup entries, and display controls which correspond to slots found in all soup entries for your application. This level and below is where you do most of the programming needed to customize a NewtApp for different applications. There are three different entryview protos. For a card-flavor application use `newtEntryView`, and for page or roll applications, use `newtRollEntryView`. The third type is the `newtFalseEntryView`, which is used for stationery or slot views in a non-NewtApp-based application. Although not discussed further here, use of the `newtFalseEntryView` is an important topic, and is detailed in the *Newton Programmer's Guide.*

Since the `entryView` has a one-to-one correspondence with the soup entry, it is responsible for updating the view from changed soup data and updating the soup entry. Every entry view has a slot called target which contains the current soup entry. Anytime you need to write data or get data from the entry, you can use target to do this. Like the layout level, an entry view also has a `DoRetarget()` method. You can use this method to update the entry view if you've changed the target.

In addition to updating the view from the target, the entry layer is responsible for writing view changes back to the target soup entry. Actually, the changes themselves are made by the individual slot views (see below), but the changed entry is not written back to the store immediately. Instead, a call is made to the entry level method `StartFlush()`, which starts an idle timer. After a few seconds, or whenever you scroll entries, change layouts or close the application, the changed entry is written back to the store. If you ever modify an entry directly, and outside of your slot views, be sure to call `StartFlush()`.

There are other entry-level slots besides target which give you a handle on your current context at runtime. They include `currentDataDef` and `currentViewDef`, which contain the current stationery components being used in conjunction with the target. The `viewDef` and `dataDef` in these slots are the templates supplied in your package. At runtime, the current `viewDef` is instantiated into a "live" view. To get at the actual runtime views in your `viewDef`, there is a slot called `currentStatView` which contains your instantiated `viewDef`.

You may wish to have a header in your entry view, like the one in the built-in Notes application. There are two versions of this proto. To include action and filing buttons on the header, use `newtEntryRollHeader`. For a simple title and icon header, use `newtEntryPageHeader`. For page- and roll-style applications, you can control the header by drag-resizing of entries through the resizable slot in these header protos. In addition to the headers and any general entry level controls you add to this level, an entry view can hold individual slot views and the stationery container.

The Stationery Level

Although "stationery" is a term used to collectively describe `viewDefs` and `dataDefs`, in the layers of NewtApp we are concerned with only `viewDefs`. The use of `viewDefs` in NewtApp is very straightforward. You merely place a `newtStationeryView` inside your entry-level proto. You do not directly lay out `viewDefs` in your `entryView`. The system will match up the currently chosen `viewDef` for the `dataDef` which corresponds to the current soup entry, or target. Designing the `viewDef` itself is much more work than actually using it. If your `entryView` corresponds to a single soup entry with controls that pertain to your soup entries in general, then a `viewDef` is like a subentry view, displaying the data from that soup entry which pertains to its `dataDef` class.

If you want to provide multiple views of your data, for example an info view and a notes view, then all you have to do is create two viewDefs which show the different slots of your entry. So long as you have included a `newtShowStationeryButton` in your application, your users can easily switch back and forth between the different views. Any `viewDefs` you add to this application via supplemental packages will also be displayed in the Show picker. Although you can display the contents of any slot in an entry in any `viewDef`, it is better design to allow `viewDefs` to display only slots which are created by the `viewDef`'s corresponding `dataDef`. To display and edit the individual slots in a soup entry, `viewDefs` contain slot views. For more detailed information on `viewDefs`, see the *Newton Programmer's Guide.*

The Slot Level

There are many protos in the NewtApp framework which are used to display and edit single slots of a soup entry. These are collectively known as slot views, and share many similarities. There are three basic classes of slot views: plain, labeled, and miscellaneous. The plain slot views are ones like `newtTextView` or `newtRONumberView` which simply display and update the value in a soup entry slot. Examples of the labeled slot views include `newtLabelInputLine`, `newtSmartNameView` and `newtLabelPhoneInputLine`. These slots are related to the `protoLabelInputLine`. In addition to the capabilities of the plain slot views, these slot views include a label and can pop-up lists of potential values. The miscellaneous slot views include the `newtEditView` and the `newtCheckBox`.

Slot views are implemented for many types of common data. Text, integers, real numbers, symbols, and entire notes can be displayed and edited by using the appropriate slot views. The names of the slot view protos, while typically long and sometimes cumbersome, are good cues to indicate where to use each one. For instance, it's easier to know when to use a `newtLabelPhoneInputLine` than to type it. In addition to the data type being in the name, if the proto name contains the letters '`RO`" (as in `newtRONumberView`), this tells you that the proto is read-only and can be used only to display data.

