Newton Technology Conference 1996
Technical Overview

This article summarizes the technical information that was presented at the Newton Technology Conference 1996. Additional information on each of these topics is provided elsewhere in this issue.

Text Engine

The Newton "edit view" view class has been part of the standard Newton user interface since Newton 1.0. It was extended in Newton 1.3 to handle deferred recognition, and then extended in Newton 2.0 OS to handle pictures. However, it is limited as simply a container for unrelated view children, and not easily extensible as a complex word processor.

Newton 2.1 OS introduces a new text-editing engine called protoTXView. This text engine allows embedded graphics within text, paragraph formatting features found in desktop word processors, and increases the maximum size of documents (stored in Virtual Binary Objects).

The new text engine allows paragraph- by-paragraph style information such as left and right indents, complex tab stops (left, right, and decimal), and lineSpacing. An optional "ruler" user interface allows modification of these settings. This paragraph-by-paragraph information is called "ruler information."

Methods are provided for getting and setting information about styles (bold, italic, underline), ruler information, and text. These APIs are based on the concepts of ranges and runs. Ranges represent places within the text, represented by character offsets, like `{first: 0, last:5}`. Runs represent style or ruler information with arrays of values representing pairs: numbers of characters and the value. For instance, `[3, fontSpec1, 5, fontSpec2]` represents eight characters of style information, with three characters of fontSpec1 and 5 characters of fontSpec2. Run arrays are used to get and set style and ruler information.

To use a basic protoTXView, just add the required view slots (viewFlags, viewJustify, and viewBounds) and use the SetStore and SetGeometry methods in your viewSetupFormScript. To save/retrieve data, you use the Internalize and Externalize messages. Note that the externalized version of the data that you can save in soups is not in a documented format.

There is an important decision to make for your own protoTXView: paged or non-paged data. Paged text indicates that there is a defined "page height" and horizontal lines appear at the end of each page to indicate the pagination. Also, if a "page break" is inserted with the InsertPageBreak message, white space will appear for the rest of that page (possibly advantageous and possibly confusing, depending on your application and your target audience).

Whether your view isPaged (the name of the argument to SetGeometry) affects other parts of your protoTXView development. For instance, functions like GetTotalHeight work differently depending on whether the view isPaged. If your view is NOT isPaged, you'll probably want to set the height of your view with SetGeometry to be arbitrarily large (see the sample for details). That will allow scrolling and clipping to work fine. In order to find the "real" height of the text (for scroll arrows or scroll bars), see the sample on how to use `GetTotalHeight` or a custom (sample code) method `GetTextHeight` to do the right thing.

There are also hooks for doing in-line pictures using graphics shapes. To do that, add a graphic of the form `{class: 'graphics, shape: aShape}` as if it were text.

If you want to do "hypertext" processing, you can use methods like viewClickScript, PointToChar, GetWordRange, GetRangeData to do it.

If you want to do "find," you have two choices: use an open protoTXView-based view, or use the Externalized data with the protoTXViewFinder object. There are methods like Find and ReplaceAll in protoTXView, and you can do simple text searching with protoTXViewFinder (useful for implementing global 'find,' since your application may not be open).

There are, however, limitations. First, graphics are always visually "in line" in the text. You cannot intelligently wrap text around large pictures. Also, no recognition capabilities are built into protoTXView. You can provide limited recognition with viewWordScript, but you will not be able to use "overwriting" or the corrector views. And finally, the data format that is exported from protoTXView for saving to a soup is in an undocumented format. You can, however, do simple text searches in saved data without opening a protoTXView -- see the documentation for protoTXViewFinder.

For more information, see the documentation and the DTS Sample "TXWord" to see how to use the APIs for font/style menus (MakeFontMenu), scrolling (Scroll, viewUpdateScrollersScript, GetScrollValues), and inserting graphics.

Grayscale Graphics Abstract

The Apple eMate™ 300 and the Apple MessagePad™ 2000 have a new, larger screen. The new screen is 320x480, one half VGA. The new screen provides 16 levels of grayscale. Shapes and text can be rendered using gray fills and gray patterns.

