Newton® Technology
JOURNAL

Volume II, Number 4 August 1996

Inside This Issue

Newton

NewtonScript Techniques

slimPicker: A Slimmer listPicker Proto

by Jeremy Wyld and Maurice Sharp, Apple Computer, Inc.

The Newton 2.0 OS provides many new prototypes for developers to use. One of the popular ones is `protoListPicker`. It was designed to provide a generalized framework and interface for presenting lists of choices to the user. Unfortunately, `protoListPicker` also tries to do everything for the developer. The result is a complex API, and overhead in space and time that is not necessary in a lot of cases. This article presents a slimmer picker.

The Data Is the Picker (listPicker)

Most of the overhead from listPicker is due to the way that it handles data. The purpose of listPicker is to display lists of data that the user can select items from. The data items can be elements in an array, soup entries, or a mix of both. To make the developers' life easier, listPicker requires some understanding of what the data is and how it is formatted. This is where the pickerDef and nameRef structures come from.

A nameRef is a generic data wrapper. It can be used to wrap an array element or a soup entry. All of the relevant data slots are part of the top level frame of the nameRef so that listPicker does not need to modify the actual data referenced by the nameRef. The pickerDef is the code object responsible for creating and managing nameRefs.

continued on page 3

Real World Newton

Practicum/power Pen: Four-Year Student Software Teams for Newton

by Jeffrey C. Schlimmer, Washington State University

New computer science graduates from most universities are not ready to go to work in modern software companies. As software professionals are quick to point out, these students have only learned how to write 300-line programs by themselves from scratch. These programs were class assignments that focused on elegance rather than efficiency or maintainability. The specifications came out of thin air, the code was poorly tested, the interface was a simple command line, and there was never any user's manual. The students did not learn how to reuse existing code or how to work in teams (that would be cheating). They did not learn how to trade-off features and deadlines with customers.

Enough battering of the current university system. That type of education provides basic knowledge of data structures and algorithms essential for engineering software. Our idea is to supplement the traditional, *fundamental* coursework with a four-year *practical* curriculum modeling the activities of a software company. From the university's point of view, entering freshmen are enrolled in a four-year, software engineering course called *Team-Oriented Software Practicum*.

continued on page 6

Newton Technology JOURNAL

Volume II, Number 4 *August 1996*

Published by Apple Computer, Inc.

Jennifer Dunvan • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

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

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

Contributors
Ryan Robertson, Jeffrey C. Schlimmer, Jeremy Wyld and Maurice Sharp,

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

John Kawakami • *Editorial Assistant*

Matt Neuburg • *Editorial Assistant*

Judith Chaplin • *Senior Art Director*

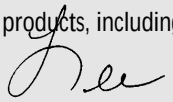
Editor's Note

Letter From the Editor

by Lee Dorsey and Jennifer Dunvan

They say that all good things must eventually come to an end. I say that all good things must eventually evolve into better things. And so it is with the *Newton Technology Journal*. Like everything and everyone at Apple Computer, we are in the process of evaluating programs, projects, and organizations and designing them to be better, more efficient and, above all else, significant in contributing to the success of Apple's key technologies and partners. This is great news for Apple's developer community and the projects, programs and publications that feed the success of this community. It is especially good news for Newton developers, who are finding renewed commitment and excitement around the Newton platform as a technology critical to Apple's future success. It is as we have always known it should be.

So, in the vein of making good things better, the Newton Systems Group is dedicating more resources to developer information, education, and support. Along with this comes new people with fresh ideas about improvement and expansion. While we continually receive feedback that the *Newton Technical Journal* is a valuable piece of your developer support portfolio, we are looking to improve it and grow it into a publication that serves your specific needs. With this goal in mind, it is with great excitement and expectation for such continued growth that I pass the editorial reins over to a new managing editor, who will bring to the publication fresh ideas, new enthusiasm and a close tie to Newton training and educational programs. All these things combined will undoubtedly result in a better, more efficient suite of developer education products, including an expanded *NTJ*. So, without further ado, let me welcome Jennifer Dunvan as the Managing Editor of the *Newton Technology Journal*.



continued on page 22

continued from page 1

slimPicker: A Slimmer listPicker Proto

The pickerDef and nameRef structure provides flexibility, but data that is only array based or only soup based pays the overhead for representing the mixed array/soup data. Every line of data displayed in the listPicker requires a nameRef structure which requires heap space. Each nameRef created requires several levels of function calls to the pickerDef object. Each access to a nameRef requires several levels of functions calls. This is true even for a simple access such as comparison. Caching the data in the nameRef can sometimes avoid the calling overhead, but it costs heap space.

The pickerDef/nameRef abstraction does provide some benefits. In addition to the obvious mixing of array and soup items, it also enables listPicker to render the individual data display lines with little developer intervention. It also makes filing easy. On the downside, the representation is quite brittle. The seemingly simple task of using an icon as the first item in the rendered line of data is actually very difficult to implement.

A hidden cost is the complexity of learning to use listPicker. To create a simple picker that displays developer data requires understanding protoListPicker, the pickerDef object (such as protoNameRefDataDef) and the nameRef wrapper. It also requires learning how these three entities interact. There is no clear delineation of data manipulation and display characteristics. A good example of this is single selection, which is a characteristic of the pickerDef not the listPicker.

Another hidden cost is the "cursor" used by the listPicker. In order to handle both array- and soup-based data, the listPicker must implement a pseudo-cursor that can wrap both types of data. Unfortunately, the details of the implementation are hidden. This means that it is very difficult to use listPicker on data that is represented across multiple soups.

If you are displaying lists of names, or if your data can be both array and soup based, listPicker provides a nice proto for you to use. However, for most developers, all they want is to display soup-based data in a list. Enter slimPicker...

You Data, Me Picker (slimPicker)

When we set out to design slimPicker, we set four goals (well five, see below):

1. Provide the listPicker look and feel.
2. Minimize space and time costs.
3. Make it easy for the developer to understand and use.
4. Allow the developer to customize it with minimal effort.

Originally, we had a fifth goal of keeping the same API as listPicker. We hoped that a developer could just substitute slimPicker for listPicker and everything would work. However, we dropped that requirement after finding that supporting the API required very similar overhead to listPicker. Most of the overhead came from an iterator that could work with both soup cursor and arrays.

Once we dropped the requirement for supporting the API, we also dropped the pickerDef and nameRef structures. This contributes to all four goals. Most of the overhead of listPicker is in the pickerDef/nameRef call chains, as is most of the complexity and brittleness of implementation.

The rest of this section gives some examples of how the goals effected the design and implementation.

Look and Feel

The look and feel was easy to do. We examined listPicker to see which individual elements were used in its construction. We had the source code, but you could do a similar thing using DV in the inspector. We used the same elements in slimPicker with only one notable exception: instead of using a `protoStaticText` for the selection counter, we draw it. This saves a bit of time and space (though not much).

The main part of slimPicker is implemented using a protoOverview because it can be used with either array- or soup-based data. It requires a cursor (or cursor-like) object for iterating over the data. The other nice feature of protoOverview is that it handles the selection check boxes.

The downside is that there is no supported way to cache the shapes used for each data display line. This results in a time penalty every time the display list of data lines is rebuilt. Unfortunately, this occurs any time you update the visual display list (that is, scrolling and the RefreshPicker call). Luckily, slimPicker does this faster than listPicker. If slimPicker were built into ROM, we could overcome this difficulty by using undocumented features. However, we did not use these features because they are likely to change in future Newton OS devices.

Space and Time

The main performance gains come from making the developers responsible for their data. There is no pickerDef or nameRef structure to provide a generic data wrapper. Instead there is a well defined interface between slimPicker and the data. The developers are responsible for rendering the shape that is a line of data. They are also responsible for providing the cursor structure used by protoOverview, handling verification, creating new items (including any editing slips) and maintaining a list of the current selections.

Note that eliminating the pickerDef also eliminates the popup and validation overhead. In listPicker, the only way to determine if a given item in a column requires a popup character is to build the popup. That means each line of data built by listPicker requires a call to MakePopup in the pickerDef. In slimPicker, if a popup is required, the developer renders the popchar into the display line for that data. They also need to detect if a hit to that line is in the popable item, pop the correct picker and handle the result.

Although it seems like we have added more work for the developer, it is actually easier to implement a simple soup-based slimPicker than an equivalent listPicker.

In essence, we speeded up slimPicker and reduced the space by removing the data abstraction layer. This does increase the work a developer must do, but it also removes most of the overhead as well as reducing the complexity of the proto.

Easy to Understand and Use

In addition to eliminating the pickerDef/nameRef/listPicker interactions, we also reduced the overall number of slots and methods that a developer needs to learn. As an example, all of the listPicker "suppress" settings are now in one bit field called `visibleChildrenFlags`.

Another good example is adding new items. In listPicker you had to enable the New button and then provide several methods in your pickerDef. In addition, the callbacks for adding the new item are sent to the pickerDef context, not to the listPicker. In slimPicker, you enable the New button and provide one method called `CreateNewItem`. On the downside, slimPicker will not do any work such as bringing up a slip or calling back when the slip is dismissed. Instead, `CreateNewItem` is called when the New button is pushed. Everything else is up to you.

The change that provides the most flexibility is letting you render a line of data. You provide an `Abstract` method that is given the data item and a bounding box and returns a shape that represents the data item. That means you can put anything in that shape that is required, including icons and columns. In listPicker, adding an icon was difficult at best. In slimPicker, just add it to the shape for your data line.

slimPicker also provides an `AlphaCharacter` call that lets you return the character used for sorting a particular item of data. In listPicker you would have to provide at least one method (possibly two) and set up the column description. If the first displayed column of data was not the one used for sorting, there are additional methods and slots that must be provided. This means things like displaying icons in the first column is very difficult. For slimPicker, you can display what you wish, and control the order with `AlphaCharacter`.