Despite their varying formats and capabilities, all slot views have a common interface. All slot views use a path slot, which contains the path reference to that slot view's corresponding slot in the soup entry. This path reference can be a simple symbol, like `'myDataSlot`, or a path expression, like `[pathExpr: 'myClassFrame, 'myDataSlot]`. They all implement the `ReTarget` method, which forces a slot view to update its view's value from the value in the specified path. When any of the slot views change, they all trigger the flush timer in the entry level. Most slot views also implement a `JamFromEntry` method which is controlled by the `jamSlot`.

Jamming

Jamming is the technique used by the slot views to incorporate data from other soups' entries in their own entry. Certain label-style slot views can query another soup for the contents of their popup, or display a picker of soup entries for the user to choose from. The most useful of these is `newtSmartNameView`. To use data from the built-in names soup in your application, simply provide a `newtSmartNameView`. When the user taps this slot view, the standard Names list picker appears. When the user chooses a name from the picker, the current entry view in your application is "jammed" with the chosen names entry. The `JamFromEntry(otherEntry)` method in each of your slot views is then called. `JamFromEntry` checks to see if you've set the `jamSlot` for that view. If `jamSlot` is not nil, it is used as a path expression in the foreign entry. The contents of the slot view's path are then replaced by the contents of the foreign entry's `jamSlot`. For example, if I wanted to include the name and title of a customer in my application's entry, I would use a `newtSmartNameView` with a path of `'customerName`. In

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

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

Caching for Maximum View Scrolling Performance

by Jeffrey C. Schlimmer, Washington State University

Techniques for writing faster code are always of interest to NewtonScript developers. Faster implementations open new avenues of functionality for applications – what was once unthinkably slow can become marketably fast. Some speed techniques capitalize on details of the execution environment. For example, in the NewtonScript interpreter, it is quicker to iterate through the elements of an array with foreach than it is with an explicit for loop [McKeehan & Rhodes, 1995]. Careful attention to these details can lead to significant performance improvements. Other speed techniques are applicable to any execution environment. For example, it is faster to move invariant statements outside of loops to avoid repeating them unnecessarily. This article explores a combination of techniques like these to speed view scrolling by a factor of four.

Scrolling

A common task in a Newton application is to present a scrollable list of items to the user for their inspection and/or selection. Figure 1 depicts the sample application used to demonstrate the ideas in this article. On a Newton MessagePad 120, the application shows 14 of 100 items. It scrolls by one item when the user taps the Up or Down arrow. (The current Newton User Interface Guidelines recommend scrolling by one screenful minus one item. This policy may reflect assumptions about scrolling speed.) Each item includes a check box, an icon formatted to look like a button, two text elements, and a gauge. Hypothetically, when the user taps a single item, the application would respond in some way, perhaps by toggling the checkbox or switching to a layout to display the item in more detail. The sample application assumes that data for the items are in an array. If an application keeps its data in soup entries, slightly different techniques are needed. The Newton DTS sample code "True Grid" demonstrates similar ideas in the context of soups.

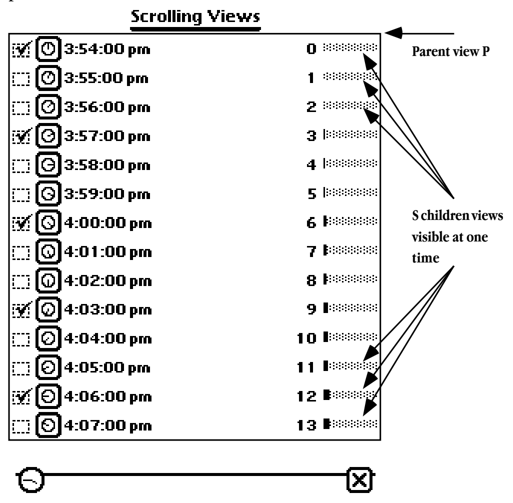

Figure 1. Sample Application with Scrolling Views

Using an Aggregate Proto and `setOrigin`

A simple and robust implementation for scrolling a list of items is to construct a proto that depicts a single item. The proto includes five children: a checkbox, a picture view, two static texts, and a gauge. These children initialize themselves using a `viewSetupFormScript`. In this script they inherit a slot value from the proto's item and set up their display accordingly. Construct a parent view (call it P) with one child for each item to be displayed (say there are N of them, for example, 100). P has its clipping flag set and is sized large enough to show one screenful of items (say there are S of them, for example, 14). The base application view responds to `viewScrollUpScript` and `viewScrollDownScript` messages by sending `setOrigin` to P. This changes P's vertical offset by the amount corresponding to the height of one child. Newton's view system automatically shows a different subset of the items depending on where the user has scrolled to.