Color support has also been added. Colors are specified as RGB values that get converted to gray tones when rendered. Pixel maps can be created using multiple bit depths. Each depth can specify its own color map. The color map will be converted to grays when the pixel values are displayed.

PICT 2 is supported now as well. Most opcodes for PICT 2 as defined in Inside Macintosh are supported with the exception of a few drawing modes related to color. NTK 1.6.4 will provide support for directly importing color PICTs into NTK projects.

Shape handling has been improved, with more flexible manipulations now supported. Selection handles can be displayed at the bounding corners of a shape, and FindShape will consider the selection handles when determining which shape has been hit.

Bitmaps can be resized and anti-aliased. The anti-aliasing results in a pixel map that uses gray values.

The User Interface protos have not been changed for gray scale with a single exception: protoFloatNGo now uses a gray outline. Any proto that supports displaying of text or shapes can display gray images.

Keyboard

Extensive enhancements have been made to improve support for keyboards in Newton 2.1 OS. These enhancements include the following:

- New APIs to capture keyboard input
- Notification of key view focus changes
- Support for key commands
- Command key support in pickers
- Default buttons and close boxes

Newton 2.0 OS laid the groundwork for keyboard support and included a limited API to intercept keyboard input. Newton 2.1 OS extends this API so that any type of view can accept keyboard input. The OS also provides new methods to intercept key down, key up, and key repeat events. In addition, views are now notified of when they gain key view focus and when they lose key view focus. The Newton user interface has also been extended to include visual cues that a view is the key view.

One major addition to the OS is an API to handle key command combinations. You can now provide an event handler which will be called when a particular key combination has been pressed. Key commands are contextual which means that you can have one view in your application which has a different set of key commands than another view. For application-wide key commands you could register the commands with your base view. You can additionally have system-wide key commands by registering key commands with the root view.

A help slip is available which will give you a list of key command possibilities for the currently active key view. This help slip is always available by pressing and holding the command key for approximately two seconds.

There is also support for showing key equivalents in your application's pickers. When the user has a picker open and types the key combination, you can specify whether the OS will directly call your key command event handler or whether the picker's pickActionScript will be called. The OS handles displaying what the key combination is for each item in the picker.

Lastly, you can create a default button which will respond to the return key. This button will be drawn with a darker border so the user knows it is a default button. Additionally, you can specify that a closebox respond to command-w or command-period. A closebox could be either a button or the standard Newton closebox.

Sound

Newton 2.1 OS adds the ability to record sound, as well as providing more support for playing back sounds. There are UI elements which make implementing sound capture very easy, as well as the providing the ability to record sounds with control over many options.

The MessagePad 2000 has an internal microphone, whereas the eMate 300 has a headphone jack. Both devices support sound input and output on the interconnect plug.

There are two built-in user interfaces for recording and playing sound. One is designed to be embedded inside your application's views, and the other is in a "floating" slip with its own close box. Both provide standard record, stop and play buttons, with an indicator to show the length of the sound.

If more control over the recording process is needed, your application can use the methods of protoSoundChannel. To record, first you initialize the sound channel and set its parameters, then you queue one or more sound buffers, then start recording. You can queue more sound buffers during the recording session. Eventually you stop the process and clean up the objects.

There are a number of sound compressors and decompressors you can use when playing and recording sounds: MuLaw, IMA and GSM. An FM synthesizer supporting up to four sine waves is a built-in "decompressor," and a Macintalk speech synthesizer, while not built in to the ROM, does exist as a package.

Newton Works

Works is a new document framework which comes pre-installed on the eMate 300 and will be included on a disk for the MessagePad 2000. This framework is intended to provide a simple yet powerful shell for productivity applications, much like the desktop products with similar names.

Four applications will be initially available in the Works framework: a word processor, a drawing tool, a spreadsheet, and a calculator with graphing capabilities.