A downside of slimPicker is that the developer is responsible for tracking selection. The developer needs to provide the `IsSelected` and `SelectItem` methods that do the right thing. Each change in selected state will require an update in the visual display of the data lines, which means redoing the children of the overview (that is, a call to `RefreshPicker`).

Single select is relatively easy to implement, use a single slot to represent the selected item and refresh the picker. Your `SelectItem` method will replace the value of the slot and your `IsSelected` method will only return true if the entry passed in matches the one in the slot. The rest is handled by slimPicker.

Easy to Customize

Even though single selection is provided by listPicker (through the pickerDef!), it provides a nice example of how slimPicker is easy to customize. It also points out that, once again, eliminating the pickerDef/nameRef representation was a good decision.

Perhaps the biggest area of customization in slimPicker is the ability to change the data types and representation on the fly. In listPicker it is not possible to change the pickerDef or modify the cursor that access the data. The only way to do that is to close and open the listPicker. With slimPicker you have complete control over how the data is accessed (`cursor`) and how it is represented (`Abstract`). All that is required is a call to `RefreshPicker`, and everything will update.

Another example is filing. To add filing you just add the support you normally would in an application. That is, activate the folder tab (set the appropriate flag in `visibleChildrenFlags`), then provide the standard system API for filing (`appAll`, `NewFilingFilter`, etc.). When the filter changes, you can change your cursor and call `RefreshPicker`.

Six of One, a Dozen of Another

This section gives you some comparisons between listPicker and slimPicker. To be candid, the tests are stacked in favor of slimPicker. They are based on simple soup-based viewing.

All the tests were performed on the same MessagePad. MP 120, running Newton OS 2.0 and having 79 entries in Names. The listPicker-based FAX picker was opened from faxing a note and choosing "Other Names" from the Name picker. Heap space was measured before and after the picker was opened using HeapShow in the accurate setting with no timed updates. Timing started when the picker including "Other Names" was closed and stopped when the FAX picker was opened.

For the listPicker based People picker, we used the PeoplePicker-1 sample from Newton Developer Technical Support. Heap usage was measured using HeapShow. Timing started after the pen was released on the icon in the extras drawer and ended when the people picker appeared and was ready for input.

The slimPicker measurements were done on the protoSlimFaxPicker-1 and protoSlimPeoplePicker-1 code samples. These will be on Newton Developer CD #10 and on the web at http://dev.info.apple.com/newton/techinfo/slimPicker.html.

		listPicker	slimPicker
Heap Used	FAX picker	9300	3856
in bytes	People picker	8496	2740
Time to Open	FAX picker	5	3
in seconds	People picker	3	2

As you can see from the table, slimPicker is more efficient than listPicker in all cases. For heap usage this is not really surprising: listPicker has the overhead of nameRefs, a more complex selection tracking, extra cursors and a pickerDef. And slimPicker also has minimal extra information.

Time to open is a bit different. Both listPicker and slimPicker use protoOverview, but listPicker also has to construct the soup/array iterator and the nameRefs. In addition, each data access has the overhead of calling through the pickerDef object. slimPicker can just iterate over the visible data and call `Abstract` on each entry. There is very little housekeeping overhead.

Unfortunately, slimPicker is still fairly slow to launch. A quick profile of slimPicker shows that about 10% of the opening time is spent in the `Abstract` method of protoOverview. The other major piece is probably soup access. This means there is no effective way to speed up the code. The usual idea of caching is not useful since the slowness is part of the opening process. If the slowness was in scrolling, caching might makes sense, but of course that would increase the memory footprint.

One measure that is harder to estimate is the size of the object. We can get a good measure of slimPicker by either looking at the size of the package on the desktop, or by using TrueSize on the MessagePad. There is no similar way to find the size of listPicker.

API

This section presents the API to slimPicker. Of course, you will have all of the source code so you could modify anything. But just in case this code shows up in ROM someday, stick to the API.

Slots

`cursor`

This slot is required.

The iterator for the data displayed by the slimPicker. Can be either a soup cursor or a developer-defined object that implements the methods required by the cursor slot of protoOverview. See the Newton Programmers Guide or Newton Developer Technical Support Q&A for more information on the protoOverview cursor structure.

`folderTabTitle`

The text to put into the protoFolderTab. This is used to identify the slimPicker that is open. The default is NIL (i.e., no title).

You can use SetValue to change the value at runtime.

`reviewSelections`

If true, the slimPicker will only display the selected items. If NIL, all items will be displayed. Corresponds to the "Selected Only" checkbox in the user interface of the slimPicker. The default is NIL.

You can use SetValue to change the value at runtime.

`viewLineSpacing`
An integer representing the height of each line of data in the slimPicker. This value must be at least the height of the checkbox. The default is 14.

`visibleChildrenFlags`

Bit flags identifying which child views are to be visible. The values are:

Constant	Value	Shows/Hides
`vNewButton`	(1 < < 0)	New button for adding new data items
`vScrollers`	(1 < < 1)	Scrollers for scrolling the list
`vAZTabs`	(1 < < 2)	AZTabs for alphabetical navigation
`vFolderTab`	(1 < < 3)	Folder tab for filing
`vSelectionOnly`	(1 < < 4)	Selection Only checkbox
`vCloseBox`	(1 < < 5)	Close box for closing the slimPicker
`vCounter`	(1 < < 6)	Count of selected items

The default is all views visible.

Methods

`slimPicker:Abstract(entry, bounds)`

This method is required.

Returns the shape that represents the given entry in the slimPicker. The shape must not be larger than bounds. This is where the developer renders an individual line of data. The returned shape must be one that DrawShape can use.

continued on page 22

continued from page 1

Practicum/powerPen: Four-Year Student Software Teams for Newton

From the student's point of view, they have joined a software company called *powerPen*. The combination of fundamental and practical works well because students become highly motivated. For example, they want to learn in their English course because they want to write effective marketing material or a clear user's manual. They want to understand compilers because they want to add an authoring language to their application.

To be more specific, Practicum is roughly divided into two stages. In the first two years, student teams study activities that surround programming in a software company, i.e., marketing, testing, project management, user documentation, and technical support. Freshmen and sophomores can tackle these topics before they become effective programmers. Each semester the team of students studies a text or two on the topic and completes a related project for a regional software company, sometimes under nondisclosure. In return, the company provides written feedback on the student work. (We have found that students take constructive criticism from future employers much more seriously than from professors.) The goal of this first stage is to develop basic skills in various software development tasks. We want them to "walk in the shoes" of others that they will be working with when they become software developers.

In the second stage of Practicum, students are reorganized into product groups to build and support commercial-quality applications. Students in these product groups design, program, market, and support either a new version of an existing application or an initial version of a new application. This process includes the following steps: a user survey, a marketing requirements document (often done in conjunction with a freshman team), a design specification, a program, testing (with a freshman team), a user's manual (with a sophomore team), and a product release. The goal of this second stage is to build experience while developing complete, high-quality applications. We want them to see the whole software process in action, and occasionally, make mistakes before jobs are riding on their decisions.

After a year's experience, we found that students had trouble applying themselves to their industry projects in the first two years because we did not have a common technology base. Some of the students knew only Windows programming (and had PCs), some only Mac. To remedy this, we adopted Newton as our common technology with the help of Apple in early 1994. Apple donated a MessagePad for each of our students and helped us set up a senior-level course in mobile computing that focuses on Newton programming. The university purchased several Macs to host the Newton Toolkit and allocated space for a development lab. Students in Practicum/powerPen take the mobile computing course and gain common, essential skills for building modern, graphical user-interface applications. Newton has turned out to be an excellent choice because of the huge potential in the hand-held market and the ease of developing NewtonScript applications. It has also helped that the Newton developer community is small and cooperative. Our freshmen and sophomore teams have been able to do Newton marketing, testing, and documentation projects for Newton companies. They have also volunteered at Newton conferences and had summer internships with Newton companies.

Our first team started in 1992 with 10 freshmen. A second team followed the next year, and a third in 1995. Currently 17 students are involved in Practicum/powerPen: 5 seniors, 6 juniors, and 6 freshmen. Like many companies, we experience regular turnover, losing about 50% of the new students in the first year and another 25% of the overall group upon graduation. The students are currently supporting six Newton applications and developing a seventh. Each application is available in at least one of six foreign languages. They have over 500 registered users in 25 countries.

Our most significant marketing channel is the Internet, and in specific the World-Wide Web. We host a series of pages with the most current version of each application. We distribute applications for free because we want to have as many users as possible. These pages list known defects, feature requests, source code, functional specs, and other development information. We make our development information and source freely available because we want to help others understand how we are doing development – including other students not at our university. For applications with a significant content market, the pages also list content we and others have developed (e.g., plug-in dictionaries for hangMan or decks of cards for flashCard). Where possible, the pages list related applications commercially available from Newton companies.

Practicum/powerPen also provides an excellent opportunity to study and refine the software development process. Two areas in which we hope to contribute are localization and design methods. In localization, we have devised techniques for managing the user-interface strings of a Newton application so it can be easily converted from one human language to another. Our latest efforts make it possible for a non-programmer (but language expert) to specify strings by filling out a Web form, submitting it, and receiving a compiled Newton application correctly localized. This represents a dramatic savings compared to Apple's marketing estimates of US$10-25K to convert a Newton application into each new language. We are inviting our international users to test this system. It will leverage their expertise and provide additional international access to Newton applications.

In design methods we have adapted a functional specification technique for event-driven programming in general and Newton programming in specific. Based on a table structure, this specification type is easy to code against and provides a ready-made test plan. Our initial attempts with this methodology are also available on our Web pages (under flashCard).

As an educational project, Practicum/powerPen relies on industry support in three major ways. First, Newton companies can help by making a tax-deductible donation to purchase new hardware. With Apple's help we have recently upgraded most of the students to Newton 2.0 OS but we will fall short when new freshmen join Practicum/powerPen next fall. Second, companies can hire our students for internships or upon graduation. Besides the benefit of personal experience and growth, returning students share their expertise. Third, companies can propose and evaluate marketing, testing, or user documentation

Mini-Meta Data: Another way to get information to your PC

by Ryan Robertson, Apple Computer, Inc.

