Newton® Technology
JOURNAL

Volume I, Number 3 June 1995

Inside This Issue

Newton®

Newton Technology At Work

Making the ACT! Connection

by Jeff Cable, Symantec Corporation

For each person on a software design team, there will be a long list of potential features to include, from which only a handful make it into the final product. When we started designing ACT! for Newton, we contemplated all the feature requests and tried to determine which combination would yield the best final product. Of course there were the obvious features: contact list, attached notes and histories, and task list. But when it came to Newton-specific features, we all had different ideas. There was one feature of ACT! for Newton, however, upon which we all agreed, and that was connectivity.

It was clear that without a solid connection to both Macintosh and Windows machines, ACT! for Newton would not offer the much-needed solution on the Newton platform. As good as the stand-alone Newton application might be, it would remain a kind of island without some connectivity.

Luckily, at the same time we at Symantec were designing our product, the Newton Group at Apple Computer was writing the Desktop Integration Libraries (DILs). This made for an obvious choice – use the DILs to both simplify and streamline data synchronization between ACT! for the Newton and ACT! for the desktop machines.

From a marketing perspective, the DILs helped save time and money, by providing the desktop connectivity element required in our marketing specification. From a development perspective, getting data moved quickly was very

continued on page 8

New Technology

Welcome to the Desktop Integration Libraries

by J. Christopher Bell, Apple Computer, Inc.

Ever since its release in August 1993, Newton has been hailed as an ideal platform for gathering information "at the source." Client-server solutions, consumer applications, and information distribution systems all require information-gathering or browsing directly on a mobile device, rather than waiting until a user returns to a local network or desktop.

For many end-user applications, the Newton Connection Kit offered a great solution since it combined backup/restore functions and third-party import/export functions, while allowing the user to edit much of their Newton data on a Windows or MacOS desktop computer. All of this was achieved without developers directly addressing communications or NewtonScript frame translation. The Newton Connection Kit (NCK) handled synchronization and communications directly, and the built-in NewtonScript interpreter made manipulation of user data easy within the context of "meta-data" code executed within NCK.

However, many users asked for a direct link between their Newtons and their favorite desktop calendar, or expense programs. They were frustrated with the extra necessary step of Newton Connection Kit synchronization.

The desired direct link is possible using the Newton's application programming interface

continued on page 22

Newton Technology JOURNAL

Volume I, Number 3 *June 1995*

Published by Apple Computer, Inc.

Lee DePalma Dorsey • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

David Glickman • *Coordinating Editor, Business Content*

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

Philip Ivanier • *Manager, Newton Developer Relations*

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

Contributors
J. Christopher Bell, Christopher Bey, Jeff Cable, Rob Langhorne, Susan Schuman, Bill Worzel

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

John Kawakami • *Editorial Assistant*

Judith Chaplin • *Art Director*

Editor's Note

by Lee DePalma Dorsey, Managing Editor

DILs Make Newton Connectivity Seamless

Newton PDA devices have brought mobile professionals the ability to organize information and communicate from a small form-factor device away from their offices and telephone lines. The Newton Connection Kit has also enabled users to download, synchronize and back-up data between their Newton PDA devices and their Windows or Macintosh computers, making desktop connection one of Newton's most important features for mobile users.

While the Newton Connection Kit has enabled the essential connection to the desktop and allowed non-PIM users easy synchronization and a simple PIM of sorts, desktop application users have been required to take an extra step with the software. Often, the data does not match up one-for-one with their favorite PIM, contact management application, or database. Conversely, applications developers have not been able to easily move data from an existing desktop application to a built-in or third-party Newton application. While some Newton developers were able to write code that enabled a direct connection built into the application, it was not an easy task and the results were slow and sometimes inaccurate synchronization. Users have been asking for an easier way and developers have asked for the tools to make one-to-one seamless connection easier.

We are pleased to be able to announce that the Desktop Integration Libraries (DILs) will solve this problem for both users and developers. As a customizable "black box," the Desktop Integration Libraries are the glue to bind Windows and Macintosh applications to Newton devices. Using the appropriate DIL libraries, desktop developers can enable their Windows and Macintosh applications to directly connect with a Newton device, exchange data and synch up with appropriate Newton applications. Similarly, Newton application developers can use these libraries to ensure more effective Macintosh/Windows connectivity for their applications, which can be developed in shorter time periods and resulting in faster time to market. Using the libraries, Newton and Desktop developers will be able to deliver more effective mobile connectivity solutions for both horizontal and vertical markets, opening up an entire new marketplace for existing desktop applications. While the Newton Connection Kit will still be essential for certain user functions, such as backup, restore, and general access to desktop data, applications useing the DILs will give users the seamless connection they've been looking for.

The feature articles in this edition of the Newton Technology Journal will provide you with a technical view into how the DILs work, and Jeff Cable of Symantec Corporation provides you with a view of the developer who has actually already produced a Newton application from their desktop contact management application, ACT. We encourage you to think about how you might take advantage of the DILs to add value to your existing desktop or Newton applications and help yourself build a new business or market segment in the process.

These Desktop Integration Libraries will soon be bundled with the Newton Toolkit, and made available to Newton developers at no extra charge. Watch the Newton Technology Journal or your Newton Developer Monthly Mailing for more details. Developers interested in more information on the DILs can send questions to directconnect@newton.apple.com. We know the delivery of the libraries will significantly enhance your ability to create powerful Newton applications that deliver the solutions Newton device users want.

Global Systems for Mobile Communications (GSM)

by Susan Schuman, Schuman Consulting, sschuman1@aol.com

Ever since the first European cellular communications services appeared in 1982, cellular technology has been central to the development of the mobile communications market. The concept of users being able to make or receive a phone call while away from a fixed location revolutionized the industry and a mobile communications market began to emerge. Wireless data meant the possibility of instant access to information and communication services anytime, anywhere.

If the introduction of analog cellular represented a revolution in mobile communications, the development of European digital cellular technology – the Global Systems for Mobile Communications (GSM) – represents the next crucial step for the industry. This technology will allow mobile communications to break out of the professional and business niches and to move into broader markets. The GSM standard and service is the first truly pan-European network providing digital technology advantages and a system that allows full inter-working between countries. But before explaining GSM further, let's first look at the basics of analog cellular communication, since it provides the foundation for transition to digital cellular networks.

Today, the most widespread option available for transmitting and receiving data wirelessly is called circuit-switched analog cellular. Basically the cellular network works like the standard telephone network. Circuit-switched cellular users send data to a base station, where it is routed over the public telephone network to its destination. From the perspective of the cellular phone system, a circuit-switched data call is the same as a cellular voice call; there is no operational difference in how the two kinds of calls are handled. All the principles and procedures of the analog cellular networks still apply, and these include roaming, hand-off and routing of calls through the cellular provider's central office.

The important point here is that data is being sent over a voice network. The analog cellular network was not designed for data calls; the priority has always been voice calls. The infrastructure, the technology of the switches, and the tariff structures were all designed for voice service. The cost of sending data is calculated on a per-minute basis, just as it is for voice. Therefore sending a short burst of data (which make up most of today's data transactions), the user pays for a full minute that is not being used completely. Sending data over analog cellular can be costly.

The term "circuit-switched" refers to the establishment of a dedicated connection or circuit between two end points (modem to modem). Because the connection remains fixed for the duration of the call, data sent this way is often referred to as a connection-oriented transmission. Once the modem connection is established, the channel is dedicated to the session until one caller terminates the session.

Currently, to send data over a cellular network requires a device such as Newton-based product (PDA) connected to a cellular phone via a cellular capable modem (such as a PCMCIA cellular modem card plugged into the PDA). This type of connection and the current tariff structure make the network suitable for larger file transfers and for session-based dial-up applications such as faxing and vertical applications, as well as for store-and-forward applications, such as e-mail. In other words, whatever a user could do with a standard landline modem, they can now do wirelessly.

The analog cellular technology and networks served well as a first generation wireless technology. However, analog services are now straining to keep up with user demand. Analog transmissions are less efficient than digital transmissions in terms of spectrum utilization, and in areas with large numbers of wireless users, analog networks can't always meet demand. Because these networks were developed for voice, connections can be jeopardized by intracell hand-offs, fading and interference from other radio-frequency signals. These mean that the modem may have trouble establishing or even keeping a connection, and that most connections can not accommodate speeds higher then 4,800 bps. Considerations like these make analog networks less than ideal solutions for sending data, especially if the device is in motion. In Europe and Asia, where international travel is often required for the mobile market, roaming across borders is not possible with analog services, except where neighboring countries have coordinated transmission frequencies and standards.

To overcome these problems, efforts are now focused on upgrading the current analog cellular networks to digital. In the US there are two competing standards: Code-Division Multiple Access (CDMA) and Time-Division Multiple Access (DMA). The Japanese have chosen their own standard, Japan Digital Cellular (JDC). However, Europe and most other countries in the Asia/Pacific region have moved forward, rapidly implementing the GSM digital cellular standard and making it the most widely accepted digital cellular standard throughout most of the world.

GSM was developed in Europe and digital cellular services began there in July of 1992. The first European GSM network was opened by the Finnish operator Radiolinja, and since that time almost all of the licensed operators across Europe have begun service. GSM has been successful in Europe because of its international roaming agreements and consistent implementation. With the transmission being purely digital, the networks benefit from improved security of communication, and full data and two-way messaging capabilities. In the Asia/Pacific region, most countries are beginning testing and implementation the European GSM standard.

Once again it is important to note that digital cellular standards were designed primarily for voice. Much of the movement to digital cellular has been to counteract the problem of limited capacity on existing analog cellular networks. However, unlike analog cellular, GSM specifications do include provisions for data transmission. These include circuit-switching data services at rates up to 9,600 bps and Short Message Service (SMS), allowing a GSM device to function as a two-way messaging (paging) device. Specifications for packet-switching capability are also in the GSM definition.

At this time, GSM voice services have been implemented in most of Europe and in many parts of Asia/Pacific, but the data, fax and SMS services are still being implemented in a process that will continue for the next few years.

One of the benefits of analog cellular networks was that they bridged to the PSTN (Public Switched Telephone Network). Because GSM is based on analog cellular networks, it also provides bridges to the PSTN, allowing a GSM device to "talk" to a regular modem or facsimile machine, to access a corporate network remotely and to access public information services. Cellular networks cover more of the world then any other type of wireless network. This wide coverage, the bridge to the PSTN, the large installed base of cellular voice users, the international adoption of GSM and the fast growing mobile market, position GSM as the dominant technology for wireless data service in Europe and the Asia/Pacific region.