Works uses a document metaphor that clearly separates individual projects. Unlike desktop productivity applications, Works on the Newton platform is extensible through the stationery mechanism. To add a new type of document, simply register a view definition and a data definition using the existing stationery API. The superSymbol slot of your data definition should be `'newtWorks`. New stationery should try as much as possible to maintain easy-to-understand interfaces and use established patterns for user interfaces such as command key equivalents and menus.

Works defines a variety of required and optional APIs that you add to your stationery's viewDef and dataDef. These APIs cover preferences, scrolling, finding, registering tools, providing help, and manipulating the status bar.

For more information on using Works, see the Works article and the DTS sample code about Works.

Multi-User/Simple Mode Abstract

The Apple eMate™ 300 is intended for an environment where several students may be sharing a single device. A teacher can easily set up an eMate™ for the students in a class. Simple mode also allows for a simplified user experience with the eMate™.

The teacher may also want to limit the availability of applications in the Extras Drawer. When setting up the eMate™ 300, the teacher can select which applications will appear in the Extras Drawer and which will not. For example, the teacher may decide that having access to the Calculator would not be a good idea during an arithmetic test, so the Calculator can be removed from the Simple Mode Extras Drawer.

Each student’s data is kept separate. Applications that are eMate™ 300-aware can determine who the current user is and use that to filter the information available to the user. Apple strongly recommends that each user have a separate soup for storing information. When in Multi-User mode, the eMate backs up data specific to the user, helping to keep different users’ information separate.

New students can be easily added to the unit. Creating a new account is simply a matter of assigning a name and a password. At the teacher’s option, a password need not be required. Optionally, students can add themselves to the unit when they first sign on.

Classroom Connection
When students want to import and export data from their Newton eMate 300s, they will connect over AppleTalk or a serial cable to the "Classroom Connection" server. This server makes backups of the user's data (both NewtonWorks data and third-party application soups) and allows selective import of the backed up files.

For data stored by NewtonWorks stationery, a single file is created on the desktop for each soup entry, containing the soup entry frame as well as a small amount of header information. Non-NewtonWorks packages have their entire soup or soups backed up to a single file.

These desktop files are readable by desktop applications which will probably use the DILs to convert to and from the Newton frame format. If you have control over the desktop application, you could have it read and write the Newton format automatically.

Alternatively, writing an XTND translator or a drag-and-drop converter application will be a good way of converting the data from the Newton format into one suitable for an existing desktop application. XTND translators are being developed to allow applications to read and write the NewtonWorks drawing and word processing documents.

In order to provide custom icons, creator codes and filetypes for the desktop files, the Classroom Connection server will work with developer-created extension files. The provided icons and other resources will be matched with the Newton data based either on the class of the NewtonWorks soup entry, or (for non-NewtonWorks data) the soup name.

DILs
The DILs (Desktop Integration Libraries) are free libraries that you use in your desktop applications (MacOS and Windows) to communicate with a Newton device. The DILs make it much easier both to move the data as well as translate from the Newton format into a non-NewtonScript application.

There are three varieties of DILs: the CDIL (for basic communications), the FDIL (for interpreting frames and other NewtonScript objects) and the PDIL (to speak the Connection application's protocol, which enables AutoDocking). The CDIL and FDIL were released in January 1996; the PDIL will be in beta form soon. An update to the Windows DILs was released in November.

The CDIL provides you with a cross-platform, media-independent API for a virtual pipe through which you send and receive data. It optionally performs byte-swapping and Unicode conversion for you, and is the core upon which the FDIL and PDIL are built. The CDIL API is at a higher level than that of raditional desktop communications APIs, and insulate the developer from the complexity of communications protocols.

The FDIL provides a binding API to translate NewtonScript frames, arrays and their contents into C-style structures or other data buffers used in desktop applications. Since frames are highly flexible and dynamic objects, the application provides a template to extract their data into the C structures, and then can access the "unbound" or freeform data as necessary.

The PDIL provides higher-level access to the Newton device, and since it communicates with the built-in Connection application (now renamed Dock), writing NewtonScript code is no longer necessary to synchronize with or transfer data to and from a Newton device.