This article describes the development of a pair of applications that export data from a Newton device to a desktop computer. There are two applications: one that runs on a Newton device and one that runs on a desktop computer. They are designed to allow a developer to register data definitions so that soup information can be transferred between the Newton device and the desktop computer.

(Mini-MetaData is a Newton DTS sample that should be available by press time. You can find the Mini-MetaData source code on AppleLink and the Newton WWW Site, http://dev.info.apple.com/newton/newtondev.html. The next Newton Developer CD will also contain this sample.)

You Are the Connection

With all the additions to the Newton 2.0 OS, it may seem like exporting data to your desktop computer has been overlooked: it has not: the intention is for application developers to incorporate the Desktop Integration Libraries (DILs) into their existing applications. This approach will allow a user to directly connect to their Newton device using their favorite desktop application. Before the DILs became available, the user was required to use a second application called Newton Connection Kit to transfer their Newton device's data to a more generic format that could then be used by desktop applications.

The Desktop Integration Libraries are a set of platform-independent C libraries and APIs that can be easily incorporated into an existing application. They provide all the necessary support to connect to your Newton and to transfer data between a Newton device and a desktop computer.

There are currently two types of DILs: the communication DILs (CDILs), and the frame DILs (FDILs). The communication DILs are used to open a connection with the Newton and to read and write bytes of data. The frame DILs let you read and write other Newton data types, such as frames and arrays.

The two largest advantages to using the DILs are:

1) The DILs abstract all underlying transport details into an easy to use API.
2) Your code will run on Mac OS™ and Windows™ with very little modification.

The implementation of the Newton application described in this article is intended to be as generic and extensible as possible and allows a developer to register information about how to format data before it is sent to the desktop computer. By doing this, the Newton application can export data from many different soups using many different formats (this format will be explained below in further detail). For instance, you will be able to export your Names file to a tab-delimited format that could be read into a database or a spreadsheet.

The desktop application will take the incoming data and dump it into a text file. It should also be designed so that it will easily port to other platforms. This means that the user interface code will be separated from the implementation code. This implementation only deals with text data, so the FDILs are not needed.

I'll begin this discussion by describing the protocol used for transferring data between the Newton device and the desktop computer. After that, I'll go into more detail of some of the major design decisions for both the Newton application and the desktop application.

Yakity Yak, Do Talk Back

The protocol used for sending and receiving data is fairly simple. First I will discuss the Newton side of the protocol.

At various times during the connection, the Newton will send one of three things: a command code, a string length, or a string. The command codes have been purposefully selected as large numbers so as to avoid conflict with the string length. Table 1 summarizes the command codes sent from the Newton during the protocol.

Table 1. Command Codes Used by the Newton Protocol

Command code name	Command code value	Description
`kNewtonCancelled`	`0x0FFF`	Sent when the user presses the "Cancel" button on the Newton.
`kNewtonFinished`	`0x0FFE`	Sent when all of the data has been transferred to the desktop application.

Command codes are only sent when the user is canceling the operation or the export has completed. The rest of the protocol on the Newton side consists of sending strings to the desktop computer. In our protocol, the length of the string is sent first to let the desktop application know how large the receiving buffer needs to be. By using this technique, we guarantee that there will be no ambiguity as to whether the received data is a command code or a string length.

Here is the C function that reads data from the CDIL pipe on the desktop. It returns a value indicating whether the read was a success, a failure, a cancel, or whether the Newton is ready to disconnect.

```
long ReadBuffer( LPSTR bufferPtr, long* length )
{
  Boolean  eom;
  CommErr  anErr;
```

```
  long     command;

  // read the first four bytes, this will either be a command code or a string length
  *length = 4;
  anErr = CDPipeRead( gOurPipe, &command, length, &eom, 0, 0,
                      kPipeTimeout, 0, 0 );
  if (anErr) {
    return kReadError;
  }

  // interpret the command code and act on it. If the data was not a command code,
  // then it is a string length, so read in the string
  if (command == kNewtonCancelled)
    return kNewtonCancelled;
  else if (command == kNewtonFinished)
    return kNewtonFinished;
  else if (command) {
    *length = command;

    // resize the buffer to the size of the string plus one for the null character.
    if ( realloc( bufferPtr, command+1 ) ) {
      anErr = CDPipeRead( gOurPipe, bufferPtr, length, &eom,
                          0, 0, kPipeTimeout, 0, 0 );
      if (anErr)
        return kReadError;

      bufferPtr[(*length)] = (char)0;// Null terminate the string
      return kReadSuccess;
    }
  }

  return kReadError;
}  // ReadBuffer
```

The desktop PC side of the protocol consists of four command codes which are summarized in Table 2.

Table 2. Command Codes Used by the Desktop PC Protocol

Command code name	Command code value	Description
kHelloCommand	0x0FFD	Sent when the connection has been established. This tells the Newton application to start the protocol.
kGoCommand	0x0FFC	Sent when the desktop application is ready to start receiving the export data.
kAckCommand	0x0FFB	Sent after the desktop has successfully received a line of data.
kErrorCommand	0x0FF9	Sent if there is an error during the connection.

Because most of the data transfer consists of the Newton device sending data to the desktop machine, the Newton application uses only one input specification for the entire protocol.

```
{form:     'number,

  InputScript: func( ep, data, termination, options ) begin
    if data = kHelloCommand then begin // Hello command was received,
                                       // start the protocol.
    ep:DoEvent( 'StartProtocol, nil );
    end else if data = kGoCommand then begin
                                       // go command was received.
                                       // Initialize and output the first line
      ep._parent.fStatusView:StopBarber();
      local numEntries := ep.fCursor:CountEntries();
      ep:Parent().fStatusView:GoGoGadgetGauge(
          numEntries, kSendingDataString );

    ep:DoEvent( 'OutputData, nil );
  end else if data = kAckCommand then
                                       // ack command was received,
                                       //output the next line
    ep:DoEvent( 'OutputData, nil );
    else begin                         // There was an error, so disconnect
      GetRoot():Notify(
          kNotifyAlert, kAppName, kProtocolErrorString );
      ep:DoEvent( 'Cancel, nil );
      ep:DoEvent( 'Disconnect, nil );
    end;

    nil;
  end,

  CompletionScript: func( ep, options, result ) begin
    ep:DoEvent( 'Disconnect, nil );
  end,
}
```

If an unknown command code is received on the Newton device, the Newton application signals a cancel and disconnects. An unknown command code is likely to be caused by a communications error. If the protocol were more robust, the Newton could try to resync with the desktop machine and start sending data again

Protocol of the Wild

Once the connection has been established, `kHelloCommand` is sent from the desktop PC to the Newton device. Seeing the `kHelloCommand`, the Newton application will send the name of the application for verification purposes. This name is checked on the desktop PC to make sure the connection is with the Mini-MetaData application and not with the Newton's built-in Connection application or with the Toolkit App.

The next step is for the Newton application to send the name of the file that data will be exported to. Once the desktop PC receives this name, the standard save dialog will be opened with that file name as the default.

When the user finishes selecting the target file, the desktop application will send `kGoCommand` indicating it is ready to begin receiving data.

At this point, the Newton application begins sending data in the following pairs: a string length followed by the string. When the desktop successfully receives and writes this string to the file, it will send an `kAckCommand` to the Newton to signal that it is ready for more data.

Finally, the Newton application sends `kNewtonFinished` when it has finished transferring data. It then disconnects.

If the desktop encountered an error during the protocol, it will send `kErrorCommand` to the Newton and disconnect.

Here is an example of the protocol in action:

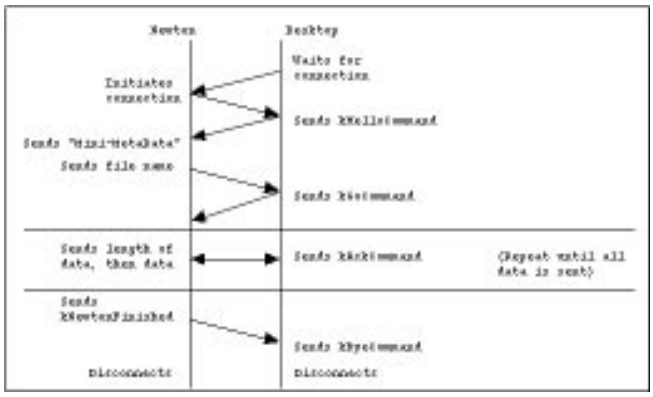


Figure 1. Newton-Desktop Communication Protocol

Now that you understand the protocol, lets dive into the code on the Newton.

NEWTON SIDE UP

To extend the mini-meta data application, you will add a format frame to a global registry. The format frame includes such information as which soup to send data from, what the query specification is, and how to create a formatted string from a soup entry. This registry will be discussed in more detail below.

The Newton application handles the format information and provides a simple interface for selecting which format to use. To keep the implementation as generic as possible, a form of meta data was created. Using this meta data, a developer can have a maximum amount of control over the format of outgoing information without explicitly having to know much information about the Newton storage or communications systems.

Here is a screen shot of what the Newton interface looks like.

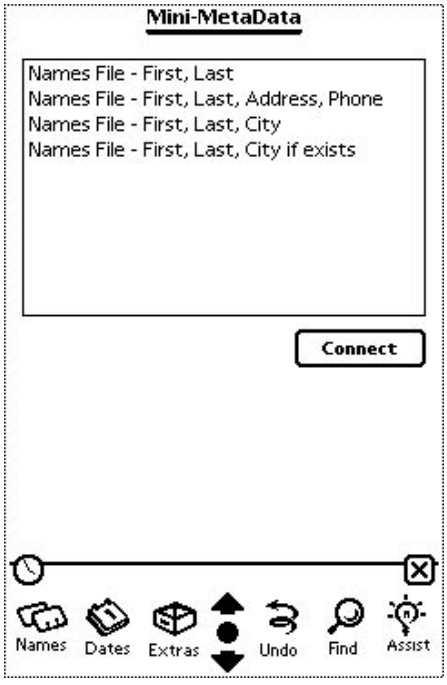


Figure 2. The Mini-MetaData User Interface

The NTK Project for the Newton application consists of 10 files, 4 of which are layout files.