The GSM specification provides a digital, spectrum-efficient, cellular architecture. GSM has clearly defined voice and data channels, including a side channel for short, 2-way messaging. GSM operates in the 900 Mhz frequency band which, compared to analog standards ,

- offers better signal quality, which translates to fewer transmission errors
- offers better security via encryption and encoding
- offers more efficient use of spectrum, which means higher network capacity.

GSM is also compatible with ISDN (Integrated Digital Services Network – the digital standard for the telephone network), and in fact some people would describe GSM as the wireless ISDN implementation. Digital cellular is a generally more robust technology because it addresses the issues of noise, interference, unreliability and poor performance.

Recognizing the importance of GSM and its widespread acceptance, Apple has put considerable effort into developing software for Newton-based products that allow users to connect to the GSM networks via the Nokia 2110 and 2140 handsets and the Nokia PCMCIA Cellular Data Card. With this combination, Newton-based GSM-capable products can wirelessly transmit faxes and can communicate from remote locations at any time. In addition, existing third-party applications can take advantage of the GSM connectivity – any application that is based on a dial-up connection to transmit data can now work seamlessly within the GSM environment. Third-party developers can now write applications to take advantage of this new connectivity and can provide a wireless element to their solutions.

Wireless communication for Newton-based products is clearly where the PDA market is heading. This is an exciting moment in the GSM world, because:

- the GSM standard has widespread acceptance
- implementation of the data portions of the GSM network has now begun
- software and hardware solutions are already available for Newton-based PDAs

This combination of advances begins to meet the needs of businesses and individuals who want instant access to information and communication services anytime, anywhere. It allows them to be truly mobile, while staying NTJ

Watch for continued coverage
on wireless products and technology
in future issues of the
Newton Technology Journal.
Next issue:
The Wayfarer Enterprise Server.

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

Modem Setup Packages

Until recently, the only modems that could be used with Newtons were those sold and supported by Apple. This has now changed. The modem setup capability described in this article provides an easy-to-implement mechanism that allows many different kinds of modems to be used with Newton.

The User's View

When a developer- or manufacturer-supplied Newton modem setup package has been installed on a Newton, a user can simply choose the desired modem setup in the Modem Preferences view, as shown in Figure 1. The Modem Setup item in this view is a picker ("Connect with…"), which when tapped, displays all of the modem setups that are currently installed in the system. The chosen modem setup is subsequently used by all applications.

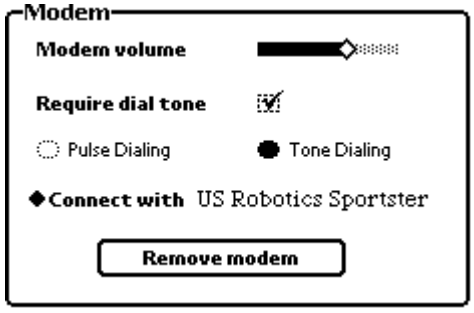


Figure 1. Modem Preferences View

What is a Modem Setup Package?

A modem setup package is installed on the Newton as an auto-load package. This means that when the package is loaded, the modem setup information is automatically stored in the system soup and then the package is removed. No icon appears for the modem setup in the Extras Drawer. Instead, modem setups are accessed through a picker in the Modem Preferences view.

Modem setup packages can be supplied by modem manufacturers, or can be created by other developers.

A modem setup package can contain two parts:

- A modem tool preferences option, to configure the modem controller
- A modem tool profile option, to describe the modem's operating characteristics

How the Modem Setup Service Works

All Newton communication applications that use a modem endpoint make use of the modem setup service. When a modem endpoint `Instantiate` call is made, but before the `Bind` and `Connect` are done, the current modem setup is invoked. (For a description of *endpoints*, *comm tools*, and other related details mentioned in this article, refer to the "Newton Communications" article by Bill Worzel in Volume 1, Number 2 of the Newton Technology Journal.)

Note: If the modem endpoint option list includes the modem profile option (kCMOModemProfile), the modem setup is not invoked. This allows modem applications to override the modem setup when configuring the modem for special purposes.

When the modem setup is invoked, the `Instantiate` method sets the modem preferences (`kCMOModemPrefs`) and modem profile (`kCMOModemProfile`) options as defined in the modem setup.

Defining a Modem Setup

The various parts of a modem setup are specified in an NTK text file. The modem preferences and profile options are specified simply by setting constants. The setup methods are specified as functions. The following subsections describe each part of the modem setup.

General Setup Information

The beginning of a modem setup contains general information about the setup and the modem to which it corresponds. Here is an example:

```
constant kModemName:= "US Robotics Sportster";
constant kVersion:= 1;
constant kOrganization:= "Apple Computer, Inc.";
```

Table-1 describes these constants

Table 1. Modem Setup General Information Constants

Constant	Description
kModemName	A string that is the name that shows up in the Modem Preferences picker to identify this modem setup. Typically this is the name of the modem.
kVersion	An integer identifying the version number of this modem setup package. The system prevents a modem setup package with an equivalent or lower version number from overwriting one with a higher version number that is already installed on a Newton.
kOrganization	A string identifying the provider of the modem setup package.

Modem Preferences Option

This modem option configures the modem controller and specifies such things as how the modem should determine if it is receiving the Carrier Detect (CD) signal, whether the modem requires a configuration and dialing option string, and what to do on disconnect. Here is an example:

```
constant kidModem :=  nil;
constant kuseHardwareCD :=  true;
constant kuseConfigString :=  true;
constant kuseDialOptions :=  true;
```

```
constant khangUpAtDisconnect :=  true;
```

Table 2 describes the modem setup preference option constants.

Note: Where the backslash (\) is used in a configuration string, you must specify two of them together (\\), since a single backslash is used as the escape character in NewtonScript.

TABLE 2:. MODEM SETUP PREFERENCES OPTIONS

Constant	Description
kidModem	Should be set to nil to prevent the modem tool from executing a modem ID sequence and automatically setting the modem profile.
kuseConfigString	Set this to true, unless the modem happens to be configured exactly correctly when it is reset, which is very unlikely. A setting of true means that a modem configuration string is to be sent to the modem before initiating a connection. The modem configuration string is defined in the modem profile option and depends on the connection type. If nil, no modem configuration string is sent.
kuseDialOptions	Set this to true to send the default dialing configuration string to the modem, following the configuration string. The default dialing configuration string is: ATM1L2X4S7=060S8=001S6=003\n. If you specify nil, the dialing configuration string is not sent to the modem.
khangUpAtDisconnect	Set this to true. This setting causes a "clean" hang-up sequence to occur when the modem disconnects. If nil, no hang-up commands are sent to the modem on disconnect.

Modem Profile Option

This modem option describes the modem characteristics, to be used by the modem controller. Here is an example: of the profile option constants for a Hayes error correcting modem:

```
constant ksupportsCellular := nil;
constant ksupportsEC := true;
constant ksupportsLCS:= nil;
constant kdirectConnectOnly := nil;
constant kconnectSpeeds:= '[300, 1200, 2400, 4800, 7200,
  9600, 12000, 14400];
constant kconfigSpeed:= 19200;
constant kcommandTimeout:= 2000;
constant kmaxCharsPerLine:= 40;
constant kinterCmdDelay:= 50;
constant kmodemIDString := "unknown";
constant kconfigStrNoEC:= "ATE0&C1S12=12W2&K3&Q6\n";
constant kconfigStrECOnly:= "ATE0&C1S12=12W2&K3&Q5S36=4\n";
constant kconfigStrECAndFallback :=
"ATE0&C1S12=12W2&K3&Q5S36=7\n";
constant kconfigStrCellular:= nil;
constant kconfigStrDirectConnect :=
"ATE0&C1S12=12W2&K0&Q0\n";
```

Table 3 describes the profile constants.

TABLE 3. MODEM SETUP PROFILE CONSTANTS

Constant	Description
kidModem	Should be set to nil to prevent the modem tool from executing a modem ID sequence and automatically setting the modem profile.
ksupportsCellular	Indicates if the modem supports cellular data connections. If true, the configuration string defined by kConfigStrCellular is used for cellular connections. If nil, the normal data mode configuration string is used for cellular connections.
ksupportsEC	Specify true if the modem supports any error correction protocols such as V.42 or MNP, and the profile contains configuration strings for error correction. Note that kdirectConnectOnly must also be nil. Specify nil if the modem does not support error correction.
ksupportsLCS	Specify true if the modem supports LCS (Line Current Sense), or nil otherwise. LCS is used for determining when a user has lifted the phone handset off hook. Applications can take advantage of this feature by allowing the modem to determine when it should release the line for a voice call.
kdirectConnectOnly	Normally this is set to nil. Set to true if the modem does not support error correction or speed buffering.
connectSpeeds	An array indicating the speeds (in bps) at which the modem can connect. This array is not used, except by application programs that want to use it to determine the modem capabilities.
kconfigSpeed	Indicates the speed (in bps) at which to configure the serial hardware communicating with the modem. You must specify a speed greater than the fastest connection speed supported by the modem. Must be set to at least 19200 for fax.
kcommandTimeout	Indicates how long (in milliseconds) the modem tool should wait for a modem response to a command before timing out. A setting of 2000 ms. is usually sufficient, though some modems may require 3000 or 4000 ms.
kmaxCharsPerLine	Indicates the maximum number of command line characters that the modem can accept, not counting the AT prefix and the ending carriage return.
kkinterCmdDelay	Indicates the minimum amount of delay required between modem commands, in milliseconds. This is the time from the last response received to the next command sent. A setting of 25 ms. is usually sufficient, (longer delays are not recommended for fax operation.)
kmodemIDString	Normally set this to the string "unknown". This string is used if the modem tool attempts to identify the modem using the ATI4 command. It should be set to the same string with which the modem responds.
kconfigStrNoEC	The configuration string used for non-error corrected data connections when kdirectConnectOnly is true, and for FAX connections. This configuration string must enable speed buffering. The default string is as follows: E0 Echo off (always required.) &C1 DCD indicates the true state of the remote carrier. S12=12 Escape guard time is 240 ms. (12*20). Modems usually set S12=50. W2 Report connection in "CONNECT bps" format. Not all modems accept this command. An alternative is to use Q0 with X1 or X4, and V1. &K3 Enables bi-directional RTS/CTS flow control. The modem uses CTS to control flow from the Newton, and the Newton uses RTS to control flow from the modem. This does not work on all modems. An alternate form is \Q3\X0. It is possible that &R0 and \D1 will be required as well. &Q6 Use normal buffered mode. Again, this does not work on all modems. An alternate form is to use \N0, or on some modems \N7. Without hardware flow control (kdirectConnectOnly is true), software flow control should be used for FAX connections. In this case, instead of &K3, use the following commands: &K4 Enables bi-directional XON/XOFF flow control. The modem and Newton halt data flow when they receive XOFF (DC3) and resume data flow when they receive XON (DC1). This does not