The PDIL libraries are what enable AutoDocking on new devices, and also can communicate with Newton 2.0 devices. PDIL functions provide full access to Newton soups (including queries), let developers download code and packages, and allow the desktop to remotely call functions and methods on the Newton device.

IrDA

IrDA (Infrared Data Association) is a consortium founded in 1993 as a non-profit organization. Its goal is to create and promote interoperable, low cost infrared data interconnection standards that support a walk-up, point-to-point user model. The standards support a broad range of appliances, computing, and communications devices.

IrDA has defined a set of industry standards which provide robust, low cost, low power, high performance point-to-point infrared communications. Newton OS 2.1 adds support for three layers of the IrDA specifications. Specifically, these are the asynchronous serial ("SIR" or "physical") communications layer, the link access protocol ("LAP") layer, and the link management protocol ("LMP") layer.

The SIR layer is implemented on the Newton using a UART and is capable of speeds from 2400 bps to 115.2 Kbps. The LAP layer provides the error correction and reliable datagram delivery mechanism. The LMP layer implements a simple name server and link multiplexer and is the layer upon which endpoints are based.

IrDA communications are physically half-duplex, yet the protocol "simulates" a full duplex serial communications link. Applications which work over a standard asynchronous serial connection can be quickly and easily modified to use IrDA.

IrDA is a networking protocol of sorts. Devices have types and names, and connection parameters such as speed and window size are negotiated between devices automatically. The standard is continually evolving and improving.

IrDA is not compatible with the IR beaming protocol used in the MessagePad 130 or any of the previous MessagePad models. In order to provide backward compatibility in beaming, the beaming transport in the Newton OS has been modified to detect which protocol is in use by the sender/receiver, and to use either Sharp ASK, Newton IR, or IrDA as appropriate.

Full IrDA specifications are available on the web at: http://www.irda.org/

Mail Enabler

With the release of Newton 2.0, developers were pleased to find supported APIs for developing transports that can be integrated into built-in and third-party applications. However, these APIs are not specifically directed at mail-specific transports. Developers of text and text-and-attachment transports still must write code and design dialogs to achieve a consistent user interface with standard transports like the Compuserve© client.

For Newton OS 2.1, Apple designed the Mail Enabler protos. Apple will release this suite of protos in a future NTK "platform file" for use with third-party transports. They are NOT in the ROM, even though they are part of the Newton 2.1 project, and will be released when the Newton 2.1 ROM is finalized. They will enable developers to simplify the development process for mail transports for both Newton 2.0 and Newton 2.1.

The features include mail-like routing slips, connection slips, automatic text export, mail header, text dataDefs and stationery, and automatic toggling between text and attachments for transports that support both dataTypes.

To use the Mail Enabler protos, you still must write your own communications code. However, much of the user interface code and transport code is simpler. As before the Mail Protos, most of your transport development time will probably be specific to your endpoint and protocol code.

Who should use Mail Enabler? Any text-only or text-and-frame (attachment) transport that has a mail-like routing slip. There are many optional features, like cc/bcc, attachment handling (the 'frame dataType), and optional preference items you can add to your preference slips for changing the mail font, and so on.

The mail transport proto also handles automatic export to text appropriately, and knows how to switch between viewing "text" and viewing the "frame" within the Out box (via a GetTransportScripts menu item).

For routing slips, you get to/cc/bcc pickers for free, as well as a "show message" text viewer. You can add additional views to the routing slip.

To use the Mail Enabler connection slip, you must provide a method to return an array of phone numbers and baud information, given the current worksite and city information (and also a view to open if the user selects "Other phone number" from a phone number picker)

If you are preparing to write a mail transport now, learning about transports and communications endpoints before starting development is recommended. See the DTS transport samples "MinMail" and "ArchiveTransport" for examples of writing text mail transports *without* Mail Enabler. Also, see the NTJ article "FSM Article," and the samples "Altered States" and "CommsFSM."

Newton Internet Enabler

NIE (Newton Internet Enabler) is a software package which extends the communications capabilities of the Newton OS. Specifically, NIE provides TCP and UDP endpoints over PPP and SLIP physical links. For security, version 1.1 adds PAP, CHAP, and Interactive Authentication support. NIE also provides a non-recursive client interface to a DNS, as well as automatic ICMP responses.