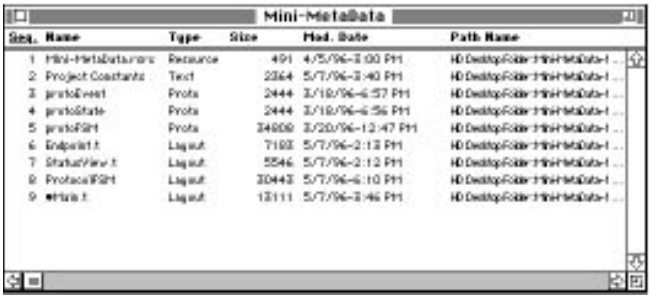


Figure 3. The NTK Project Window for the Mini-MetaData Application

The important files to look at are: Endpoint.t, StatusView.t, ProtocolFSM, and Main.t.

The hierarchy of the Newton application is illustrated in Figure 4.

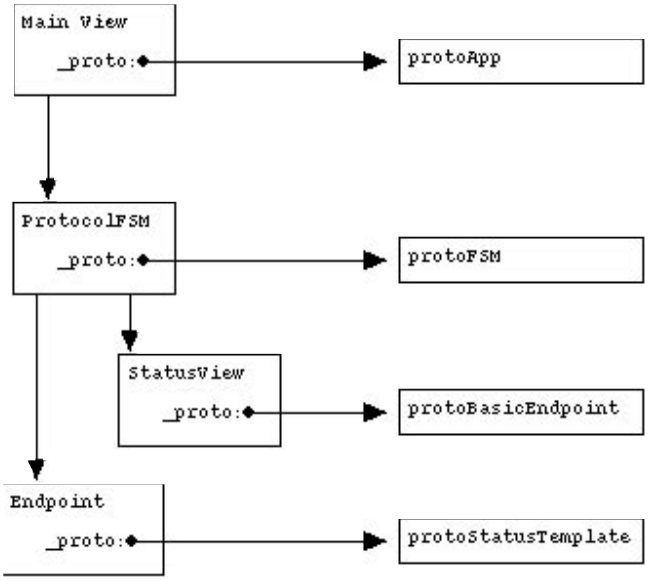


Figure 4. Hierarchy of the MiniMetaData Application

GOGOGADGETSTATUSVIEW

It is very important to give the user feedback during the connection. Newton 2.0 OS provides a terrific proto, `protoStatusTemplate`, for conveying status information to a user. StatusView.t contains the template for the status view that is used during the connection. One of the beauties of using `protoStatusView` is that it has multiple personalities. Among other things, a view based on protoStatusView can be a single line of text, a barber pole, or a gauge. During our connection we will use all three of these.

The barber pole element is used during the connection phase. The barber pole was chosen because the time it takes to connect is not a known value, and a simple line of text doesn't necessarily give the user the impression that a lengthy operation is taking place. During the connection phase, the user may forget to signal a "wait for connection"

event on the desktop which leaves the Newton waiting until the connect request times out.

The gauge element is used while data is being sent. Because we know the number of items that will be sent, a deterministic interface element is a more appropriate choice here.

The simple status view is used for disconnecting. A barber pole was not used because the disconnect operation is usually very fast. The disconnect operation will also complete successfully regardless of whether the desktop computer is disconnecting.

The status view template has three main methods of interest. They are: `GoGoGadgetBarberPole`, `GoGoGadgetGauge`, and `GoGoGadgetSimpleStatus`. Each of these methods will set up the status template with the correct information and open it if necessary.

There are also some additional methods for updating the text, the gauge, and the barber pole once the view has already been opened.

Back to the Basics

The mini-meta data application uses `protoBasicEndpoint` as the prototype for the connection endpoint. Using `protoEndpoint` is not recommended, and is actually impossible to use in a "2.0 only" application. This new endpoint proto is much more reliable and functional than `protoEndpoint`.

Endpoint.t contains the template for our endpoint. In addition to the standard endpoint methods, there is one other method of interest: `OutputLine`. `OutputLine` calls a helper function to format a soup entry into an output string (this method will be discussed in more detail later). It then outputs that string and updates the status view.

Here is the definition of OutputLine:

```
func() begin
  local entry := fCursor:Entry();

  // if there is an entry, then output the next line of data.  Otherwise,
  // output kNewtonFinished and disconnect.
  if entry then begin
    fData := :CreateStringFromEntry( entry, fMetaDataFrame );
    fCursor:Next();

    // Output the length of the data then output the data.    If either
    // output fails then post a 'cancel event.
    :Output( StrLen(fData), nil,
          {async:  true,
          form:    'number,
          CompletionScript: func( ep, options, result)
          begin
            if NOT result then
              ep:Output( ep.fData, nil,
                    {async: true,
                      form: 'string,
                      CompletionScript: func(
                                    ep, options, result)
                      begin
                        if NOT result then begin
                        ep._parent.fStatusView:UpdateGauge();
                        ep.fData := "";
                        end else
                        ep:DoEvent( 'Cancel, nil );
                      end,
              } );
            else
                ep:DoEvent( 'Cancel, nil );
          end,
  } );
  end else begin
    // Output kNewtonFinished command and disconnect when the Output completes.
    :Output( kNewtonFinished, nil, {async: true,
                      form: 'number,
                      CompletionScript: func(
                                    ep, options, result )
                                    begin
                                      ep._parent.fStatusView:FinishGauge();
                                      ep:DoEvent( 'Disconnect, nil );
                                    end;
                                    } );
  end;

end
```

Holy Finite State Machines Batman!

Using a deterministic finite-state machine for communications was covered in depth in the April 1996 issue of NTJ (volume II, issue 2). This application leverages off of the sample code produced for that article. The file of interest is ProtocolFSM which has the layout of all the states and events needed for our application.

There are three events worth pointing out. The first event is the 'Create' event in the Genesis state. This event sets up the endpoint, the status view, and registers a power off function. Any initializations needed for the connection should be done here.

Next we have the 'Connect Success' event in the 'Connect' state. This event sets the input specification for our protocol, and also has the definition of our input specification in the `fInputSpecification` instance variable. This event is performed once there has been a successful connection with the desktop computer.

Finally, we have the 'OutputData' event in the 'Connected' state. This event simply calls the endpoint's `OutputLine` method described above. So why is this event of interest to us? Another possible implementation for outputting data would have been to call the `OutputLine` method directly from the input specification. Doing this would remove an event from the state machine, and make the code more centralized. However, by placing the `OutputLine` method in an event, canceling functionality is provided for free. When the finite state machine receives a cancel event, all posted communications requests will be canceled, including the input specification.

By using the finite state machine sample, the code is more understandable, and more modular. This type of modularity provides an almost complete separation between the interface code and the communications code. Having this separation will make future revisions easier.

All communications code on the Newton side is asynchronous. This decision was made because synchronous comms are generally evil. When you post a synchronous comms request on the Newton, an additional task is created – that's Newton lingo for a new thread. This adds needless overhead to the system, and can potentially reveal some interesting problems. For instance, you may be outputting lots of data in a loop using synchronous output requests. Each time through the loop a new task will be created, which is a rather expensive operation. The new task will take up system memory, and will not release control until it returns to the main event loop (which does not happen until you are finished with your output loop). As a consequence, the Newton will eventually run out of system memory and come crashing to its knees. Another drawback of using synchronous comms is that the user loses control of their Newton while the comms request is waiting to complete.

Grand Central

Our main layout file is main.t. This file contains the code for selecting a format, and creating an output string from a soup entry.

The important function to look at is `CreateStringFromEntry`. This method is called repeatedly during the protocol. It is passed a soup entry and will return a string representation of that entry by using the format frame. It iterates over the field array in the format frame, building a string from the elements of that array.

```
func( entry, metaFrame )
begin
  local line, lineItem, result;

  line :=  foreach lineItem in metaFrame.fields collect begin
         // build the item string from the meta data frame.

         // if lineItem is a path expression, the resolve it and return the value
         if ClassOf( lineItem ) = 'pathExpr OR
         ClassOf( lineItem ) = 'symbol then begin
           if entry.(lineItem) then
             entry.(lineItem) & metaFrame.itemSeparator;
           else
             metaFrame.emptySpace & metaFrame.itemSeparator;
         end else if IsFunction(lineItem.format) AND
                     HasSlot( lineItem, 'pathExpr ) then begin
           // if we have a format function then pass in the value found using
           // the pathExpr slot to the function.
           result := call lineItem.format with (
                       entry.(lineItem.pathExpr) );
           if result then
             result & metaFrame.itemSeparator;
           else
             metaFrame.emptySpace & metaFrame.itemSeparator;
         end else if lineItem.format = 'quotedString AND
                     HasSlot( lineItem, 'pathExpr ) then begin
           // if format is 'quotedString, then quote the value found using
           // the pathExpr slot.
           result := result;
           if result then
             $" & result & $" & metaFrame.itemSeparator;
           else
             metaFrame.emptySpace & metaFrame.itemSeparator;
         end else if lineItem.format = 'quoteIfExists AND
                     HasSlot( lineItem, 'pathExpr ) AND
                     entry.(lineItem.pathExpr) then begin
           // if format is 'quoteIfExists then quote if the value found using
           // the pathExpr slot exists
           $" & entry.(lineItem.pathExpr) & $" &
                           metaFrame.itemSeparator;
         end else
           metaFrame.emptySpace & metaFrame.itemSeparator;
       end;

   // return a string with the proper line separator
   return Stringer( line ) & metaFrame.lineSeparator;
end
```

Don't Forget the Desktop

As discussed earlier, the desktop application uses the DILs to transfer data between the Newton device and the output file. The requirements of this application were simple enough that only the CDILs were needed.