Constant		Description
	&R1	Assume RTS is always asserted. This does not work on all modems.
	\D0	Force CTS on at all times. This does not work on all modems.
kconfigStrECOnly		The configuration string used for data connections that require error correction. This configuration string must enable speed buffering and can be used only if hardware flow control can be enabled. The default string is nil. Here is an example:
	E0	Echo off (always required.)
	&C1	DCD indicates the true state of the remote carrier.
	S12=12	Escape guard time is 240 ms. (12*20). Modems usually set S12=50.
	W2	Report connection in "CONNECT bps" format. Not all modems accept this command. An alternative is to use Q0 with X1 or X4, and V1.
	&K3	Enables bi-directional RTS/CTS flow control. The modem uses CTS to control flow from the Newton, and the Newton uses RTS to control flow from the modem. This does not work on all modems. An alternate form is \Q3\X0. It is possible that &R0 and \D1 will be required as well.
	&Q5	Use reliable mode. Again, this does not work on all modems. An alternate form is to use &M4 or \N6.
	\N6	Try to establish a reliable LAPM link, and if that fails, try to establish an MNP link, and if that fails, disconnect. You could also try \N4, especially for cellular connections.
	%C1	Enable bilateral MNP 5 or V.42bis data compression. (Note that this can be interpreted differently on different modems.)
	\M1	Enable V.42 detection phase.
kconfigStrECAndFallback		The configuration string used for data connections that allow error corrected communication, and if error correction negotiation fails, the modem falls back to a non-error corrected connection. This configuration string must enable speed buffering and can be used only if hardware flow control can be enabled. The default string is nil. Here is an example:
	E0	Echo off (always required.)
	&C1	DCD indicates the true state of the remote carrier.
	S12=12	Escape guard time is 240 ms. (12*20). Modems usually set S12=50.
	W2	Report connection in "CONNECT bps" format. Not all modems accept this command. An alternative is to use Q0 with X1 or X4, and V1.
	&K3	Enables bi-directional RTS/CTS flow control. The modem uses CTS to control flow from the Newton, and the Newton uses RTS to control flow from the modem. This does not work on all modems. An alternate form is \Q3\X0. It is possible that &R0 and \D1 will be required as well.
	&Q5	Use reliable mode and fall back depending on the value in register S36. Again, this does not work on all modems. An alternate form is to use &Q9, &M4 or \N7.
	%C1	Enable bilateral MNP 5 or V.42bis data compression. (Note that this can be interpreted differently on different modems.)
	\M1	Enable V.42 detection phase.
kconfigStrCellular		If the modem supports cellular connections, set this constant to the cellular configuration string, otherwise it should be nil. This applies to data only. No FAX equivalent exists. Most modems have an alternate default configuration that is used to set cellular mode, although some sense the handset automatically. This alternate default could be AT&F1 or AT&F5. At the very least, this should be prefixed to the normal data configuration string. Some manufacturers suggest using \N4 or \N5 instead of \N6 or \N7.
kconfigStrDirectConnect		The configuration string used for data connections for modems that have no speed buffering, and have no error correction or compression built in (kdirectConnectOnly is set to true). The default string is as follows:
	E0	Echo off (always required.)
	&C1	DCD indicates the true state of the remote carrier.
	S12=12	Escape guard time is 240 ms. (12*20). Modems usually set
	W2	Report connection in "CONNECT bps" format. Not all modems accept this command. An alternative is to use Q0 with X1 or X4, and V1.
	&K0	Disable serial port flow control. The Newton must be dynamically configured to match speeds with the modem's negotiated speed. This does not work on all modems. An alternate form is \Q0\X0.
	&Q0	Use direct connect mode. Again, this does not work on all modems. An alternate form is to use \N1.
	%C0	Disable data compression. (Note that this can be interpreted differently on different modems.)

When the modem tool establishes communication with the modem through an endpoint, a configuration string is normally sent to the modem (as long as kuseConfigString is `true`). There are several configuration strings defined in a typical modem profile, and the one that is sent depends on the type of connection requested and other parameters set in the modem profile. Table 4 summarizes when each kind of configuration string is used:

TABLE 4. SUMMARY OF CONFIGURATION STRING USAGE

Configuration String	When Used
kconfigStrNoEC	The default configuration used for data connections when kdirectConnectOnly is nil. Also used for FAX connections.
kconfigStrECOnly	Used for data connections that require error correction. This configuration string is used only if requested by an application. The constant ksupportsEC must be true for this configuration string to be used.
kconfigStrECAndFallback	Used for data connections that allow error correction, but can fall back to non-error corrected mode. This configuration string is used only if requested by an application.
kconfigStrCellular	The default configuration used for cellular data connections when ksupportsCellular is true.
kconfigStrDirectConnect	The default configuration used for data connections when kdirectConnectOnly is true.

IMPORTANT: MODEM TOOL REQUIREMENTS

It is important that modem setup developers understand the basic requirements and expectations of the Newton modem communications tool. This tool expects a modem to have the following characteristics:

- The modem tool expects a PCMCIA modem to use a 16450 or 16550 UART chip.
- Hardware flow control is expected in both serial and PCMCIA modems. In modems not supporting hardware flow control, direct connect support is required, and the modem profile parameter `kdirectConnectOnly` must be set `true`. This means that the modem tool and the modem must be running at the same bit rate, allowing for no compression or error correction protocols to be used by the modem. (When operating in direct connect mode, the data rate of the modem tool is automatically adjusted to the data rate stated in the "CONNECT XXXX" message.)
- The modem tool expects control signals to be used as follows:
- The modem tool uses RTS to control data flow from the modem

- The modem uses CTS to control data flow from the modem tool.
- Support of the DCD signal is optional. In general, the modem tool expects DCD to reflect the actual carrier state. The usage of this signal by the modem tool is governed by the `kuseHardwareCD` constant.
- The modem tool expects the CONNECT XXXX message to report the modem-to-modem connect speed – not the computer-to-modem serial interface speed.
- The modem tool expects non-verbose textual responses from the modem.
- The modem tool expects no echo.
- The modem tool supports the Class 1 protocol for FAX connections. The configuration string defined by the constant `kconfigStrNoEC` is used for sending FAXs. Additionally, these other requirements apply to the FAX service:
- Flow control is required. In modems not supporting hardware flow control (where `kdirectConnectOnly = true`), XON/XOFF software flow control must be enabled.

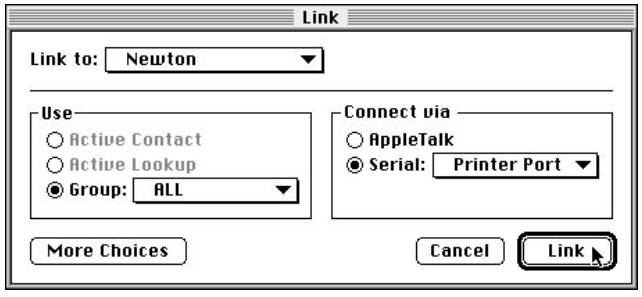
NTJ

continued from page 1

Making the ACT! Connection

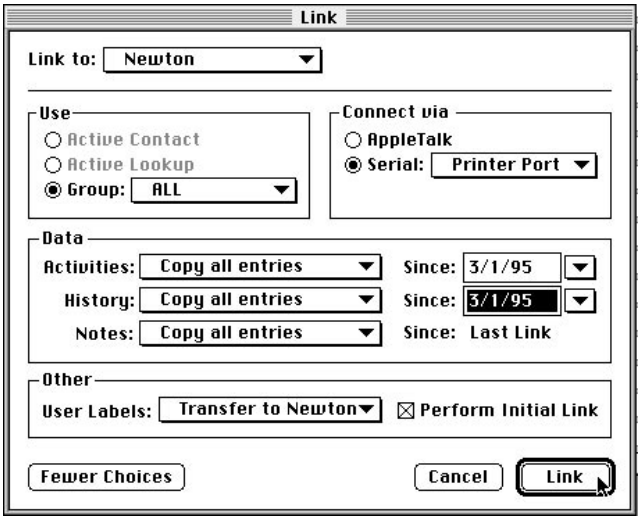
important to us. We knew that if data synchronization was a long slow process, people would not take advantage of it; simplicity for the user was another of our key concerns. Using the DILs allowed us to move data from one platform to the other in a simple, quick way – much more easily and rapidly, in fact, than we could have without them. Using the DILs also meant that we would be able to add efficient synchronization to our product without having to write our own code!

The typical ACT! user is not a computer expert, but what we like to refer to as a "computer casual user." People like these do not particularly want to know *how* we are moving data from the Macintosh or Windows machine to their Newton – just that it can be done without much thought. With the DILs, we were able to create a one-button synchronization that is extremely easy to use, but still provides a very sophisticated link between desktop and Newton.



On the desktop, you select "Link to Newton" and on the Newton device you select "Link", and all of your contact information, notes, histories, and activities are synchronized intelligently — that is, the information is exchanged between the machines in a two-way synchronization.

This is an ideal solution for Newton users who previously had to duplicate their efforts. They might have scheduled an activity on their Newton and then had to schedule that same event on their desktop machine as well. Not any more! Whether they have added four new contacts to the Newton or five new meetings to the desktop machine, the next time that they link, the two-way synchronization exchanges the data for them automatically.



Of course, the user may want to select how much data is sent between the Newton and the desktop; for this, we added a "More Choices" dialog. Here, we let the user filter and select the information to meet their individual needs. For example, some users might want to copy all their notes from the desktop system to the Newton for reference information, or they might want to copy just the history information on each contact from a specific date.

From our early testing of ACT! for Newton, it has become clear that direct connectivity offers the exact solution our users wanted. The simple "plug and sync" solution has been a big hit with those who have used the product.

NTJ

AppleTalk and Complex State Machines

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

Introduction

In this article we introduce several AppleTalk concepts and discuss the use of AppleTalk on the Newton. We will then talk about a general solution to the problem of Newton communications that includes the ability to support a complex state machine.

AppleTalk

AppleTalk is an ISO-compatible network protocol. Figure 1 shows the AppleTalk protocol stack with the protocols most significant to the Newton highlighted.