To test how fast this implementation is, the application was sent a series of commands from the Newton Toolkit Inspector window: `open`, `viewScrollDownScript` 10 times, `viewScrollUpScript` 10 times, and `close`. Each message was followed by a call to `RefreshViews` to force Newton's view system to update. This sequence of commands was highlighted and evaluated all at once. Times reported here are measured by profiling and are rounded to the nearest tenth of a second.

To its credit, this affords fast scrolling. In a test using data similar to that shown in Figure 1, it took only 9.8 seconds to scroll down by 10 items and back up by 10 items. The N views corresponding to each item were constructed when P was opened, and the view system can rapidly move the child views into and out of the clipping region of P. This implementation is also easy to implement correctly; P can set up all N of the children at one time in its `viewSetupChildrenScript` by assigning each of them one item to display. Each child proto only has to initialize itself with a `viewSetupFormScript`. The price for this performance is the overhead associated with creating all N of the child views when P is opened. This creation takes time and heap space: 3.6 seconds to open and 32K of heap (as measured by GC and Stat, rounded to the nearest K). Along both of these dimensions this implementation is the worst of those considered here.

Using an Aggregate Proto and `syncChildren`

Another alternative is to build a parent view P with only S child views (that is, only as many as can be displayed at one time). As before, P is large enough to show one screenful but need not clip. The base view forwards the `viewScrollUpScript` and `viewScrollDownScript` messages to P. In turn P sends itself the `syncChildren` message. In response to this, Newton's view system sends P the `viewSetupChildrenScript` message and synchronizes the children views of P to match the stepChildren array. Specifically, any child view whose template is no longer in the stepChildren array is closed, a child view is constructed for any new template in stepChildren, and remaining views are redrawn if their template viewBounds have changed. Conceptually, to make P scroll down by one item, P only has to remove the template for the first item from the stepChildren array and add a new template to the end of stepChildren for the new item to be displayed. The view system will close the unneeded child view, shift remaining children up, and open a new view at the bottom.

In practice, this method works well for scrolling down. P's viewSetupChildrenScript uses a single call to ArrayMunger to splice all but the first element of the stepChildren array into the beginning of a new array containing only the new template. Newton's view system closes the top-most view, redraws remaining views just above their previous position, and opens the new view at the bottom of the display. However, scrolling up does not work as expected. Following the pattern used for scrolling down, P removes the last template from its stepChildren array with a call to SetLength and adds a new template to the beginning using ArrayMunger. Surprisingly, the view system closes the obsolete view at the bottom and draws the new view in its place without shifting the remaining views downward. To make this work, P can recreate the entire stepChildren array when scrolling up. This forces the view system to close and reopen each child view.

This approach alleviates both the worst properties of the first implementation. P opens much more quickly (only 1.1 seconds) and requires much less heap (only 5K). The drawback is that it scrolls very slowly taking 32.4 seconds to scroll up and down by 10 items. This time is unevenly divided between the fast down scrolling (when one view is closed and one opened) and the slow up scrolling (when S views are closed and opened).

Using an Aggregate Proto and an Update Message

An obvious third alternative is to avoid closing and reopening any children views. Using a parent view P with S children as before, when P receives the viewScrollUpScript and viewScrollDownScript messages it sends an update message to each of its S children views passing an index for a new item to display. To scroll the display up by one, each child view is sent an index one larger than it had before. To scroll down, an index one smaller.

Implementing this is slightly more complex than the first approach using setOrigin. P must keep track of the index of the first item currently displayed. The proto for the children must respond also to an update message and suitably initialize its components (the checkbox, picture view, static texts, and gauge). The proto could do this by sending itself a redoChildren message leaving the components to initialize themselves with their viewSetupFormScripts. A slightly faster alternative is for each component to have an update method that resets its part of the display, probably by calling SetValue. If the user is going to be allowed to scroll until the last item is at the top of the display (and the remainder of the display is blank), then the proto must also be able to handle the case where there is no item to be displayed at all.

This complexity yields an implementation that is almost as fast opening as the one using syncChildren and uses less heap (4K versus 5K). However, it scrolls more slowly. Even if we pre-compute the length of the array of items and cache childViewFrames in P's viewSetupDoneScript, it still takes 36.5 seconds to scroll down by 10 and back up by 10 items. The reduction in scrolling speed is a direct result of the extra messages sent to the S children views and to their grandchildren views, including the SetValue calls.

These approaches represent extremes. The approach using setOrigin exhibits aggressive computation; it computes all results at the first opportunity and then relies on a cache of these results. The cache takes time to construct and uses space but is fast in subsequent use. In contrast, the approaches using syncChildren and an update message exhibit lazy computation; they do a minimum of computation initially and compute only what is needed later. They pay no penalty for a cache but enjoy no benefit either.