To help in the effort to create cross platform code, the project is broken into two C files. There is a file for the main OS event handling code and a file for the protocol code. They are Interface.c and Protocol.c. The event code and the dialog code is not cross platform because much of that code is specific to either platform. The protocol code is cross platform and consists of the code to open the connection with the Newton, handle the protocol, and close the connection.

There are four functions in Interface.c that are not used for handling OS events. They are `CreateNOpenFile`, `WriteToFile`, `UpdateNCloseFile`, and `InitializePipe`. The first three are not in Protocol.c because they contain file access routines that are specific to one platform. Why `InitializePipe` is not in Protocol.c is not as obvious: the underlying transport options are specified slightly differently depending on whether you are running on MacOS or on Windows.

OS Event Handling

The interface code is in the Interface.c file if you are using MacOS and is in the INTERFAC.C file if you are using Windows. These files contain all the standard event handling code and should probaby look pretty familiar. In addition to the above mentioned functions (`CreateNOpenFile`, `WriteToFile`, `UpdateNCloseFile`, and `InitializePipe`), the MacOS code contains one other function of interest: `SetupPortMenu`. This function correctly creates a list of the ports available on the given machine. For instance, most Macintosh's have a printer and a modem port. However, if the user is running on a Duo there is one printer/modem port.

Protocol.c

This file contains all the code necessary to handle the protocol and the various states of the connection. It also contains the code to handle error reporting to the user. Most of the functions and procedures in this file are easy to understand. However, there are a couple of areas that warrant further discussion.

The procedure that handles most of the protocol is `DoProtocol()`, and is defined as follows:

```
void DoProtocol()
{
  StandardFileReply fileReply;
  short      fileRef = 0;
  long       length;
  char       *bufferPtr = NULL;
  long       fBufferResult;
  long       anErr;

  // preallocate a buffer that we think will be large enough for most data.
  // This buffer will be resized as data is received.
  if ( !(bufferPtr = malloc( 256 )) ) {
    ConductErrorDialog( kNoMemoryString );
    return;
  }

  // Send kHelloCommand to the Newton
  fBufferResult = WriteCommand( kHelloCommand );
  if (fBufferResult == kWriteError)
    Fail(FailWrite);

  // Make sure the we have connected to the Mini-MetaData app on the Newton
  fBufferResult = ReadBuffer( bufferPtr, &length );
    switch (fBufferResult) {
      case kReadSuccess:
        if ( strcmp( kHeloResponse, bufferPtr ) ) {
          Fail(FailWrongApp);
        }
        break;
      case kNewtonCancelled:
      Fail(NewtonCancelled);
      case kReadError:
      Fail(FailRead);
    }

  // Read the filename to save the incoming data to
  fBufferResult = ReadBuffer( bufferPtr, &length );
    switch (fBufferResult) {
      case kNewtonCancelled:
        Fail(NewtonCancelled)
      case kReadError:
      Fail(FailRead);
    }

  // create and open the file, then start dumping data into it.
  anErr = CreateNOpenFile( bufferPtr, &fileReply, &fileRef );
  if ( anErr == noErr ) {
    fBufferResult = WriteCommand( kGoCommand );

    if (fBufferResult == kWriteError) {
      UpdateNCloseFile( fileRef, &fileReply );
      Fail(FailWrite);
    }

    // Loop until there is either an error, or until the Newton sends a cancel
    // command or a finished command
    while( true ) {
      CDIdle( gOurPipe );
      fBufferResult = ReadBuffer( bufferPtr, &length );

        switch (fBufferResult) {
          case kReadSuccess:
            anErr = WriteToFile( fileRef, &length, bufferPtr );

            // if there was an error writing to the file, close the file, display
            // an error and return.
            if (anErr) {
              Fail(FailWriteFile);
            }

            // send an kAckCommand, if there was an error then handle it.
            fBufferResult = WriteCommand( kAckCommand );
            if (fBufferResult == kWriteError) {
              UpdateNCloseFile( fileRef, &fileReply );
              Fail(FailWrite);
            }
            break;
          case kNewtonCancelled:
            UpdateNCloseFile( fileRef, &fileReply );
            Fail(NewtonCancelled);
          case kNewtonFinished:
            ConductErrorDialog( kDownloadWasSuccessful );
              UpdateNCloseFile( fileRef, &fileReply );
          free( bufferPtr );
          bufferPtr = NULL;
          return;
          case kReadError:
            UpdateNCloseFile( fileRef, &fileReply );
          Fail(FailRead);
        }
    }
  }

  WriteCommand( kErrorCommand );
  free( bufferPtr );
  bufferPtr = NULL;
  return;

  // These are the Goto locations that are jumped to using the Fail() macro.
  FailWrite:
      WriteCommand( kErrorCommand );
    ConductErrorDialog( kBufferWriteErrorString );
    free( bufferPtr );
    bufferPtr = NULL;
    return;

  FailRead:
      WriteCommand( kErrorCommand );
    ConductErrorDialog( kBufferReadErrorString );
    free( bufferPtr );
    bufferPtr = NULL;
    return;

  NewtonCancelled:
    ConductErrorDialog( kNewtonCancelledString );
    free( bufferPtr );
    bufferPtr = NULL;
    return;

  FailWriteFile:
      ConductErrorDialog( kFileWriteErrorString );
      WriteCommand( kErrorCommand );
      UpdateNCloseFile( fileRef, &fileReply );
      free( bufferPtr );
    bufferPtr = NULL;
    return;

  FailWrongApp:
      ConductErrorDialog( kWrongAppString );
      free( bufferPtr );
    bufferPtr = NULL;
    return;

}  // HandleProtocol
```

`ReadBuffer` and `WriteCommand` are helper functions used to read to and write from the CDIL pipe.

The return value is checked to make sure there were no errors in the protocol. If an error occurred, it is assumed that there is a problem with the connection, and the connection is aborted. In an effort to retain some amount of synchronicity, `kErrorCommand` is sent after an error has occurred. There is a good chance that the `kErrorCommand` may not be sent because the original error was a communications error. A more robust protocol would examine the error value and take appropriate action based on that value. It may be possible to recover from the error and continue receiving data.

Registering the Meta Data

To extend the mini-meta data application, you will add a format frame to a global registry. The symbol for this registry is `'|MiniMetaDataRegistry:DTS|`. Here is an example of how you add a format frame:

```
local registry;
if GlobalVarExists( '|MiniMetaDataRegistry:DTS| ) then
  registry = GetGlobalVar( '|MiniMetaDataRegistry:DTS| );
else
  registry := DefGlobalVar(
```

```
      EnsureInternal('|MiniMetaDatRegistry:DTS|),
            EnsureInternal([]) );

AddArraySlot( registry, myFormatFrame );
```

Here is how you would remove your format from the registry:

```
if GlobalVarExists( '|MiniMetaDataRegistry:DTS| ) then begin
  local registry = GetGlobalVar( '|MiniMetaDataRegistry:DTS| );
  local pos :=
      LSearch( registry, myFormatSym, 0, '|=|, 'symbol );
  if pos then
    RemoveSlot( registry, pos );
end;
```

Here are some examples of what a format frame might look like:

```
{title: "Names File - First, Last",
  symbol: '|Format1:DTS|,
  soupName: "Names",
  lineSeparator: unicodeCR,
  itemSeparator: ",",
  emptySpace: " ",
  fields: ['name.first, 'name.last]}

{title: "Names File - First, Last, Address, Phone",
  symbol: '|Format2:DTS|,
  soupName: "Names",
  fileName: "Names Export",
  fields: ['name.first, 'name.last, {format: func(s)
             if s then CapitalizeWords(s) else nil,
             pathexpr: 'address},
  [pathexpr: 'phones, 0]]}

{title: "Names File - First, Last, City",
  symbol: '|Format3:DTS|,
  soupName: "Names",
  itemSeparator: ",",
  lineSeparator: unicodeCR,
  emptySpace: " ",
  fields: ['name.first, 'name.last,
      {format: 'quotedString, pathexpr: 'city}]}
```

Each MetaData frame must have the following slots: title, symbol, fields, either GetSoupName or soupName, and either GetQuerySpec or querySpec. It may optionally have a lineSeparator slot, emptySpace slot, and an itemSeparator slot.

Here is a more in-depth description of each slot:

`title`
The name of this particular meta data frame. This is the name that will appear to the user in the list of installed meta data frames.

`Symbol`
This is a unique symbol used to identify this particular meta data frame. If you register two meta data frames with the same symbol, the second one that is installed will overwrite the first one. Append your developer signature to the symbol.

`soupName`
The name of the soup to export data from, or nil.

`GetSoupName`
If you cannot know the name of the soup at compile time, specify this slot instead of the soupName slot. GetSoupName will hold a function

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

Newton Programming: Communications Overview

Newton Developer Training

The Newton operating system is designed with communications as an integral part of the system. The pervasive approach is that whatever you can see in a Newton application, you can send. Part of this approach is the notion that, as much as possible, the user will have a very similar experience in sending data regardless of the medium used to send it.

From a programming point of view things are not quite so simple but the architecture is designed in a layered way so that little programming is required unless the situation is fairly unusual. In other words, there is a great deal of built-in communications software in the Newton which can be used to provide basic communications functionality for almost any program.

Figure 1 shows the various layers comprising the Newton communications system and the programming interfaces used to access these elements. The rest of this article gives brief descriptions of these APIs as well as providing references to more details about them.