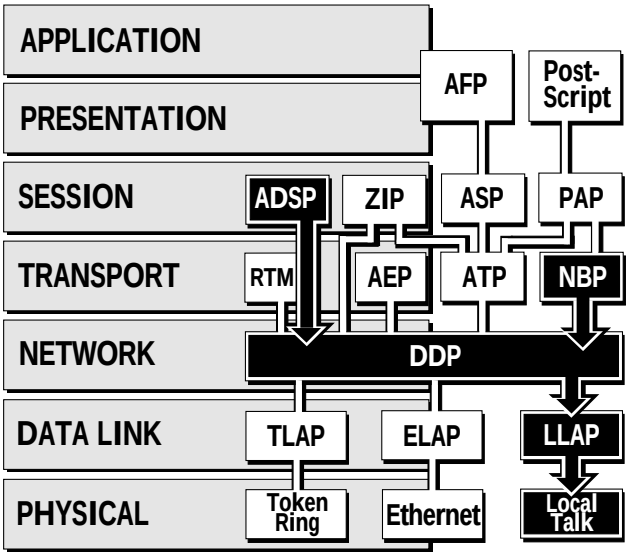


Figure 1: AppleTalk Protocol Stack

At the physical level you would use the LocalTalk format going out through the Newton's SCC port. The LLAP (LocalTalk Link Access Protocol) forms the packets that go out over the physical layer. These protocols are not accessible on the Newton from NewtonScript.

DDP (Datagram Delivery Protocol) is a service built on top of LLAP. The DDP provides "best effort" delivery of data packets. As with LLAP, DDP is not accessible from NewtonScript.

NBP (Name Binding Protocol) is a protocol built using DDP. NBP allows names to be assigned to network entities so that they can be referenced without knowing their network addresses. Certain NBP functions are available from NewtonScript as described later in this article.

ZIP (Zone Information Protocol) is a protocol that allows network entities to get information about other local networks (called zones) in an internet. There are a few ZIP routines available from NewtonScript.

ADSP (AppleTalk Data Stream Protocol) is a data stream protocol built on DDP. ADSP guarantees delivery of data in the order sent. It is used extensively on the Newton to send and receive data through a NewtonScript endpoint.

The other protocols are not used or are not available from NewtonScript and are not described here. For details on these, see *Inside AppleTalk* from Addison-Wesley, or more recently, *Inside Macintosh: Networks*.

Sockets and Nodes

Each device on an AppleTalk network is a node and is given a unique ID number on the local network. These devices can include Macintosh computers, routers, Newtons and any other device capable of running AppleTalk. Each network in its turn has a unique ID number, so the combination of the network ID and the node ID uniquely defines a device.

Within each node there are as many as 127 virtual connections called *sockets*. Sockets are software connections that are multiplexed through a single physical connection. Usually, sockets are associated with a single use, often a single application. For example, in machines running filing protocol software, a socket is opened for all file transfers across the network.

Network Visible Entities

A Network Visible Entity (NVE) is a service that has made itself available through a socket. For example, if a database on a Macintosh registers its service on a node through a socket, other machines can select this service and can connect and access the database through this NVE.

In order for an NVE to be visible to other nodes it must register a name on the network. This is done using the Name Binding Protocol as described below.

NBP (Name Binding Protocol)

The Name Binding Protocol (NBP) allows a text name to be associated with a service. Each NVE that wants to register its service must pass NBP the name it wants to use.

Each name has three parts:

- The entity name, which is a string of as many as 31 characters
- The entity type, which usually describes the service, and is a string of as many as 31 characters
- The name of the zone (local network) which is again a 31-character string

The colon character (:) separates the name from the type and the “at” character (@) separates the type from the zone name. For example: *LlamaBase:LlamaShare@LlamaWorld* describes an entity name "LlamaBase", of type "Llama Share”, in the zone named "LlamaWorld". Taken in its entirety this NBP name is called an NBP *tuple*.

In searching for a network visible entity on a local network there are three special characters that are useful: the asterisk (*), the “equals” sign (=), and the “about equals” sign (≈)(which is made by typing Option-x). These special wildcard characters can be used when you don't know or don't want to specify the exact name of an NVE. Note that these characters cannot

be used in giving yourself a name; they can only be used to for a search.

The asterisk (*) is used when specifying the zone to be searched. It means that the default (local) zone should be used. So, *LlamaBase:LlamaShare@** looks for the NVE called "LlamaBase" of type "LlamaShare" in the local zone.

The “equals” (=) and “about equals” (≈) characters are used to specify part or all of the entity name or type. The difference between them is that the equals (=) character substitutes for an entire name or type, while the “about equals” character (≈) is a partial substitution. For example the string *=:Llama≈ @** describes any NVE in the local zone whose type begins with "Llama". So the NVEs *LlamaBase:Llama Share@Llama World* (assuming *Llama World* was the default zone) and *Fred:Llama Llama@Llama World* are both matched by the example string while *Fred:ShareLlamas@Llama World* and *Fred:LlamaShare@Another Zone* would not be matched.

Figure 2 shows an example of a network with the local zone *LlamaWorld* containing a Macintosh and a Newton, each with a node address and two sockets open. On the Macintosh, the application *LlamaBase* has registered as an NVE of type *LlamaShare* and an entity name of *LlamaBase*, and thus should be visible to the Newton without specifying the zone name.

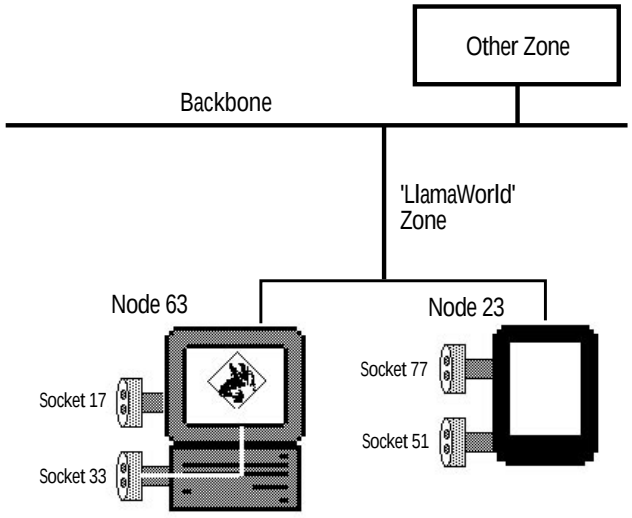


Figure 2: Example Network

ZIP (Zone Information Protocol)

The Zone Information Protocol (ZIP) is used to get information about other networks or zones in an internet. Each router maintains a table of information about networks that is dynamically updated as the internet changes. A node may access this information using ZIP to find out about other zones in the internet and, using NBP, other NVEs in other zones.

Newton Functions

When using AppleTalk functions on the Newton you must first open the AppleTalk driver by calling the global function `OpenAppleTalk()`. This is usually done at startup (for instance, in `viewSetupDoneScript`) or when AppleTalk is chosen from a number of different communications options.

Similarly, when the communication code is finished, you should call `CloseAppleTalk()`; this is usually in the `viewQuitScript` or when the application is done using AppleTalk. If you forget to close AppleTalk and then try to open it again, you will get an error (error ID = -12014). In the future, the AppleTalk driver will not need to be opened or closed at all from NewtonScript but for the moment the key thing to remember is that you must always explicitly close the driver.

To get information about available zones, there are three global functions you can use:

```
HaveZones()       // returns true if there are zones
GetMyZone()       // returns the Newton's zone
GetZoneList()  // returns array of all zones
```

Note that to be compatible with AppleTalk on the Macintosh, all AppleTalk functions on the Newton must return 0 if there is no error, as opposed to returning the value `nil` that is usually used in NewtonScript.

The following NBP calls are available on the Newton:

```
NBPStartLookup(nve)  // starts NBP search for entity
                              // described by 'nve'
NBPLookupCount()     // number of matches found so far
NBPGetLookupNames()  // Returns array of matches found
NBPStopLookup()      // stops search for matches
```

An NBP lookup can take an arbitrary amount of time depending on the size of the internet being searched, so the basic strategy for doing a lookup is to start a search by calling `NBPStartLookup()` (perhaps in the `viewSetupFormScript`) and then periodically checking (in a `viewIdleScript`, for instance) to see whether the list of names returned by `NBPGetLookupNames()` has stopped growing. If it has, then it is probably the case that you have found all candidates across all networks within a reasonable distance from your Newton, and you would then call `NBPStopLookup()`. An example of this technique is shown below.

in viewSetupFormScript for base view...

```
nve:="=:LlamaShare@*";     // look for all LlamaShare's
NBPStartLookup(nve);       // ...entities in local zone
:SetupIdleScript(1000);  // call idlescript in 1 sec
gNumEntities:=0;           // haven't found any yet
```

in viewIdleScript

```
local numberFound;

numberFound:=NBPLookupCount();
if numberFound =gNumEntities then
   return nil;      // no more idle calls
else
begin
   gNVEList:=NBPGetLookupNames();  // use what's found
   return (1000);     // call again in a second
end;
```

One important thing to note is that, primarily because of battery considerations, Newtons cannot register network-visible entities. While your Newton application can connect to entities discovered in a lookup, it cannot register an NVE to which other machines can connect. This is largely a question of who initiates the connection. The Newton must start the connection; thereafter, as we will see below, communication is bidirectional and full-duplex.

The next question is how to allow a Newton user to choose among NVEs. You could take the array of names returned by `NBPGetLookupNames()`, put up a floater to display the list, and allow the user to pick an entity from the list — but it turns out that Apple has already provided this in the form of a root object called `NetChooser`. The `NetChooser` is the Newton equivalent of the Chooser on the Macintosh. It is used to select from among NVEs. Figure 3 shows the slip that `NetChooser` puts up while it is looking for candidate NVEs:

Figure 3: NetChooser SIip

After it has found all NVEs meeting the specified criteria, the box changes to a list of choosable NVEs:

You invoke `NetChooser` by calling the following method:

```
GetRoot().NetChooser:OpenNetChooser(zone, name,
   startselection, who, connect, header, lookfor);
   // puts up slip that lists NVEs to choose from
```

The `zone` argument is the zone to look in for the entity specified by the `name` argument. If `zone` is `nil` (*not* "*") then the default (local) zone is used.

The `name` is the NVE name without the zone but with the "at" sign (@) at the end of the name (for example, *=:anyName@)*.

The `startselection` argument is an NVE name for the item that will be highlighted when the `NetChooser` slip NVE list is displayed.

The `who` argument is a reference to the view to which `NetChooser` will send a message when the user has chosen an item.

The `connect` argument is a string that is displayed in the button below the selection list. (In Figure 3 it is the string "Pick this one".)

The `header` argument is a string that appears above the selection list ("Select a Llama Server:" in Figure 3.)

The `look for` argument is the string that appears in the selection box while `NetChooser` is compiling the list of entities to display ("Looking for Servers..." , in Figure 3.)

An example of the `NetChooser` call for the slip shown in Figure 3 would be:

```
GetRoot().NetChooser:OpenNetChooser(nil,
   "=:LlamaShare@",nil,self,"Pick this one","Llama
   Server","Servers");
```

After the user selects an item from the list displayed, `NetChooser` sends the view specified by the `who` argument the message *NetworkChooserDone* as shown here:

```
// method called in 'who' view after selection made
NetworkChooserDone(curSelection, curZone)
```

As with `OpenNetChooser`, the zone will be `nil` if the zone where the selection was made is the local zone, so make sure that this is converted to the asterisk (*) character if the NVE name is saved or used in your application.