NIE is composed of three main components: Link Controller, Domain Name Resolver, and Internet Communications Tool.

The Link Controller is responsible for managing the physical connection to an internet service provider. Multiple applications can share the same physical link to the service provider. Only one physical link may be active at a time.

Using the Internet Setup application, the user defines the IP information necessary for communication (for example, the protocol to use, the local IP address, the IP address of a domain name server, the phone number, and so on) as well as the login script the link controller uses to establish the connection. Once defined, the user simply refers to this connection information by name. Multiple connection setups may be stored, and a particular setup can be recalled when establishing the connection.

The Domain Name Resolver is a simple non-recursive "stub" interface to a Domain Name Server. The DNR can resolve host names to IP addresses and IP addresses to fully qualified domain names for both standard and mail server name entries. The DNR requires authoritative responses from the DNS.

The Internet Communications Tool is a standard communications service which implements UDP and TCP endpoints. Note that although UDP is not a connection-based or stream-oriented protocol, the endpoint API is used with packetization flags to send and receive individual UDP datagrams. For TCP, the full protoBasicEndpoint API (sans end-of-packet flags) for reliable stream-based connections is supported. Multiple endpoints may be simultaneously active and sharing the same physical connection to the service provider.

Programming for NIE is essentially no different than for any other protoBasicEndpoint service. Only a few additional steps are required, specifically, choosing a physical connection setup and establishing the physical link, and tearing down the physical link when finished. For as long as the physical link is established, the application simply uses endpoints like any other communications application on the Newton.

For more information, please refer to the NIE web page:

http://devworld.apple.com/dev/newton/tools/nie.html

Building A Better State Machine

What is a Finite State Machine or FSM? Quite simply, it's an ordered system of states, state transitions, input events, and output results. An FSM is an implementation of a logical model. Conversely, state diagrams, state tables, and state outlines are logical models of the implementation.

A state diagram (sort of a cross between a Venn diagram and a Karnaugh map) is usually the first step in defining an FSM. From here, a state table or state outline is usually generated. Creating a state diagram is an inductive process, utilizing the power of the human mind to garner the often abstract relationships between components of the system. FSMs are not a panacea of program development and problem solving, but they are extremely useful tools in many kinds of software development.

The use of Finite State Machines (FSMs) is but one method of defining a sequential process, finding its weaknesses, and preparing for its implementation in an obvious, deterministic fashion. FSMs can be applied to just about any sequential process, from language parsing to process control. And for computer communications applications in particular, they are often a very powerful tool.

There are essentially two "types" of Finite State Machine: Deterministic and Non-deterministic. In a deterministic FSM, for any given input event there is exactly one unique state transition. In a non-deterministic FSM, for any given input event there are multiple possible state transitions. A non-deterministic FSM may become deterministic in some future state, but the goal is generally to make the FSM as deterministic as possible, as early as possible.

Many issues arise when dealing with FSMs. One of the most common is the problem of synchronization; organizing the parallel activities of two or more disparate FSMs. Among the many ways to enforce synchronization behavior is by the use of "Do" and "Done" events, and "Wait" states. A "SyncFSM" coordinator is used as a higher-level interface.

Another, perhaps more obvious, issue is the fact that the number of states in an FSM can easily "explode" as new required state transitions are added to the system. This "factorial explosion" behavior is often addressed through the use of state reduction techniques. In essence, flow-control intelligence is added to the system in order to reduce the sheer number of states and volume of code required for implementation. Incorrect state reduction can quickly lead to unstable or non-deterministic finite state machines, however, so the developer is cautioned to proceed in this area with great care.

There are other issues to be addressed when dealing with finite state machines, far too many to list here. For more information and some very useful examples of FSM implementations, please refer to the Newton DTS sample code projects "protoFSM", "Altered States", "ArchiveTransport", "Comms FSM", "Mini-MetaData", and "Thumb," to name a few.