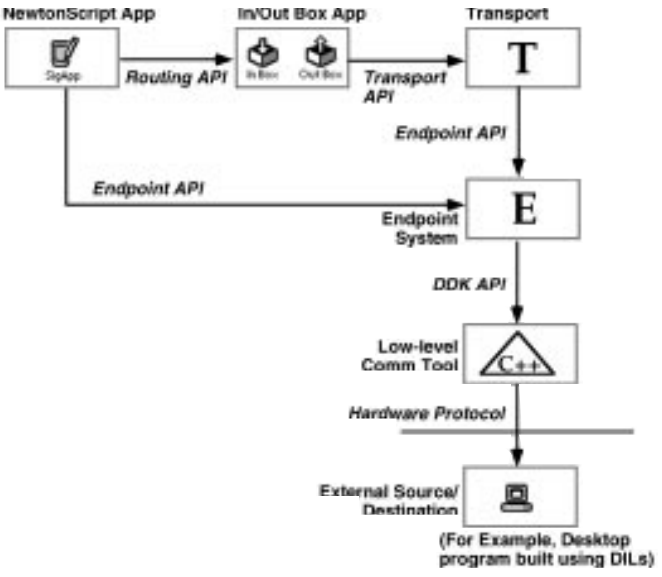


Figure 1: Newton Communications Layers

The following is a brief description of the items shown in Figure 1 and their APIs.

A NewtonScript application using the Routing API is the simplest way for application programmers to provide communications support from a Newton device. Any application written in NewtonScript can use communications modules which have been installed as Transports. In Newton 2.0 OS, this includes the built-in transports for beaming, faxing, mailing, and printing. To use the available transports, the Routing application programming interface (API) is used to specify which data is being "routed", what form it takes and how it should appear (for example, print format). The Newton 2.0 OS uses a store-and-forward model for this kind of communications and the In/Out boxes are where incoming or outgoing data is stored in the routing model.

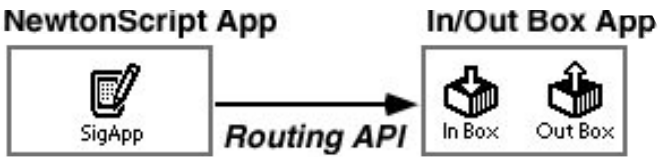
In/Out Box Application and Transport API. Built into the system is the In/Out Box application which manages the soups used to store incoming and outgoing data. This application communicates with one of several Transports, which consist of code provided to move the data to or from the appropriate destination or source. While there are several built-in transports in the system, NewtonScript programmers may write their own transport to provide system-wide data management.

Endpoint API and Endpoint System. The Endpoint API is a NewtonScript interface for performing direct communications with the outside world. Applications programmers may add endpoint code to their programs to communicate directly with an external source or destination. An example of this might be endpoint code which communicates directly with a GPS device on demand. Transports have endpoint code to move the data they receive out to an external destination or to receive data from an external source prior to passing it to the In/Out Box Application.

Low-level Communication Tools. These tools are implemented in C++ and actually communicate with the C++ interfaces to the hardware drivers. While these interfaces are not yet available, they will be published in the future.

Each of the APIs will be described in more detail in the remainder of this article.

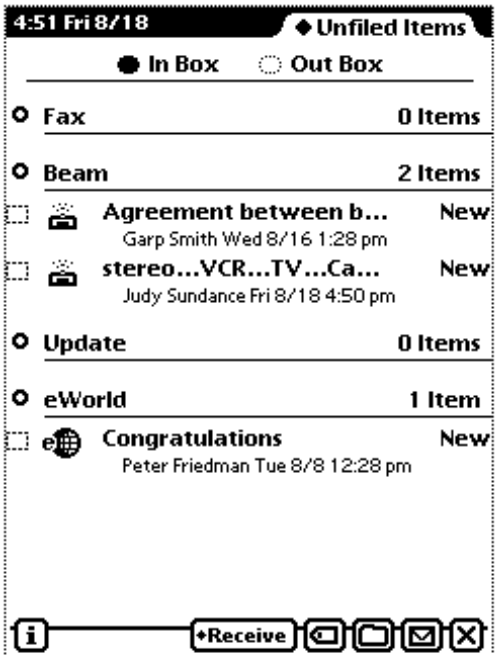
Routing



The Newton OS is provides a store-and-forward model for communications and uses the In/Out Boxes as the place where target data is stored. The term **store and forward** means that messages are routed (directed) to a distant communications device or from such a device to a Newton application through an intermediate holding area (the In/Out Boxes). Target data is any piece of information which is being routed in or out of the Newton.

The In/Out Boxes are actually a single application which provides the storage in the form of a soup, the functionality of sending and

receiving messages, and the interface that lets the user look at pending message and dispose of them as he or she desires. Figure 2 shows what the In/Out Boxes might look like when there are messages pending. Note that at any time the user can switch from In Box to Out Box and vice versa via the radio buttons at the top of the view.



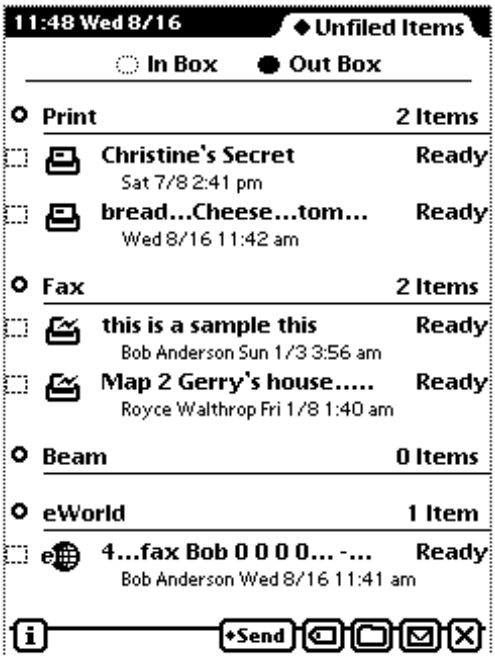


Figure 2. The In/Out Box User Interface

Routing is usually triggered by using the Action button that is displayed in the view from which something will be routed. The Action button is displayed in the view as a pop-up which shows available user actions as illustrated in Figure 3. Some applications will have one Action button in the status bar, others will have one in each of several views. The Names application is an example of a single Action button because normally only one name at a time is viewed. The Notes application has an Action button attached to each note since there may be many notes on the screen at any given time.

The Action button is created on screen by adding the prototype `protoActionButton` to the desired view.

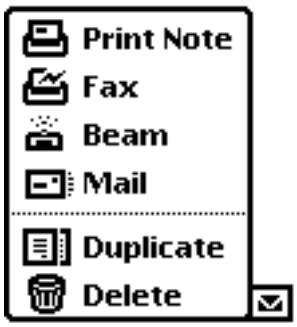


Figure 3. The Action Button Popup

Each target object which is routed must have a meaningful class. For frames, this means that the frame must have a class slot which identifies the type of data associated with this kind of object. Normally, each application will supply its own class of data for routing, such as `'|myData:MYSIG|`. This class is used by the system to look up in the Data View Registry the list of routing frames which may be used to route data of a specific class. From these routing frames, a list of transports or communication methods (for example, faxing, printing, beaming) which can route the target data are supplied to the Action button. The net result of this is that when the user taps on the Action button a list of the destinations appear which are appropriate for the target data.

Figure 4 shows how this is all interconnected. An application, usually in its InstallScript, will put one or more frames named for the classes of data which it will route into the Data View Registry. These Frames will consist of one or more Routing Frames which describe what format the target data can take when it is routed. The system uses this to search the list of installed transports and, when it finds a transport which supports one of the routingTypes, adds the transport name to the list to be displayed in the Action button.

So in the example shown in Figure 4, the application has installed a frame `|forms:MYSIG|` in the View Definition Registry which supports dataTypes of 'view, 'frame and 'text. These are used to choose the transports for printing, faxing, beaming and mailing so these appear in the Action button the user has pressed. Note that it doesn't pick up the compress transport whose dataType is 'binary.

When the user selects a transport from the Action button, an appropriate routing slip is displayed and all formats that in which the data can be displayed are displayed in the format picker as shown in Figure 5. Formats describe how the target data should be organized before sending it onward to the appropriate destination. For example, when printing, there might be several formats such as letter, memo, two-column, and so on, which describe how the target data will be

printed.

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

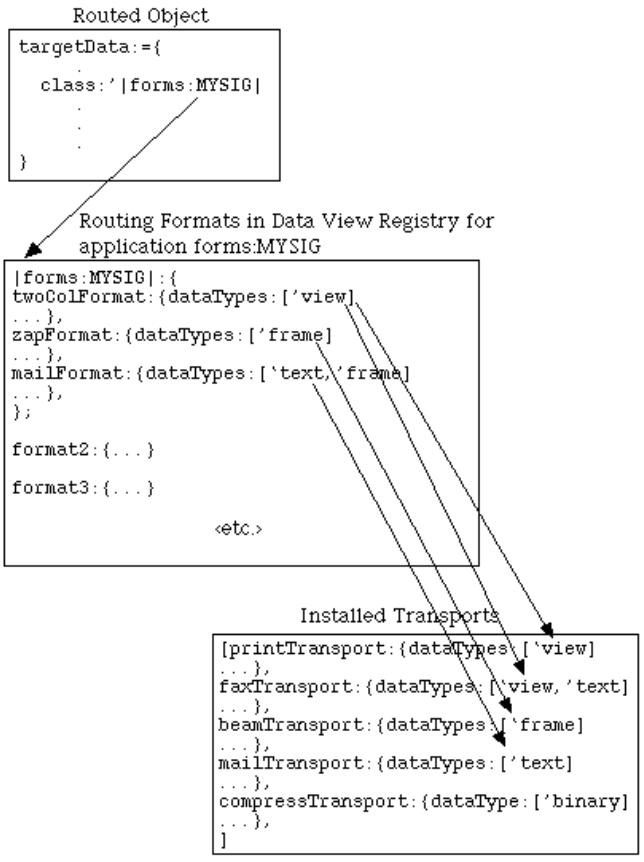


Figure 4. The Routing System

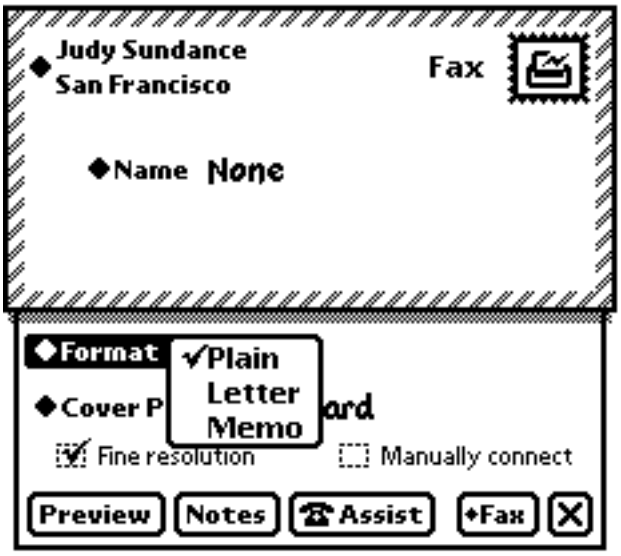
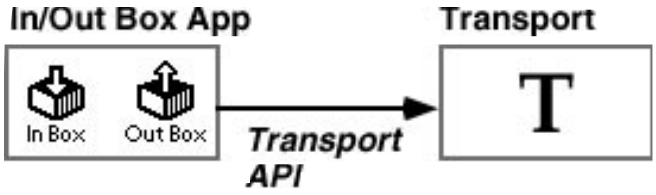