ADSP

The Newton currently uses the ADSP (AppleTalk Data Stream Protocol) to send and receive data through AppleTalk endpoints. As with other endpoints, the implementation of these calls is hidden from the user; to the programmer sending and receiving data once a connection is established looks very much the same as it does for other endpoints. However, it is worth describing the characteristics of ADSP briefly, so that you have an idea of how it will behave.

ADSP is a full-duplex, bidirectional protocol with guaranteed delivery of data in the order sent. This means that you can send and receive at the same time (no modes) and that what you send will either get to the receiver in the exact byte-order you sent it, or you will get an error message back. Data acts as if it is traveling down a pipeline with each byte in order numbered, so that if data is received out of order, the receiver is able to ask the sender for the missing data. This mechanism is implemented below the ADSP client's API level, and thus is invisible to the caller.

To use an AppleTalk endpoint, you will first use `NetChooser` or some other method of deciding what entity the user wants to connect with. Then you define an endpoint using the NVE address as the endpoint address, send the `Instantiate` and `Connect` messages as before, call the `Send` or `SendFrame` messages for outgoing data and use `InputSpec` frames to receive.

An example of an ADSP endpoint definition is shown below:

ADSP Endpoint Definition

```
myADSPep:= {
 _proto: protoEndpoint,
 configOptions: [
  { label: kCMSAppleTalkID, type: 'service,
    opCode: opSetRequired },
  { label: kCMSAppleTalkID, type: 'option,
    opCode: opSetRequired,
    data: kCMOAppleTalkADSP },
  { label: kCMDEndpointName, type: 'option,
    opCode: opSetRequired,
    data: kADSPEndpoint }
  { label: kCMARouteLabel, type: 'address,
    opCode: opSetRequired,
   data: {
     addressType: kNamedAppleTalkAddress,
     addressData: "LlamaFarm:LlamaServer@*"
  }
  }
 ]};
```

As of this writing, there are two bugs that you should know about. The first bug is caused by the Newton's habit of always trying to open a socket with the same ID when AppleTalk is first started. Normally this doesn't matter, but in at least two instances it can cause problems. The first scenario in which this socket ID can be a problem is if the Newton crashes and the user restarts the application too quickly. In this case, machines that had been connected to the Newton won't see the Newton crash; when the same socket ID is used it will connect the "new" Newton socket with its "old" connection. The result is usually an instance of cross protocols, because the Newton thinks it is a new connection when the other machine does not. In other words, your Newton state machine may be trying to log on to a server, while the server thinks the Newton is connected and is waiting for the next request.

There is no simple solution to this other than to wait for about two minutes, then the AppleTalk connection will time out and the other machine will tear down the connection. Usually this problem occurs when you are debugging your application, but of course if a user has a crash after the product ships, this may affect him or her.

The second problem the socket ID causes is very similar to the first: if the user quits an application that has an open connection and then restarts it quickly, the same situation may arise, even if the application disconnected.

This is because the disconnection may not have fully occurred, since it happens in a different thread. Again, the other machine may think it is still connected when the new connect request comes in.

In this latter case you have a little more control over matters. Here, you can force the situation by not quitting the application until the connection has actually been destroyed. This can be done in several ways; probably the preferred way is to set up a delayed action to handle disconnecting and disposing. The delayed action should then set a state variable in your root view to reflect that you have disconnected. Then if the user restarts too quickly, the state variable will still show the endpoint as being connected.

The second bug is more serious and to deal with it completely you must control transmission speeds on the Newton. When an endpoint is instantiated and connected, a direct link between the NewtonScript endpoint and the Comms Tool is formed. Requests and responses pass between them through a buffer, as shown in Figure 4 below.

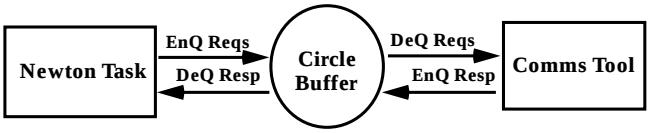


Figure 4: Request Buffer

This situation can create a deadlock problem where the Comms Tool wants to put a response into the buffer but cannot because the buffer is full, and the Newton Task cannot take out a response, because it is busy trying to put in a request. This is akin to the classic "Dining Philosopher's Problem" posed so often in introductory computer programming classes.

For most endpoints this situation never occurs, as the requests and responses are sparse enough and are handled quickly enough that the respective tasks are able to keep the circular buffer from filling. However, AppleTalk is a fairly "chatty" set of protocols, particularly in the version currently in use on the Newton. Thus this situation can occur when there is a lot of high-speed traffic between the Newton and another machine. Once this occurs, communication becomes locked up and the Newton must be restarted to clear the buffer.

The solution to this problem at the moment is to "throttle back" transmission of data so that the respective tasks get a chance to clear the buffer before the next request is sent. An example of this is shown in the *LlamaTalk* code included at the end of this article. This code uses a `viewIdleScript` in the `protoLlamaTalk` user proto to break outgoing data into 1K chunks and to send chunks only every `fIdleSpeed` milliseconds.

PROTOLLAMATALK

Theory of Operation

Jim Schram of Apple Computer's PIE DTS group developed a user prototype called `protoLlamaTalk`. Because it handles so many issues so well, we have included it here to give it additional distribution.

The `protoLlamaTalk` proto is a `clView`-based prototype with no visual components in the layout (although it does draw some shapes to show the state of a connection). It supports a binary data protocol in which an object is sent with a header describing the type and length in bytes, followed by a binary stream of data bytes. This protocol is used for both input and output of data.

Output is transaction-based with the methods `MBeginTransaction` starting an outgoing transaction and the methods `MWrite` and `MWriteObject` being used to post data to the current transaction. A transaction is closed with a call to `MEndTransaction`. The data is not actually sent when `MWrite` and `MWriteObject` are called; instead the data is queued in the `fWriteQueue` for later transmission. Each call to these methods adds a new binary item to the queue.

All data is actually sent from LlamaTalk's `viewIdleScript`. Because of the problem described above, ADSP data transmission is "throttled back" as noted above by setting the time between idle calls to a larger value. This avoids filling the request queue, and causing gridlock. Whether the ADSP protocol selected is or not, data is sent in chunks of 1024 bytes (or fewer if there is there are fewer than 1024 bytes remaining in the outgoing queue) from `viewIdleScript`. The data being sent is actually moved from the `fWriteQueue` to the `fOutputQueue` by queuing up the transaction queue frame in the output queue. The `viewIdleScript` then sends the data by removing the objects from this queue and sending them a chunk at a time.

Input is controlled by two key input specs: `FInputHeaderComponent` and `FInputDataComponent.` The first input spec is set when new data is expected starting with the protocol header information. Thereafter the `FInputDataComponent` input spec is used until all data are received in the incoming binary object. Data is received 512 bytes at a time and is in turn put onto the `fInputQueue.` To remove data from this queue, an application can simply define a script called: `MLlamaTalk_InputScript,` That will be called in the `viewIdleScript` if there is data in the queue. This application input script will be passed the next chunk of data from the queue and may do with it as it pleases since it is outside of the queue before being passed to the application.

```
// Copyright © 1994 Apple Computer, Inc. All rights reserved

constant kAppSymbol := '|LlamaTalk:PIEDTS|;
constant kAppName := "Llama Talk 1.0a7";

constant kMaxAppWidth := 240;     // original MP width
constant kMaxAppHeight := 336;    // original MP height

// ---- End Project Data ----
```

File LlamaTalkMain.t

```
// Before Script for "vDali"
// This file and project is Copyright © 1994 Apple Computer, Inc. All rights reserved.

vDali :=
  {viewFormat: 83951953,
   viewBounds: {left: 2, top: 2, right: 242, bottom: 338},
   title: "",
   viewSetupFormScript:
    func()
    begin
        // make view no bigger than the original MP
     local b := GetAppParams() ;
     viewBounds := RelBounds( b.appAreaLeft, b.appAreaTop,
                         MIN( b.appAreaWidth, kMaxAppWidth ),
                         MIN( b.appAreaHeight, kMaxAppHeight ));
    end,
   viewSetupDoneScript:
    func()
    begin
     :DoUpdateButtons();
    end,
   DoConnect:
    func()
    begin
     :DoMessage("Connecting…");
     if not vLlamaTalk:MIsOpen() then
       vLlamaTalk:Open();
```

```
    if vProtocol.clusterValue = 1 then begin
        vLlamaTalk:MSetProtocol('ADSP);
        vLlamaTalk:MSetAddress("Echo Server:EchoLlama@*");
      end
    else begin
        vLlamaTalk:MSetProtocol('MNP);
        vLlamaTalk:MSetAddress(nil);
      end;

    vLlamaTalk:MSetQuietDisconnect(true);

    local err := nil;
    if vAsync.viewValue then
      err := vLlamaTalk:MConnectAsync()
    else if not err := vLlamaTalk:MConnect() then
      :DoMessage("Connected, waiting for disconnect…");

    if err then begin
        :DoMessage("Ready for connect...");
        vLlamaTalk:Close();
      end;

    :DoUpdateButtons();
  end,
 DoDisconnect:
  func()
  begin
   vLlamaTalk:MDisconnect();
   vLlamaTalk:Close();
   :DoUpdateButtons();
   :DoMessage("Ready for connect…");
  end,
 DoMessage:
  func(message)
  begin
   SetValue(vMessage, 'text, Clone(message));
   RefreshViews();
  end,
 DoUpdateButtons:
  func()
  begin
   if vLlamaTalk:MIsOpen() then begin
        if vLlamaTalk:MIsConnected() then
          SetValue(vConnect, 'text, "Disconnect")
        else if vLlamaTalk:MIsConnecting() then
          SetValue(vConnect, 'text, "Stop Connecting")
        else
          SetValue(vConnect, 'text, "Connect");

        SetValue(vOpenClose, 'text, "Close");

        if vLlamaTalk:MIsVisible() then
          SetValue(vShowHide, 'text, "Hide")
        else
          SetValue(vShowHide, 'text, "Show");
        vShowHide:Show();
      end

    else begin
        SetValue(vConnect, 'text, "Connect");
        SetValue(vOpenClose, 'text, "Open");
        vShowHide:Hide();
      end;
  end,
 MLlamaTalk_InputScript:
  func(data)
  begin
   local s := ExtractCString(data, 0);
// note: ExtractCString is currently undocumented. Some of the Extract/Stuff routines have special
// restrictions and workarounds. Read the PIEDTS "Utility Functions" Q&A for more information.
    :DoMessage(s);
  end,
 MLlamaTalk_StdError:
  func(messageText, errorNum)
  begin
   :DoUpdateButtons();
   if errorNum = 0 then
     local text := messageText
   else
     local text := messageText & "\n" & NumberStr(errorNum);
   :DoMessage(text);
   GetRoot():Notify(kNotifyAlert,
     EnsureInternal(kAppName), EnsureInternal(text));
  end,
 fLlamaTalkState:
      // this is a REQUIRED slot for use with protoLlamaTalk it MUST have the initial value of NIL
      // when compiled it orchestrates the connect and disconnect process
  nil,
 MLlamaTalk_StateChanged:
  func()
  begin
   :DoUpdateButtons();
  end,
 _proto: protoApp,
 debug: "vDali"
}
```