To get better scrolling performance out of the update message approach, we need to avoid the recomputation entailed in each round of update messages. When the user scrolls, only one new item is displayed, and all but one of the previous items are re-displayed. In our example 14 items are displayed at one time. Each time the user scrolls, 13 old items are re-displayed. Nearly 93% of the computation is repeated!

Using a Simple Proto and a viewDrawScript

We can speed up scrolling with a lazy computation involving simpler views. Instead of building a proto with multiple child views, we'll use a drawShape message within a viewDrawScript to draw a comparable visual representation . What was once a child check box view is now one of two bit maps: either a checked or an unchecked bit map. The icon becomes a bit map, and its border a MakeRoundRect shape. The static texts become MakeText shapes. The gauge becomes a pair of MakeRect shapes.

We can optimize this slightly by taking advantage of the fact that the drawShape message is sufficiently flexible to draw an array of shapes at once. So the proto will send drawShape once to image all seven of its shapes to save on the overhead of repeated message sends. To further minimize drawing, the checked and unchecked bit maps can be combined with a bit map that looks like a round rectangle eliminating the need for a separate call to MakeRoundRect. The gray rectangle underlying the gauge could also be combined if the application was a fixed width and the distance between the gauge and checkbox were known at compile time.

The base view must still pass the viewScrollDownScript and viewScrollUpScript messages along to the parent view P. As before, P must pass along an update message to each of the S children. The update message includes the index of the item a specific child is to display. The proto just sends itself the "dirty" message instead of passing the update message to its components. The "dirty" message in turn triggers the proto's viewDrawScript.

This approach opens the application very quickly (1.3 seconds) but is still about twice as slow at scrolling as the setOrigin implementation (21.9 versus 9.8 seconds). However, it is quite a bit faster than the update method with aggregate protos. Newton's view system is able to execute drawing commands faster than it is able to update a comparable number of views. That the aggregate proto is slower should not be too surprising since the view system is doing quite a bit of work for each child of the proto sending viewSetupFormScript, viewSetupChildrenScript, viewSetupDoneScript, and other messages. Another advantage of a simple proto with a viewDrawScript is that it uses very little heap because there are very few views active. In our example shown in Figure 1, at least 5x14 = 70 views are used with aggregate protos but only 14 with simple protos. This estimate of a 5-fold savings in heap is comparable to the actual difference of 4K versus 1K.

Using a Simple Proto and a viewDrawScript with Caching

Though faster, the simple proto with a `viewDrawScript` is still repeating much work. Each of the shape creation calls, like `MakeText`, are repeated for the same item as the user scrolls up and down. To compensate for this, we could save the shapes for each item in a cache and reuse them. To construct the cache itself, P will allocate an array with N entries, one for each item. We could cache each individual shape created, indexed by an item's index, but the Newton OS provides a better function called `MakePICT`. This function accepts the same array of shapes and styles that `drawShape` does, but it constructs a PICT picture that can be stored and passed to `drawShape` as needed. This allows us to cache all of the drawing commands for an item as a single picture.

To use the cache, the proto's `viewDrawScript` will first check the array for a cached picture. If there isn't one, it calls `MakePICT` with shapes and styles, caching the result. Then the `viewDrawScript` sends `drawShape` message with the cached picture.

We don't expect the use of a cache to slow down the application as it opens, but it does slightly: 1.8 versus 1.3 seconds. There is some time penalty for constructing the pictures and then imaging them. Storing items in the cache takes very little time. Because the cache is an array indexed by the item's index, it also takes very little time to retrieve stored shapes. As a result, scrolling is a bit faster, down from 21.6 seconds to 14.7 seconds. Apparently only some of the overhead is due to shape creation.

The biggest difference is in the increase in heap used, up from 1K to 12K. The cache takes space; about 0.4K per item's picture. If we explore what happens if the user scrolls to the bottom of the list of 100 items and back up again, the cache grows to the point where the application uses more than 44K of heap! A simple and efficient solution to this is to uncache one picture each time the user scrolls (by setting the appropriate element of the array back to NIL.) Recall that each scroll brings one new item into view and removes one from view. The `viewDrawScript` can ensure that the new item's picture is cached, and the `viewScrollUpScript` and `viewScrollDownScript` can uncache the hidden item's picture. Making these changes slows scrolling by 0.5 seconds but limits the application's heap usage to 8K.

To save even more space, the cache can be made to hold exactly S pictures, just enough for the S child views. Using modulo arithmetic, each child view's index can be mapped into a unique location in the cache; for example, the sixteenth item has an index of 15. Taking this modulo S (= 14) yields an index of 1 in the cache. This position is also shared by the second item (with an index of 1), but it's easy to show that the second and sixteenth items are not displayed at the same time. This saves N – S empty array elements and could be significant for large numbers of items N.