Figure 5. The Format Picker

TRANSPORTS



The simplest definition of a transport is – something to which data can be routed. But a more useful definition is that a transport is a globally available service offered to applications for sending or receiving data. Because of the global nature of transports, it is not necessary, or even likely, for an individual application to define a transport.

The built-in transports include printing, faxing, mailing, and beaming, but one might imagine additional transports such as messaging, scanning, compressing, archiving, or encrypting. Thus, while transports are usually associated with hardware (printers, mail servers, and scanners, for example) this is not necessarily the case (e.g., compressing, archiving, encrypting), since a service may be offered that alters the data being routed without sending it to any outside hardware.

Transports are usually built as auto-load parts; they appear in the Extensions folder of the Extras Drawer. A transport's InstallScript registers it with the system by calling the global function `RegTransport()`. As described earlier in the routing discussion, if appropriate target data is routed, the transport's name will appear in the action list in an application when the user taps the Action button.

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

Transports usually work with an application via the In/Out Box application. When an application routes data out to a transport, the transport provides a routing slip and is notified when the routed data reaches the Out Box. When items are sent, the transport will get the data from the Out Box and do whatever is necessary to send the data, setting status information at appropriate stages during the transfer.

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

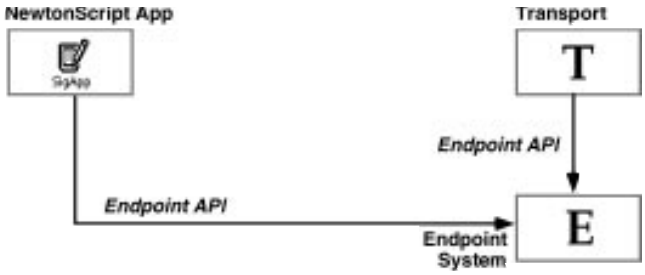
There is also an option for a transport to get information about the data being routed from the remote source and post this information into the In Box without actually getting the data. This is useful in a situation such as a mail transport where the user often wants to simply get the titles of pending messages so he or she may choose which messages they want to download to the Newton device.

The main proto used to create a transport is `protoTransport`. The powerful thing about `protoTransport` is that in many cases, surprisingly little code other than the actual endpoint code must be written. This is because the transport defaults typically "do the right thing" to provide an interface and behavior for the transport. Only those features specific to the transport (for example, archive name for an archive transport) must be added to the standard interface.

Transports which can send data also have their own routing slips based on the prototype `protoFullRouteSlip`. This allows users to provide transport-specific options such as addresses in a particular format, and so on.

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

ENDPOINTS



Endpoints are the primary NewtonScript API for programming communications on the Newton device. They provide a "virtual pipeline" for all communications. They are designed to hide most of the specifics of a particular communications media and, once connected, endpoint input and output code is usually the same regardless of the media being used.

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

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

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

Figure 6 shows the life cycle of an endpoint. An endpoint is initially defined as a frame proto'ed from `protoBasicEndpoint`. The frame has several slots describing the settings of the endpoint and methods that may be called by the system during the course of its existence. However, such a frame is not an endpoint. That is, it describes what an endpoint might look like, but it is not a NewtonScript object. To create such an object, it must first be instantiated. Note that since most objects in the Newton OS are views, and since the view system automatically instantiates a view object when it is opened, we usually

don't see this step. But since an endpoint is independent of the view system, we must explicitly instantiate it to create an endpoint object.

Life Cycle of an Endpoint

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

Note: EP is a fictitious reference to a NewtonScript endpoint frame

Figure 6. Life Cycle of an Endpoint

Once instantiated, an endpoint is opened by sending the `Open()` message to it. This ties the endpoint to a low-level communications tool in the system and spawns a new task at that lower level.

Once the endpoint is connected, it may need to be bound to a particular media-dependent address, node, and so on. An AppleTalk endpoint, for example, is bound to a node on the network. This is done by sending the endpoint the `Bind()` message. Note that some protocols (such as serial communications) do not have a required binding phase but it is still necessary to call `Bind()` (and later, `Unbind()`).

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

The endpoint method `Listen()` may be used to establish a connection instead of the `Connect()` method if the endpoint is instantiated and ready to listen to an "offer" by the remote source. Based on the particular situation with the communications media, an application may either reject the connection by sending the `Disconnect()` message to the endpoint, or accept it with the `Accept()` message. (Note that since infrared connections have one side sending and the other side receiving, in this case the passive side connects by calling `Listen()` instead of `Connect()`.)

After connecting, the endpoint is ready to send and receive data. Sending is fairly straightforward and is done by using the method `Output()`. When sending data, information about the form of the data (such as that it is a string, a NewtonScript frame, and so on) is usually sent. This gives the system a description of how the data should be formatted as it is being sent.

Output may be made either synchronously or asynchronously with asynchronous calls requiring that a callback method be specified.

Receiving data is a little more complex. Incoming data is buffered by the system below the application endpoint level. An application must set up a description of when it wants to get incoming data. This description is in the form of an `inputSpec`. For example, an `inputSpec` could be created which looked for the string "login:", or it could be set to trigger when 200 characters were received. To some extent, it can be set to notify the endpoint of incoming data after a

NTJ

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

Newton Programming: DILs Overview

Newton Developer Training

An important new addition to Newton communications programming is a set of libraries called the Desktop Integration Libraries or DILs. The first and most important thing to understand about DILs is that *they have nothing to do with programming communications on the Newton device*. Instead, they are used to create a communications link between Newton devices and *desktop* machines. In other words, they are an aid for writing code for a desktop machine which will communicate with a Newton.

Figure 1 shows this relationship. Essentially, endpoint code on the Newton, whether part of an application or in a transport, transfers data between a Newton device and a desktop machine. On the desktop machine, an application which uses a DIL sends and receives data which is handled by a desktop application. Currently DILs are available for the MacOS and for Windows-based machines.

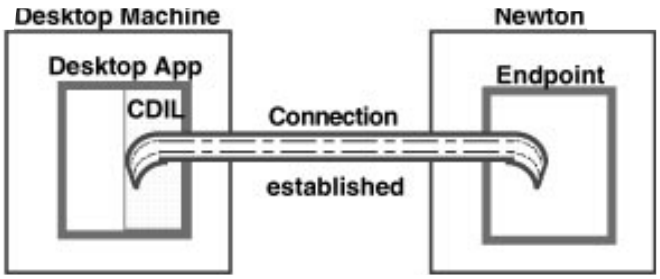


Figure 1: DILs and Newton Devices

The main advantage that DILs provide is that they make it easier to write code for a desktop machine which communicates with a Newton device. In particular, they abstract the Newton connection to a virtual pipe for bytes and provide control over such things as ASCII-to-Unicode conversions and Newton data structures and types such as frames and 30-bit integers.

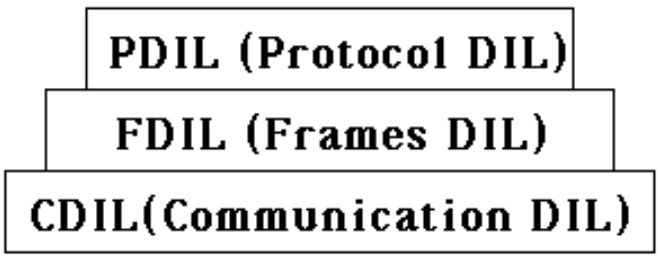


Figure 2: Hierarchical Structure of DILs

As shown in Figure 2 there are three DILs which build on one another: CDIL, FDIL and PDIL. The Communications DIL (CDIL) provides basic connectivity to a Newton device and must be used to establish a connection before you can use the FDIL and PDIL. The Frames DIL (FDIL) provides a relatively simple way to map NewtonScript frames to C structures and also provides a mechanism to handle data which was added dynamically to the frame. The Protocol DIL (PDIL) provides an easy mechanism for synchronizing data between a Newton application and a desktop application. At the time of writing, the PDIL is not yet available but will be available in the future.

All of the DILs are libraries written originally in C++ but called using a C-like syntax with a "magic cookie" object token passed into the calls. On the MacOS side, there are MPW and Metrowerks libraries. On the Windows side, DILs are implemented as DLLs and so should be independent of particular C language implementations.

CDIL

The CDIL essentially has the following phases: initialization, connecting, reading or writing, and disconnecting. This is purposefully very similar to the normal endpoint life cycle. The idea is to create and open a virtual pipe to the Newton device and then communicate using some user-defined protocol by sending and receiving messages or data down the pipe. Figure 3 shows the normal order of calls in using the CDIL.

```
CDInitCDIL()
CDSetAppplication()  // Windows only
CDCreateCDILObject()
CDPipeInit()
CDPipeListen()
CDPipeRead()/CDPipeWrite()
CDPipeDisconnet()
CDDisposeDILObject()
CDDisposeCDIL()
```

Figure 3: CDIL Calls

The `CDInitCDIL()` routine must be called before anything else can be done with the CDIL. On Windows machines the routine `CDSetAppplication()` must be called next. There is no equivalent to this call on the MacOS.