File protoLlamaTalk

```
// Before Script for "LlamaTalk"
/*  protoLlamaTalk

    This layout file is copyright © 1994 Apple Computer, Inc. All rights reserved.

    Modification Status
    YY/MM/DD Name          Comments
    94/09/22 Jim Schram    Released as 1.0a7 - removed power-off patch code
                           (install "MP110 Power Off Update.pkg" instead)
    94/07/07 Jim Schram    Released as 1.0a6 - modified to use kViewIsOpenFunc &
                           kLlamaTalkStuffCStringFunc
    94/06/21 Jim Schram    Released as 1.0a5 - added improved CloseAppleTalk support
    94/06/10 Jim Schram    Released as 1.0a4 - added dynamic GotoSleep patch code
    94/05/13 Jim Schram    Released as 1.0a3 - added support for typed transaction data
    94/05/04 Jim Schram    Released as 1.0a1
    94/03/17 Jim Schram    Initial Development
    ------------------------------------------------------------------------
    The LlamaTalk packet format is as follows:
    1 byte packet type
    3 bytes packet length (MSB -> LSB order)
    N bytes packet data

    Currently only packet type cLTPacketType_Simple (zero) is implemented.
    Future packet types will include features such as compression and encryption.
    All other packet types are reserved.
*/

DefConst('kDebugLlamaTalk,                    nil);
// true = print comms debugging statements on bunwarmers, nil = supress them
DefConst('kLlamaTalkState_Disconnected,     nil);
// ready-to-go (default state)
DefConst('kLlamaTalkState_Connect,        1);
// preparation for (asynchronous) connect
DefConst('kLlamaTalkState_Connecting,       2);
// in-process of (asynchronous) connect
DefConst('kLlamaTalkState_Connected,     3);
// connected (requires disconnect)
DefConst('kLlamaTalkState_Disconnecting,    4);
// in-process of (asynchronous) disconnect

DefConst('kLlamaTalkType_nil,          0);
// transaction data item types, as used in :MWriteObject() -- DO NOT RENUMBER!
DefConst('kLlamaTalkType_true,        1);
DefConst('kLlamaTalkType_char,        2);
DefConst('kLlamaTalkType_integer,     3);
DefConst('kLlamaTalkType_real,     4);
DefConst('kLlamaTalkType_string,   5);
DefConst('kLlamaTalkType_binary,   6);

DefConst('kLlamaTalkError_IllegalOperation,  -666);
DefConst('kLlamaTalkError_EndpointInUse,      -667);
DefConst('kLlamaTalkMessage_EndpointInUse,"Another application
seems to be using the communications port.");

DefConst('kLlamaTalkNewBinaryFunc, func(size)
SetLength(SetClass(Clone(""), 'binary), size));

// Note: StuffCString is currently undocumented. Use it at your own risk. Some of the Extract/Stuff
// routines have bugs or special workarounds. A forthcoming item in the "Q&A - Utility Functions"
// document will discuss the workarounds.
DefConst('kLlamaTalkStuffCStringFunc, func(theObject, theOffset,
theString) begin    // modifies and returns theObject
     local len := StrLen(theString);
     if len < 4 then begin        // work-around for StuffCString bug
          for i:= 0 to len - 1 do
            StuffChar(theObject, theOffset + i, theString[i]);
          StuffByte(theObject, theOffset + len, 0);
       end
```

```
    else
      StuffCString(theObject, theOffset, theString);

    return theObject;
  end
}

DefConst('kLlamaTalkQueueTemplate,
// create a new queue like this:  kLlamaTalkQueueTemplate:MInstantiate('FIFO);
{
  MInstantiate:   func(queueType)
  // currently 'FIFO (first-in-first-out) and 'FILO (first-in-last-out) queues are supported
    begin
      if queueType = 'FIFO or queueType = 'FILO then
        return { _proto:  self,
          fType:    queueType,    // 'FIFO or 'FILO
          fNbElem: 0,             // 0 - N
          fData:    [], }         // array [] of fNbElem enqueued items
      else
        return nil;
    end,

  MDispose:         func()
                  begin
                    fType := nil;
                    fNbElem := nil;
                    fData := nil;
                    return nil;
                  end,

  MReset:         func()
  // empties the queue, allowing potential garbage collection to occur
                  begin
                    if fType then begin
                        fNbElem := 0;
                        SetLength(fData,0);
                      end;
                    return nil;
                  end,

  MEnQueue:       func(data)
  // adds a non-nil data item to a queue according to queue type
                  begin
                    if data then
                      if fType = 'FIFO or fType = 'FILO then begin
                          fNbElem := fNbElem + 1;
                          SetLength(fData, fNbElem);
                          fData[fNbElem - 1] := data;
                          return nil;
                        end;
                    return data;
                  end,

  MDeQueue:       func()
  // removes a data item from a queue according to queue type
          begin
            if not fType or fNbElem <= 0 then
              return nil;
            if fType = 'FIFO then begin
                fNbElem := fNbElem - 1;
                local data := fData[0];
                ArrayMunger(fData, 0, 1, nil, 0, 1);
              end
            else if fType = 'FILO then begin
                fNbElem := fNbElem - 1;
                local data := fData[fNbElem];
                ArrayMunger(fData, fNbElem, 1, nil, fNbElem, 1);
              end
            else
              local data := nil;
            return data;
          end,

  MGetQueueSize: func()
  // return the number of data items in the queue
                  begin
                    if fType = 'FIFO or fType = 'FILO then
                      return fNbElem
                    else
                      return 0;
                  end,

  MGetDataSize:   func()
  // return the sum of the binary Length() of each data item in the queue
                  begin
                    local len := 0;

                    if fType = 'FIFO or fType = 'FILO then
                      foreach item in fData do
                        len := len + Length(item);

                    return len;
                  end,
});

// the following constants (kDA_StatusFrame, kDA_ActionFunc, kDA_AddDeferredAction) are used to
// implement "killable" deferred actions --> see MAddDeferredAction for more info

DefConst('kDA_StatusFrame,
          {  fRunIt: true,
             fRanIt: nil,
             RanIt:  func()
                     fRanIt,
             KillIt:  func()
                     begin
                       fRunIt := nil;
                       fRanIt;
                     end,
          });
DefConst('kDA_ActionFunc,
          func(fn, args, status)
          begin
            status.fRanIt := true;
            if status.fRunIt then
              Apply(fn, args);
          end );
DefConst('kDA_AddDeferredAction,
          func(fn, args)
          begin
            local status := {_proto: kDA_StatusFrame};
            AddDeferredAction(kDA_ActionFunc, [fn, args, status]);
            status;
          end );

LlamaTalk :=
  {
   MWriteObject:
    func(data)   // WARNING:  This func contains work-arounds and currently
                          // undocumented functions.  Use at your own risk!
    begin
     if fLlamaTalkState <> kLlamaTalkState_Connected then
       return;

     IF kDebugLlamaTalk THEN LT_PRINT("Enqueueing data...");

     local translation := nil;

     if not data then begin // Note: if-then-else tree is structured for common-case efficiency
          translation := :MNewBinary(1);
          StuffByte(translation, 0, kLlamaTalkType_nil);
                                                          // object type = nil
       end

     else if IsInstance(data, 'string) then begin
          local len := StrLen(data);
          translation := :MNewBinary(len + 4);
          StuffByte(translation, 0, kLlamaTalkType_string);
                                                          // object type = string
          StuffByte(translation, 1, len >> 8); // MSB of length
          StuffByte(translation, 2, len);         // LSB of length
          call kLlamaTalkStuffCStringFunc with (translation,
                    3, data); // N single byte chars (Mac encoding for chars 128-255)
          SetLength(translation, len + 3);
                          // get rid of that zero terminator byte from StuffCString
       end

     else if IsInstance(data, 'int) then begin
          translation := :MNewBinary(5);
          StuffByte(translation, 0, kLlamaTalkType_integer);
                                                          // object type = integer
          StuffLong(translation, 1, data);
                                                  // MSB -> LSB of 4 byte sign-extended value
       end

     else if IsInstance(data, 'char) then begin
          translation := :MNewBinary(2);
          StuffByte(translation, 0, kLlamaTalkType_char);
                                                          // object type = char
```

```
        StuffChar(translation, 1, data);
                                // 1 single byte char (Mac encoding for chars 128-255)
      end

    else if IsInstance(data, 'boolean) then begin
        translation := :MNewBinary(1);
        StuffByte(translation, 0, if data then
                kLlamaTalkType_true // object type = true
              else
                kLlamaTalkType_nil);  // object type=nil (=false)
      end

    else if IsInstance(data, 'real) then begin
        local s := NumberStr(data);
        len := StrLen(s);
        translation := :MNewBinary((len*2) + 2);
                    // workaround for bug in StuffPString (overwrites buffer by factor of 2)
        StuffByte(translation, 0, kLlamaTalkType_real);
                                        // object type = real (as an N-char string)
        StuffPString(translation, 1, s);
                            // pascal string representation (length byte followed by chars)
        SetLength(translation, len + 2);
                                            // reduce the buffer to the correct size
      end

    else if IsInstance(data, 'binary) then begin
        local len := Length(data);
        translation := :MNewBinary(len + 5);
        StuffByte(translation, 0, kLlamaTalkType_binary);
                                                    // object type = binary (bytes)
        StuffLong(translation, 1, len);
                                // MSB -> LSB of 4 byte sign-extended length of object
        BinaryMunger(translation, 5, len, data, 0, len);
                                                                // data bytes
      end

    else begin
        IF kDebugLlamaTalk THEN LT_PRINT(