The above approaches illustrate the general tradeoff between the time and space a computation requires. By caching pictures we can save 6.4 seconds of scrolling time at the expense of 7K of heap. The `setOrigin` implementation saves a further 5.6 seconds of scrolling time for an additional 24K of heap. A specific application's requirements usually help decide which point in this tradeoff is the most preferable though often it amounts to choosing between several unattractive alternatives.

Using a Simple Proto and Bit Map Caching

Newton 2.0 OS provides a useful capability that gives us another alternative in the time-space tradeoff for our scrolling application. Using the simple proto and a `viewDrawScript` we will cache the whole bit map for an item's visual representation. In the previous implementation we cached a picture, so that is a series of drawing instructions had to be executed by the Newton view system to image the view. Now we'll cache a bit map that specifies each pixel's value and can be imaged more quickly.

When the item is first drawn, it uses `drawShape` and shape creation functions as before, but the resulting image is then converted into a bit map by sending the `viewIntoBitmap` message. The resulting bit map is then cached in an array indexed by the item's index. When the item is to be re-displayed, the bit map is drawn instead of the various shapes.

Making this work efficiently requires a `viewDrawScript` with three separate conditions. The first condition checks to see if the cache contains a bit map for this particular item. If the array implementing the cache is non-NIL for this item's index, there is a bit map, and the `viewDrawScript` calls `drawShape` with it. The second condition is triggered if there is no cached bit map. It will create an empty bit map, send itself `viewIntoBitmap`, and `drawShape` and cache the result. The `viewIntoBitmap` message calls the `viewDrawScript` of the view, so the second condition sets and checks a flag to prevent an infinite loop. The third condition executes when the flag has been set and images the view in response to the `viewIntoBitmap` message. This is where the shapes are drawn as they would be in a non-caching `viewDrawScript`. For reference, a `viewDrawScript` that caches bit maps for the simple proto in our example is listed in Table 1. (Recall that in this code, the rounded rectangle shape is created as part of the checkbox bit maps rather than as a separate call to `MakeRoundRect`.)

Table 1. viewDrawScript for Simple Proto that Caches Bit Map Images.

```
func() begin
  if index then begin
    local B := madeBitmaps[index];
    // Draw the cached bitmap if there is one.
    if B then begin
      :drawShape(B,nil);
    end;
    // Otherwise, trigger viewIntoBitmap and cache result.
    else if NOT cachingInProcess then begin
      cachingInProcess := true;
      B := MakeBitmap(itemWidth,viewBounds.bottom,nil);
      :viewIntoBitmap(nil,nil,B);
      madeBitmaps[index] := B;
      :drawShape(B,nil);
      cachingInProcess := nil;
    end;
    // Called by viewIntoBitmap to image bitmap.
    else begin
      local Item := kItemArray[index];
      :copyBits(if Item.check then kCheckIcon else
        kUncheckIcon,0,1,modeCopy);
      :copyBits(Item.icon,19,5,modeCopy);
      :drawShape(
        MakeText(Item.text1,36,1,text1R,12),
        kTextStyle);
      :drawShape(
        MakeText(Item.text2,text2L,1,text2R,12),
        kTextStyle);
      :drawShape(
        MakeRect(sliderL,6,itemWidth,12),
        kBarBackgroundStyle);
      :drawShape(
        MakeRect(sliderL,6,
          itemWidth-RIntToL(36-Item.value*36/100),
          12),
        kBarForegroundStyle);
    end;
  end;
end
```

As with picture caching, using a bit map cache does not slow down the application much as it opens. This implementation opens in 1.5 seconds, about as fast as all of the implementations so far except the one using `setOrigin`. Our real target was to speed up scrolling. Caching bit maps yields the fastest performance of all the implementations: 8.5 seconds to scroll down by 10 and back up by 10 items. This is faster than even the `setOrigin` implementation and more than four times faster than the update method implementation. By uncaching bit maps for items scrolled off screen, heap requirements are limited to 12K, which is half that of the `setOrigin` implementation but about 50% more than the simple proto implementation that caches pictures. Each bit map takes about 0.6K.

The improvement in speed and the loss of heap echo the difference between commands to draw a picture and an image of that picture. The commands take less space to store but must be executed each time the picture is drawn (for example, cached shapes). An image of the picture takes more space to store but can be quickly displayed (for example, cached bit maps).

Using No Protos and a `viewDrawScript`

For logical completeness, consider an approach that doesn't use any children views. A view P could respond to `viewScrollUpScript` and `viewScrollDownScript` messages by changing a stored index and sending itself the "dirty" message. P's `viewDrawScript` could image all visible items without any children views.