`Next`, the routine `CDCreateCDILObject()` is called to create a CDIL pipe. It returns a pointer to a pipe which must be used for all subsequent calls which involve that pipe.

`CDPipeInit()` initializes the pipe so that it is "open for business." In particular, it sets the communications options including the media details such as media type (for example, serial, AppleTalk, and so on), and relevant media options (for example, speed of connection, data bits, modem type, and so on).

Next, the pipe uses the `CDPipeListen()` call to wait for a

connection from the Newton device. When the Newton device contacts the desktop machine, the application using the CDIL may accept the connection once (`CDPipeListen()`) returns by calling `CDPipeAccept()`. This allows a connection to be canceled if, for example, the application decides that the actual connect rate was too slow. At any time in this process, the desktop application can cancel an attempted connection by calling `CDPipeAbort()`.

Once a connection is established and working, streams of bytes can be sent and received using the routines `CDPipeRead()` and `CDPipeWrite()`. As with most CDIL routines, these calls may either be made synchronously or asynchronously with a callback routine.

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

When the decision is made to terminate the connection, the routine `CDPipeDisconnect()` should be called. Once this routine has completed, connection has been broken and both sides must re-establish the connection before data can again be sent or received.

Finally, when the desktop application is completely finished with the pipe, it must call the routines `CDDisposeDILObject()` to tear down the pseudo-object and `CDDisposeCDIL()` to close the CDIL environment. On the MacOS, `CDDisposeCDIL()` closes the Communications Toolbox tool which was opened, and on a Windows machine it closes the appropriate driver.

There are several other additional CDIL calls which may be of use to the desktop programmer. These fall into four categories: encryption, utilities, status, and miscellaneous.

There are two encryption routines: `CDEncryptFunction()` and `CDDecryptFunction()`. These pass callback routines used to the CDIL. These callback routines are called by the CDIL library at the appropriate time to encrypt or decrypt data passing through the pipe. There is no attempt by the CDIL to packetize data and so, if the programmer is using a unit-oriented encryption scheme (for example, cipher block chaining), it is up to the application to buffer the incoming or outgoing data until there is enough data to encrypt or decrypt the block.

The utility routines include such things as `CDFlush()` which is used to flush the contents of the pipe in a given direction, `CDIdle()` which is used to call completion routines passed into asynchronous calls as well as checking on and updating the status of the pipe, and `CDPipeAbort()` which aborts any transactions in a given direction which are pending.

The status routines return information about the pipe. `CDBytesInPipe()` returns the number of bytes currently waiting in a given direction in a particular pipe. `CDConnectionName()` returns the name set by `CDPipeInit()`, `CDGetConfigStr()` returns the media configuration string passed into `CDPipeInit()`, and `CDGetPortStr()` which returns the name of the port the pipe is connected to (for example, "COM2" or "Modem").

`CDGetPipeState()` and `CDSetPipeState()` get and return the current state of the pipe. Figure 3 shows a list of the possible states of a pipe.

```
kCDIL_Unitialized
kCDIL_InvalidConnection
kCDIL_Startup
kCDIL_Listening
kCDIL_ConnectPending
kCDIL_Connected
kCDIL_Busy
kCDIL_Aborting
kCDIL_Disconnected
kCDIL_Userstate
```

Figure 3: CDIL Pipe States

The last of the utility class functions is `CDGetTimeout()` which returns the amount of time in milliseconds before a read or write call will time out.

Finally, the miscellaneous category includes two routines which may be useful: `CDPad()` and `CDSetPadState()`. `CDPad()` pads the write buffer so that it is an even multiple of a value passed in as a parameter. This is useful for some packetized protocols or if a unit-oriented encryption scheme is being used. `CDSetPadState()` turns the padding specified by `CDPad()` on or off.

FDIL

The FDIL (also sometimes called HLFDIL for High Level FDIL) is used to support the transfer of NewtonScript objects (frames and arrays) to the desktop. A CDIL connection must be established before FDILs can be used in order to provide the underlying pipeline for FDIL transfers.

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

The most common use of the FDIL is to map NewtonScript frames into C structures. If the frame shown in Figure 4 is going to be uploaded to a desktop machine, the desktop application can use FDILs to map this frame into the `fromNewt` C structure shown in the figure.

```
aNewtFrame:={ slot1:'b,
  slot2: {slot3:24,
          slot4: {slot5:$f,
                  slot6:$c}
                  slot7: "TROUT"};

struct fromNewt {
        char
        slot1[5];
        // symbols to str
        struct slot2 {
                long subslot3;
                        struct
                subslot4 {
                        long subsubslot5;
                        char subsubslot6;
                    }; // slot4
                }; // slot2
                char slot7[32]; // or max strlen
}
```

Figure 4: Mapping a NewtonScript Frame to a C Struct

To build the mapping between the NewtonScript frame and the C structure, an FDIL object is first created by calling the FDIL routine `FDCreateObject()`. This object acts as the central linkage for all items in the frame or array being sent or received. To map the slots in a frame or elements in an array repeated calls to `FDbindSlot()` are made. These calls match a Newton slot name (or an array object) to a C variable or buffer.

In the case shown, there would be repeated calls to `FDbindSlot()` each of which would specify a slot name of an element in the frame, the address of a memory location the slot value will be copied into, and the FDIL object which keeps all of the frame mappings. Once this binding in completed, the data can be transferred with a single call to `FDget()`.

When the Newton sends the frame data (presumably by calling `Output()`) to the desktop, the FDIL will move the data into the locations on the desktop machine specified in the bind calls. The FDIL object created keeps track of all bindings previously made so that when `FDget()` is called, the FDIL knows where each piece of incoming data should be put in the desktop machine's memory.

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

`FDget()` and `FDput()` may be called asynchronously. If an asynchronous call of these routines is made, it is important to remember that the memory to which the data is bound must still be available when the completion routine is called. In particular, memory should not be allocated using local variables since the stack of the subroutine making an asynchronous get or put call will disappear when the program exits the subroutine and, on the MacOS, it is also necessary to avoid using references to unlocked handles since heap compaction could cause memory blocks to move.

The steps for creating bound frames are shown in Figure 5.

Figure 5: FDIL Cookbook For Bound Data

1. If not previously opened, open CDIL pipeline.
2. Allocate memory on desktop for frame values.
3. Initialize FDIL.
4. Create the FDIL objects for each frame or array object.

continued from page 2

Letter From the Editor

Wow, what an introduction! It will no doubt be an enormous challenge to continue the tradition of service and technical excellence that Lee has established with the production of the *Newton Technical Journal*. So with both enthusiasm and trepidation I accept the challenge of managing and growing the *Newton Technical Journal* to be an outstanding informational and technical reference for Newton developers.

As for me, I have spent the past year in the Newton Systems Group managing Newton developer training. My current major projects are the development of an on-line communication course (excerpts of which are included in this issue), the conversion of the Newton Essentials Course to a self-paced on-line version, and the preparation of new technologies training. Prior to joining the Newton Systems Group, I was at Borland International as an engineering manager in the C++ developer support group. I am committed to providing Newton developers all the information they need to write great applications, whether it be in the form of training, technical journals, documentation, on-line information, or support.

I'd like to start by inviting you, the Newton developer, to communicate your ideas, interests, and requests via email. Several months before each issue of *Newton Technical Journal* is published, a core team of representatives from DTS, engineering, marketing, solutions relations, and training meet and discuss possible topics to

NTJ

continued from page 5

slimPicker: A Slimmer listPicker Proto

`slimPicker:AlphaCharacter(entry)`

This method is required if the AZTabs are visible.

Returns a character representing the index value for the given entry. The character will be used to set the appropriate tab in the AZTabs.

`slimPicker:CreateNewItem()`
This method is required if the New button is visible.

The method is called when the user taps the "New" button. The developer is responsible for any work that needs to be done to add the new entry. This includes creating and opening any editing or data entry slip, adding the data to the cursor, selecting the new item, and refreshing the slimPicker (see RefreshPicker below).

`slimPicker:GetHiliteShape(xcoord, bounds)`

This method is called to get the hilite box for a list item. The developer should provide this method if they wish to create a multiple column picker. This method should return something suitable for DrawShape.

`xcoord` is the current x coordinate of the pen normalized to bounds.

`bounds` is the bounding box for the item being hilited.

`slimPicker:GetNumSelected()`

This method is required if the counter is visible or `UpdateSelectedText` is called.

Returns the number of selected items.

NTJ

Correction

In *NTJ* Volume II Number 2, in the article "Apple Announces New MessagePad 130 with Newton 2.0!", there was an error in the description of the backlight API. On page 21, a function called `BackLightPresent` is documented. This function should not be used. Although it exists in the MessagePad 130, it will not be present in future hardware that has a backlight. The correct way to test for the presence of a backlight is to use the `Gestalt` function as follows:

```
// define this somewhere in your project until
// platform file defines it
constant kGestalt_BackLight :=
  '[0x02000007, [struct,boolean], 1] ;

local isBacklight := Gestalt(kGestalt_BackLight) ;

if isBacklight AND isBacklight[0] then
  // has a backlight
else
  // has not got one
```

NTJ

Apple Developer Group

Newton Developer Programs

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

Newton Associates Program

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

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

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

Savings on Hardware, Tools, and Training
- Discounts on development-related Apple hardware
- Apple Newton development tool updates
- Discounted rates on Apple's online service
- US $100 Newton development training discount

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

Annual fees are $250.

Newton Partners Program

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

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

Unlimited Expert Newton Programming-level Support
- One-to-one technical support via e-mail

Apple Newton Hardware
- Discounts on five additional Newton

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

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

AppleLink: DEVSUPPORT

Newton