            "Unsupported data class!!!  Cannot enqueue...");
        :?MLlamaTalk_StdError("Unsupported data class (" &
              SPrintObject(ClassOf(data)) & ").  Cannot
              enqueue.", -48215);
      end;

    fWriteQueue:MEnQueue(translation);
// FYI :MEnQueue() won't enqueue a nil (this is good...)

    return translation;
  end,
MAddDeferredAction:
    /*  use this method like you would use AddDeferredAction the difference is that this routine
    returns a frame containing closures which can be used to "cancel" the deferred action before it
    executes and to query whether or not the deferred action has executed.
    Example:
        local x := :MAddDeferredAction( func() GetRoot():SysBeep(), [] );
        if x:RanIt() then print("already executed");
                  else print("waiting to run");
        if x:KillIt() then print("already executed");
                  else print("canceled");

    obviously you'll keep x around in a slot if you ever want to cancel the deferred action...
    */

  kDA_AddDeferredAction         // (fn, args),
viewSetupDoneScript:
  func()
  begin
    AddPowerOffHandler(SELF);   // do not allow the Newton to sleep
                                    // while we're connected
  end,
MIsVisible:
  func()
  begin
    if fBusyShapes then
            true
          else
            nil;
  end,
MRead:
  func()
  begin
    if fInputQueue then
      fInputQueue:MDeQueue()
    else
      nil;
  end,
MIsOpen:
  func()            // this function will only work when 'self' is the protoLlamaTalk view!
  begin         // In other words,  we're called using:  if  view:MIsOpen() then...
    return call kViewIsOpenFunc with (self);
  end,
viewFormat: nil,
MBeginTransaction:
  func()
  begin
    if fLlamaTalkState <> kLlamaTalkState_Connected then
      return;

    IF kDebugLlamaTalk THEN LT_PRINT("Beginning write
transaction...");

    fWriteQueue:MReset();
    return true;
  end,
fDisconnectSlip:
  {
    _proto:            protoFloater,
    viewBounds:          {  left:   0,
                            top:      0,
                            right:  108,
                            bottom:  44,  }
    viewJustify:       vjParentCenterH + vjParentCenterV, // 80

    powerOffScript:      func(what) nil,
    viewSetupDoneScript:func()  AddPowerOffHandler(self),
    viewQuitScript:      func()  RemovePowerOffHandler(self),

    stepChildren:  [
                  { _proto:     protoStaticText,
                    viewBounds:     {  left: 8,
                                       top: 8,
                                       right: 104,
                                       bottom: 40,        }
                    viewJustify:  vjCenterH,   // 2
                    text: "Disconnecting...\nPlease Wait...",
                    }
                ]
  }
viewQuitScript:
  func()
  begin
    :MDisconnect();

    fInputQueue := fInputQueue:MDispose();
    fOutputQueue := fOutputQueue:MDispose();
    fWriteQueue := fWriteQueue:MDispose();

    fEndPoint := nil;
    fEndPointAddress := nil;
    fEndPointConfig := nil;

    fOutputData := nil;
    fOutputPhase := nil;

    fIsQuietDisconnect := nil;
    fBusyShapes := nil;

    RemovePowerOffHandler(SELF);    // allow the Newton to go to
                                        // sleep now we're no longer connected
      if kDebugLlamaTalk then
      RemoveSlot(functions, 'LT_PRINT);
  end,
MGetAddress:
  func()
  begin
      if HasSlot(fEndPointAddress, 'data) and
                HasSlot(fEndPointAddress.data, 'addressData) then
      return fEndPointAddress.data.addressData
    else
      return fEndPointAddress;
  end,
MNewBinary:
  func(size)
  begin
    return call kLlamaTalkNewBinaryFunc with (size);
  end,
viewDrawScript:
  func()
```

```
  begin
   if fLlamaTalkState = kLlamaTalkState_Connected then
     :MDrawBusyIcon('IDLE)
   else
     :MDrawBusyIcon('DISCONNECTED);
  end,
FInputHeaderComponent:
  {
   inputForm:    'raw,
   byteCount:    4,
   inputScript: func(ep, data)
   begin
        local size := ExtractByte(data, 1) << 16 +
              ExtractByte(data, 2) << 8 + ExtractByte(data, 3);
        IF kDebugLlamaTalk THEN
              LT_PRINT("FInputHeaderComponent called,
                byte count =" && NumberStr(size));

        if ExtractByte(data, 0) = 0 then begin
             ep.FInputDataComponent.fData := call
                        kLlamaTalkNewBinaryFunc with (size);
             ep.FInputDataComponent.fDataIndex := 0;
             ep.FInputDataComponent.fDataBytesToGo := size;
             ep.FInputDataComponent.byteCount := if size < 512
                                    then size else 512;
             ep:SetInputSpec(ep.FInputDataComponent);
          end
        else begin
             ep.FInputDataSkipper.fDataBytesToGo := size;
             ep.FInputDataSkipper.byteCount := if size < 512
                                    then size else 512;
             ep:SetInputSpec(ep.FInputDataSkipper);
          end;
     end,
  }
MConnect:
  func()
  begin
   if not :MIsOpen() then     // return an error if this view has not yet been opened!
     return kLlamaTalkError_IllegalOperation;

   if fLlamaTalkState <> kLlamaTalkState_Disconnected then
     return kLlamaTalkError_IllegalOperation;

   :MSetLlamaTalkState(kLlamaTalkState_Connect);

   fEndPoint.fDeferredObj := nil;
   return :MConnectAction(fEndPoint);
  end,
fBusyShapes:
   nil// holds an array of shapes to be drawn during I/O when this view is visible, nil otherwise
   ,
MDisconnectCompProc:
  func(ep, state, slip)  // this routine is called as a deferred action from MDisconnect
  begin
   if state = kLlamaTalkState_Connected then
     ep:Disconnect();
   ep:Dispose();
   ep:MCloseNetStack();

   if slip then
     slip:Close();

   ep:MSetLlamaTalkState(kLlamaTalkState_Disconnected);
  end,
viewFlags: 32,
fEndPointConfigADSP:
  {
   fIODelay:        if kDebugLlamaTalk then 7 else 2,
   // must slow down I/O for ADSP to avoid RPC deadlock bug
      // bunwarmers need extra slow I/O to handle all the LT_PRINT statements
   configOptions:
     [
       { label:    kCMSAppleTalkID,
         type:     'service,
         opCode:     opSetRequired},

       { label:    kCMSAppleTalkID,
         type:     'option,
         opCode:     opSetRequired,
         data:       kCMOAppleTalkADSP  },

       { label:    kCMOEndpointName,
         type:     'option,
         opCode:     opSetRequired,
         data:       kADSPEndpoint},
     ]
  }
MIsConnected:
  func()
  begin
   return fLlamaTalkState = kLlamaTalkState_Connected;
  end,
MConnectAsync:
  func()
  begin
   if not :MIsOpen() then     // return an error if this view has not yet been opened!
     return kLlamaTalkError_IllegalOperation;

   if fLlamaTalkState <> kLlamaTalkState_Disconnected then
     return kLlamaTalkError_IllegalOperation;

   :MSetLlamaTalkState(kLlamaTalkState_Connect);

   fEndPoint.fDeferredObj :=
            :MAddDeferredAction(MConnectAction, [fEndPoint]);
   return nil;
  end,
viewIdleScript:
  func()
  begin
   if fLlamaTalkState <> kLlamaTalkState_Connected then begin
     // if not connected, turn off idle handler
        :Dirty();
        return nil;
     end;

   if :MGetState() < 3 then begin   // if connection has dropped,
                                                  // disconnect and turn off idle handler
        :MDisconnect();
        if not fIsQuietDisconnect then
          // avoid calling the error method if we're supposed to keep quiet about it
          :?MLlamaTalk_StdError( "The connection has
                 closed unexpectedly.", -1);
        return nil;
     end;

   local outputDone := nil;
     // this idle script can be turned off when outputDone and no pending input

   IF kDebugLlamaTalk THEN LT_PRINT("Idle Speed" &&
                        NumberStr(fEndPoint.fIdleSpeed));

   if fInputQueue:MGetQueueSize() > 0 then begin
          // if there is any input data available, call input script if defined
        :DoDrawing('MDrawBusyIcon, ['INPUT]);
        :?MLlamaTalk_InputScript(:MRead());
     end;

   if fOutputPhase = 0 then
        // if there is any data to output, start the output state machine
     if fOutputQueue:MGetQueueSize() > 0 then begin
          IF kDebugLlamaTalk THEN LT_PRINT(
             "Beginning output phase...");
          fOutputData := fOutputQueue:MDeQueue();
             // fOutputData = the transaction to be output
          local len := fOutputData:MGetDataSize();
          // len = total bytes of this transaction
          IF kDebugLlamaTalk THEN LT_PRINT(
              "Total length to be written =" && NumberStr(len));
          fEndPoint:Output([0, len >> 16, len >> 8, len],nil);
             // output the transaction length
          fEndPoint:FlushOutput();

          fOutputPhase := 1;  // go to next state (outputting transaction data)
        end
     else
        outputDone := true;

   else if fOutputPhase = 1 then
        // output chunks of the transaction until all bytes have been output
     if fOutputData:MGetQueueSize() > 0 then begin
          :DoDrawing('MDrawBusyIcon, ['OUTPUT]);

          local buffer := :MNewBinary(0);
             // concatenate transaction data items into a single binary object
          repeat
             local data := fOutputData:MDeQueue();
```

```
            if data then begin
                 local dataLen := Length(data);
                 local bufferLen := Length(buffer);
                 SetLength(buffer, bufferLen + dataLen);
                 IF kDebugLlamaTalk THEN LT_PRINT(
                    "bufferLen = " & NumberStr(Length(buffer)));
                   BinaryMunger(buffer, bufferLen, dataLen,
                                                 data, 0, dataLen);
              end;
         until Length(buffer) >= 1024 or
                                 fOutputData:MGetQueueSize() = 0;
         if Length(buffer) > 0 then begin // output the transaction data chunk
              fEndPoint:Output(buffer,nil);
              fEndPoint:FlushOutput();
            end;
       end
     else begin
          fOutputData := nil; // allow garbage collection on this transaction (queue)
          fOutputPhase := 0; // go to next state
       end;

  if outputDone and fInputQueue:MGetQueueSize() = 0 then begin
          // if no more data to process turn off idle handler
       :DoDrawing('MDrawBusyIcon, ['IDLE]);
       return 15000
          // slow idle handler (check for dropped connections every 15 seconds)
     end
   else
     return fEndPoint.fIdleSpeed; // number of milliseconds to delay until next idle
 end,
MIsQuietDisconnect:
 func()
 begin
  return fIsQuietDisconnect;