This approach opens in 1.0 seconds (the fastest by a small margin). It uses very little heap (less than 1/2 K). But because it doesn't cache any of its shapes or `drawShape` results, it's scrolling performance is about half as fast as others. Adding caching would be relatively simple because the `drawShape` method will image nested arrays of shapes and style frames. Shapes for the item just scrolled on screen could be cached in the appropriate location, and shapes for the item just scrolled off could be uncached. The entire cache could be passed to `drawShape`. Taking this idea a step further, an implementation could send `drawShape` with the cached PICTs discussed previously. Or, an implementation could send `drawShape` with the cached bit maps also discussed previously. These variations should show the same scrolling speed improvement relative to their non-caching variants as those discussed previously. The heap advantage of eliminating children views would be negligible compared to the size of the cache.

The only major drawback to approaches using no children views arises if we want to allow the user to tap on single items to indicate some action. In this case, the view P would have to use the `hitShape` method to determine which item the user tapped, and P would have to implement the correct visual and auditory feedback for the tap, that is, highlighting and clicking. The added complexity of implementing this functionality from scratch rather than using Newton's view system may not be worth the incremental improvement in scrolling speed and heap space.

Summary

Table 2 summarizes the main results. Using aggregate protos either slow the application when it opens or when it scrolls. They can impose a very high or very low heap penalty. Using simple protos with a `viewDrawScript` can be fast both when the application opens and when it scrolls, but a cache must be used. The cache must be carefully maintained in order to avoid consuming too much heap.

Table 2: Speed, heap, and application size for alternative scrolling implementations.

	Time (seconds)		Size (K)		Lines
	Open/Close	Scroll	Heap	Package	Source Code
Aggregate Proto & `setOrigin`	3.6	9.8	32	10	157
Aggregate Proto & `syncChildren`	1.1	32.4	5	10	202
Aggregate Proto & update method	1.6	36.5	4	11	206
Simple Proto & `viewDrawScript`	1.3	21.9	1	10	152
Simple Proto & Cache PICTs	1.8	14.7	12	11	169
Simple Proto & Uncache PICTs	1.8	15.2	8	11	173
Simple Proto & Cache Bitmaps	1.5	8.5	18	11	194
Simple Proto & Uncache bitmaps	1.5	8.9	12	11	197
No Proto	1.0	17.0	0.4	11	155

This analysis might imply that a proto constructed with a number of child views would never be appropriate for a scrolling application. To help balance this impression, consider two rules of thumb for deciding between an aggregate approach and one using a `viewDrawScript`:

- An application should use an aggregate proto when datum components are to display differently and react differently to user input.
- An application should use a `viewDrawScript` when datum

NTJ

continued from page 1

Newton Toolkit Does Windows

separate utility applications.

In addition to Newton Toolkit for Windows, Apple will also soon be providing Newton C++ Tools for Mac OS, which will allow Newton developers to link certain compiled C++ routines directly into Newton applications without porting into NewtonScript. All of these products demonstrate Apple's commitment to delivering a comprehensive set of tools to meet the needs of Newton software developers.

Newton Toolkit for Windows is shipped on CD-ROM, and includes complete sample code and on-line documentation. A beta version is available right now on the WWW at http://www.dev.info.apple.com/newton, and the final version will ship through the Apple Developer Catalog (formerly APDA) and selected retail locations this summer.

NTJ

Creating Quality Newton Applications

by Peter Murray, Apple Computer, Inc.

Introduction

Quality does makes a difference. It makes a difference to your customers and ultimately to your bottom line. But quality assurance goes beyond just making sure you find and fix bugs – that's a given. However, you can't *test* quality into your application. Quality consists of meeting or exceeding customer expectations in all facets of the user experience. Aspects such as the simplicity, robustness, and utility of your application are critical to your success.

Whether you're a large or small developer, a strong quality assurance program can add a lot of value to your product. Besides giving you the confidence to ship a well-tested product, Software Quality Assurance (SQA) can act as the focal point for customer feedback upon which to build increasingly customer-driven products.

Qualifying software on the Newton is not much different from doing so on other platforms. There are three basic building blocks which comprise a good Newton SQA program:
- an understanding of software development
- a foundation of general SQA practices
- knowing the specifics of testing on the Newton platform

Software Development

Your product development should be planned before you begin implementation and you should execute to that plan. Marketing should define the target customers and gather and relate their requirements. Engineering should respond with a description of how those requirements will be met by the software, both at the human interface and low level. Based on the marketing and engineering requirements, SQA develops a test plan defining how testing will be done to converge on a high quality product. All of the teams must agree on what the end product will look like, who it's for, and what problems it solves. Never get so caught up in the details of the project that you lose sight of the big picture.

Software development is an iterative process. What does it mean for your software to reach the alpha, beta, final, and golden master milestones? What metrics do you choose to measure your progress, and by what milestone criteria do you judge the status of the project? These questions are largely dependent upon the goals of the project. Choose metrics and criteria with care. Minimalism is better than weighing your project down with bureaucracy.

A well planned product development process provides the guidelines for getting the work done, but software development is intrinsically dynamic. Make sure your day-to-day decisions still reflect the project goals and customer requirements.

General SQA Practices

The following concepts underlie any good SQA program.

- **Prioritized Test Coverage**

Since you can't test everything, prioritized testing is key.

What absolutely has to work from the customer's perspective to make your product successful? Concentrate more of your resources on those features and less on others based on their relative importance.

Coverage should be partially driven from development engineering.

Ask the development engineers about the areas of their code and hacks that concern them the most.

Look for interdependencies and assumptions. These might be bug-rich areas.

Test any error-checking. Simulate the errors and validate the error-handling.

Development and quality engineers should collaborate on creative ways to break the code.

Weaknesses might be uncovered in code or design reviews.

Inexperienced programmers might require more test coverage, especially in the system-specific features.

- **Shared Knowledge**

As much as possible, development and quality engineers should share a common understanding of the features of the software and the underlying code paths.

- **Scientific Method to Isolate Bugs**

The best quality and development engineers have internalized the formal scientific method and subconsciously use it when debugging. "The real purpose of the scientific method is to make sure Nature hasn't misled you into thinking you know something you don't actually know." [1] The following is a terse description of the scientific method as applied to SQA:
1) statement of the problem
2) hypotheses as to the cause of the problem
3) test cases to test each hypothesis
4) predicted results of the test cases
5) observed results of the test cases
6) conclusions from the results of the test cases

- **Managed Risk**

Most decisions related to qualifying and shipping the product entail varying degrees of risk. The challenge lies in correctly analyzing the risk and taking steps to minimize it. Risk analysis should be undertaken seriously, and any judgments should be based on facts, not conjecture. Always try to think of creative ways to minimize the risk. Here are some examples:

Code changes

If there are code changes late into the project or near critical milestones, there are some obvious ways to minimize the risk of the change:

The change should be code-reviewed by another development engineer.

The developer making the change should write a release note about the change, its impact on other areas of the system, and important tests to run.

SQA should verify and test the change and any related areas.

Bugs

Since you can't test everything or use the product in every way that a real customer might, you won't find every bug, and you'll almost certainly ship with known bugs. Make sure you prioritize the bugs to fix based upon their importance to the customer. If a bug happens in only obscure cases it may not be worth fixing. Fix the bugs and usability problems that will affect the majority of users, or which might cause bad press. Sometimes you'll guess wrong. Keep track of the bugs you defer and see if real customers report them, then escalate them to fix in the next version.

• Customer Feedback

Improving product development is based on a feedback loop. There are three main programs for incorporating customer feedback at different parts of the product cycle: user testing, beta testing, and (uh oh) customer support.

User Testing

Bringing in target users to give feedback on your product early in the development cycle can give you data about the intuitiveness of the design and validate whether your feature set is the right one. SQA engineers can also improve their tests by watching how real customers use the product. Fresh perspectives are always valuable, because the development team becomes too close to the product.

Beta Testing

Software used in real-world situations often provides some of the best feedback and bugs. During the Newton 2.0 OS project, most people on the Newton team carried around flash ROM MessagePads with the latest software. A lot of the good bug reports came from people using them in meetings. That was the difference between 1.x and 2.0 – you can actually take notes in meetings.

Customer Support

Product development doesn't end when you ship your product. Customer support quantifies customer reaction, and based upon call frequency you can prioritize bugs to fix in future revisions. Feature requests provide a source of good ideas from people actually using the product. The development of Newton 2.0 OS was largely driven from customer requests and usability problems, along with inspired design work from the software engineering team.

SPECIFICS OF NEWTON TESTING

Testing a Mac OS, Windows, or Newton application is all essentially the same. Take your software development and SQA background and overlay the technical details of the platform. When testing a Newton application, know how the Newton works and how it's architected. The best sources of information are the user's manual, *Newton Programmer's Guide* (*NPG*), and *Newton User Interface Guidelines.* Applications written for the Newton plug into a layer of system services. So, besides testing the features of your software in isolation, you need to test the areas where your application intersects with these system services.

Some of the important things to test are:

- Make sure the recognition view flags are set for the right kind of input.
- Test data storage and filing on an unlocked and locked card and internally to validate the soup code.
- Erase your application from extras to test its removeScript and optional deletionScript. Make sure the application removes itself from various registries to avoid the "grip of death."
- Eject a card with application data on it. Make sure the screen is refreshed.
- Test the differences when your application modifies a built-in prototype.
- If your application supports landscape, make sure all the dialogs fit.
- Test your application's use of Find – especially if it does something different like highlight the entry.
- Make your application the backdrop and see if it behaves correctly.
- If you link in custom dictionaries, test them in the various fields.
- Make sure soup change and other notifications get intercepted by your application.
- Optimize your program's use of frame and/or system memory.
- Make sure your application doesn't create global functions unless absolutely necessary.
- Test with the latest version of the system update. (The easiest way to remove a system update is to remove all batteries and short the backup battery terminals with a penny).
- Make sure your application uses screen-relative bounds to remain compatible with different screen sizes.
- Make sure the user interface of your application conforms to the guidelines.
- Delete your soup from the storage folder and see if your application handles it gracefully.
- If your application supports both 1.x and 2.0, test on both platforms.
- Test how your application reacts to its own data on a read-only 1.x and locked 2.0 formatted card in a 2.0 system.
- Use the Newton Keyboard with your application and make sure the tab order for views is correct.
- Test any application interfaces exported for public use.
- Test all peripherals you support.
- If your application runs on 1.x and 2.0, make sure it uses `gestalt` to test for the existence of Newton features.
- If your application modifies the behavior of a built-in application, then research existing applications and see if anyone else is doing what you're doing and document any incompatibilities.
- Make sure your application uses documented APIs or DTS-approved methods, for example, when accessing built-in application soup data. This will avoid incompatibilities in the future.

Applying the background and Newton-specific information, the following five steps provide the framework for testing a Newton application through its development:

1. Create a very detailed, hierarchical feature list for the product based upon documentation, prototypes, and communication with development engineers. Every feature that a user can access should be

represented, but don't go overboard on the details. Keep the feature list updated to reflect the current state of the product.

2. From that feature list, create test cases. For each specific feature there will be one or more test cases. Your base functional test cases should only be testing one thing. That way you can more easily isolate bugs and relate test cases to bugs. Be sure to go through the list of system services and create test cases for the way your application interacts with the rest of the Newton.

   Example: A feature in the Newton 2.0 Time Zones application is the ability to delete cities. Some test cases to derive from this feature would be:

   a. delete a built-in ROM city
   b. delete a city added by the user
   c. delete a city that was specified as a worksite
   d. delete a city and tap undo
   e. delete a built-in ROM city, then delete the Time Zones soup from the storage folder in extras

3. Identify a subset of your test cases as "quicklooks". These test the major areas of product functionality. When a new build of the software is handed off to SQA, you start your structured testing by running through your quicklook test cases. It's more productive to discover that a major part of your application is broken in the first 30 minutes of your test cycle than two days into it.

4. Determine the most effective test cycle time for normal engineering builds and milestone builds. You won't want to run all your test cases for every build. It's more effective to run all the quicklooks and then concentrate testing on the changed areas and bug fixes documented in the build release notes. You will want to run all your test cases at milestones such as alpha, beta, and final.

   Be sure to devote some time in each test cycle to *ad hoc* testing. Use your understanding of how your application works and how customers might use its features. If you've ever observed a user test, you're immediately struck by the fact that most users don't follow the "rules" – even when using the "simplest of features". Towards the end of testing 2.0 we placed more importance on *ad hoc* testing by splitting the test team into two groups, Validators and Exterminators (for lack of better names). The Validators performed the systematic testing of all the product features by running through all of the developed test cases, and the Exterminators were exclusively *ad hoc* testers with free reign across the whole system.

5. As the project progresses and the features of the software coalesce into stability, expand your testing into more stress and boundary conditions. The goal is to increase the robustness of the software in its ability to respond gracefully to unusual conditions. And since you can't test everything in every way that customers will use your product, this is one more thing you can do to minimize the risk of serious bugs in the field. A simple way to stress a Newton application is to see how it handles a lot of data. For example, the average Newton user probably has 100–200 entries in the Names application, but in 2.0 we tested with 1500 and sometimes more.

Conclusion

There's no guarantee that you'll ship quality applications even if you follow all of this advice. The glue holding everything together is caring – the pride in your work and the satisfaction of knowing you did your best. "Care and Quality are internal and external aspects of the same thing. A person who sees Quality and feels it as he works is a person who cares. A person who cares about what he sees and does is a person who's bound to have some characteristics of Quality." [2]

1. Pirsig, Robert. *Zen and the Art of Motorcycle Maintenance.* 1974. Pa NTJ

Newton®

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