 end,
MSetProtocol:
 func(protocol)  // returns the endpoint configuration frame, or
       // nil if protocol symbol is unsupported or view is not open
 begin
  if not :MIsOpen() then
     return nil;
  if IsFrame(protocol) then
     fEndPointConfig := EnsureInternal(protocol)
  else if protocol = 'MNP then
     fEndPointConfig := fEndPointConfigMNP
  else if protocol = 'ADSP then
     fEndPointConfig := fEndPointConfigADSP
  else
     return nil;

  :MResetConfig();

  return fEndPointConfig;
 end,
fEndPointConfig:
 nil // contains the "live" endpoint config frame used during endpoint instantiation
 ,
viewBounds: {left: 100, top: 100, right: 116, bottom: 116},
MCountPendingOutputTransactions:
 func()
 begin
  return if fOutputQueue then
           fOutputQueue:MGetQueueSize()
         else
           0;
 end,
MEndTransaction:
 func()
 begin
  if fLlamaTalkState <> kLlamaTalkState_Connected then
     return;
  IF kDebugLlamaTalk THEN LT_PRINT("Transferring transaction
       to output queue.  Enabling idle handler.");

  fOutputQueue:MEnQueue(fWriteQueue);
  fWriteQueue := kLlamaTalkQueueTemplate:MInstantiate('FIFO);

  :SetUpIdle(100);
  return nil;
 end,
MDrawBusyIcon:
 func(state)
 begin
  if fBusyShapes then begin
       :DrawShape(fBusyShapes[0].fShape,
                                    fBusyShapes[0].fStyle);
          // erase the background, then draw the appropriate icon
       if state = 'DISCONNECTED then
          :DrawShape(fBusyShapes[1].fShape,
                                    fBusyShapes[1].fStyle)
       else if state = 'IDLE then
          :DrawShape(fBusyShapes[2].fShape,
                                    fBusyShapes[2].fStyle)
       else if state = 'INPUT then
          :DrawShape(fBusyShapes[3].fShape,
                                    fBusyShapes[3].fStyle)
       else if state = 'OUTPUT then
          :DrawShape(fBusyShapes[4].fShape,
                                    fBusyShapes[4].fStyle)
       else
          :DrawShape(fBusyShapes[1].fShape,
                                    fBusyShapes[1].fStyle)
     end;
 end,
MCountPendingInputTransactions:
 func()
 begin
  return if fInputQueue then
           fInputQueue:MGetQueueSize()
         else
           0;
 end,
MResetConfig:
 func()
 begin
  fEndPoint.configOptions := fEndPointConfig.configOptions;
  fEndPoint.fIdleSpeed := (fEndPointConfig.fIODelay + 1)*100;
  fEndPoint.MOpenNetStack := func() nil;
  fEndPoint.MCloseNetStack := func() nil;

  if fEndPoint.configOptions then
    foreach option in fEndPoint.configOptions do
      if option.type and option.type = 'service and
           option.label and
           option.label = kCMSAppleTalkID then begin
         fEndPoint.MOpenNetStack :=func() OpenAppleTalk();
         fEndPoint.MCloseNetStack := func() CloseAppleTalk();
         break;
        end;
 end,
fOutputPhase: nil      // holds phase of output state machine during execution,
MResetQueues:
 func()
 begin
  fInputQueue:MReset();
  fOutputQueue:MReset();
  fWriteQueue:MReset();
 end,
powerOffScript:
 func(what)
 begin
  if what = 'okToPowerOff and fLlamaTalkState =
                      kLlamaTalkState_Disconnected then
    TRUE
  else
    NIL;
 end,
fEndPointConfigMNP:
 {
  fIODelay: 0,    // MNP has no RPC deadlock bug, so it can run full-speed comms
  configOptions:
    [
      { label:    kCMSMNPID,
        type:     'service,
        opCode:     opSetRequired},

      { label:    kCMOSerialIOParms,
        type:     'option,
        opCode:     opSetNegotiate,
        data:       { bps:      k38400bps,
                      dataBits: k8DataBits,
                      stopBits: k1StopBits,
                      parity:   kNoParity } },


      { label:    kCMOMNPAllocate,
        type:     'option,
        opCode:     opSetRequired,
        data:       kMNPDoAllocate }
```

```
      { label:    kCMOMNPCompression,
        type:     'option,
        opCode:     opSetRequired,
        data:       kMNPCompressionNone }
          // also works with kMNPCompressionV42bis (but that uses lots more memory)

      { label:    kCMOMNPDataRate,
        type:     'option,
        opCode:     opSetNegotiate,
        data:       k38400bps }
    ]
  }
MSetAddress:
  func(address)  // returns the address, or nil if view is not open
  begin
   if :MIsOpen() then
     if address then
       if IsFrame(address) then
         fEndPointAddress := EnsureInternal(address)
       else
         fEndPointAddress :=
           {
             label:     kCMARouteLabel,
             type:      'address,
             opCode:      opSetRequired,
             data: { addressType: kNamedAppleTalkAddress,
                     addressData:EnsureInternal(address), },
           }
     else fEndPointAddress := nil
   else nil;
  end,
MNewQueue:
  func(queueType)
  begin
   return kLlamaTalkQueueTemplate:MInstantiate(queueType);
  end,
MDisconnect:
  func()   // NOTE: this method may also be called from
             // :MEndPointExceptionHandler in which case SELF is the endpoint frame
  begin
   if fEndPoint
   and fEndPoint.fDeferredObj
   and not fEndPoint.fDeferredObj:KillIt() then
     :MSetLlamaTalkState(kLlamaTalkState_Disconnected);

   if fLlamaTalkState <> kLlamaTalkState_Connected
        and fLlamaTalkState <> kLlamaTalkState_Connecting
        then return;

   IF kDebugLlamaTalk THEN LT_PRINT("Disconnecting endpoint");

   local currentState := fLlamaTalkState;
   :MSetLlamaTalkState(kLlamaTalkState_Disconnecting);

   local slip;
   if slip := BuildContext(fDisconnectSlip) then
     slip:Open();

   fEndPoint.nextInputSpec := nil; // kill future input
   fEndPoint:SetInputSpec(nil);    // kill current input
   fEndPoint:Abort();              // kill pending input

   :MResetQueues();                    // blow away all queued data

   AddDelayedAction(MDisconnectCompProc, [fEndPoint,
        currentState, slip], 2500);      // we must do this as a delayed action!
   :Dirty();

   return nil;
  end,
fEndPoint:
  nil     // contains the "live" endpoint while we're connected, nil otherwise,
  ,
MSetLlamaTalkState:
  func(newState)
  begin
   local appBaseView := GetRoot().(kAppSymbol);
   appBaseView.fLlamaTalkState := newState;
   if call kViewIsOpenFunc with (appBaseView) then begin
        fLlamaTalk:?MLlamaTalk_StateChanged();
        if call kViewIsOpenFunc with (fLlamaTalk) then
          fLlamaTalk:Dirty();
        RefreshViews();
      end;
  end,
MAbortPendingTransactions:
  func()
  begin
   fInputQueue:MReset();
   fOutputQueue:MReset();
   return nil;
  end,
MToggle:
  func()
  begin
   if :MIsOpen() then
     if :MIsVisible() then :Hide()
     else :Show();
   return nil;
  end,
fOutputData:
  nil     // queue of outgoing transaction data items (transaction is sent in chunks)
  ,
MWrite:
  func(data)
  begin
   if fLlamaTalkState <> kLlamaTalkState_Connected then
     return;

   IF kDebugLlamaTalk THEN LT_PRINT("Enqueueing data...");

   local translation := nil;

   if IsInstance(data, 'string) then
     translation := call kLlamaTalkStuffCStringFunc with
                    (:MNewBinary(StrLen(data) + 1), 0, data)

   else if IsInstance(data, 'binary) then
     translation := data

   else begin
          IF kDebugLlamaTalk THEN LT_PRINT("Unsupported data
               class!!!  Cannot enqueue...");
        GetRoot():SysBeep();
     end;

   fWriteQueue:MEnQueue(translation);
     // Note :MEnQueue() won't enqueue a nil (this is good...)

   return translation;
  end,
declareSelf: 'fLlamaTalk,
viewSetupFormScript:
  func()
  begin
   if kDebugLlamaTalk then
     functions.LT_PRINT := func(data)   // This is VERY dangerous,
          // but since we only do it for a PRIVATE debug build...
          begin
            Print(data);
              // Sleep(10);
          end;

   self.fEndPointAddress := TotalClone(fEndPointAddress);
     // ADSP address frame must be in RAM so we can modify it
   self.fEndPointConfig := fEndPointConfigADSP;
     // ADSP is the default configuration for this endpoint proto
   self.fEndPoint :=
   // the endpoint frame must be in RAM so we can modify it
     {
       _proto:              protoEndpoint,
       _parent:               self,
       ExceptionHandler:      MEndPointExceptionHandler,
       FInputHeaderComponent: FInputHeaderComponent,
       FInputDataComponent:   Clone(FInputDataComponent),
            // shallow clone because top level slots must be modifiable
       FInputDataSkipper:   Clone(FInputDataSkipper),
            // shallow clone because top level slots must be modifiable
       configOptions:         nil,
            // these nil slots defined here to help reduce the number of
       fIdleSpeed:            nil,
            // frame map entries, their values are set in :MResetConfig()
       MOpenNetStack:         nil,
            // bug fix for endpoints that use network protocols
       MCloseNetStack:        nil,
            // bug fix for endpoints that use network protocols
       fDeferredObj:          nil,
```

```
            // holds a "magic frame" that allows us to cancel deferred actions(!)
    }
  self:MResetConfig();  // initialize the endpoint frame's configuration slots

  self.fInputQueue := :MNewQueue('FIFO);
  self.fOutputQueue := :MNewQueue('FIFO);
  self.fWriteQueue := :MNewQueue('FIFO);

  self.fOutputPhase := 0;
 end,
FInputDataSkipper:
 {
  inputForm:      'raw,
  byteCount:      0,
  fDataBytesToGo: 0,
  inputScript:    func(ep, data)
    begin
      IF kDebugLlamaTalk THEN LT_PRINT("FInputDataSkipper
            called...");
      fDataBytesToGo := fDataBytesToGo - byteCount;

      if fDataBytesToGo > 0 then begin
          byteCount := if fDataBytesToGo < 512 then
              fDataBytesToGo else 512;
          ep:SetInputSpec(ep.FInputDataSkipper);
        end
      else
        ep:SetInputSpec(ep.FInputHeaderComponent);
    end,
 }
MEndPointExceptionHandler:
 func(exception)
 begin
  IF kDebugLlamaTalk THEN LT_PRINT(
      "MEndPointExceptionHandler called!");

  if exception.data exists
  and exception.data <> -16005
      // exception generated by ep:Abort() while connected -- really not an error
  and exception.data <> -16013
      // exception generated by ep:Abort() during connect -- really not an error
  and fLlamaTalkState<>kLlamaTalkState_Disconnecting then
      begin
      if exception.data = -20003   // the other end has torn down its connection
      or exception.data = -16009 then
                  // wouldn't it be nice if all endpoints used the same error codes?
        if fIsQuietDisconnect then
            nil // avoid calling the error method if we're supposed to keep quiet about it
        else
          fLlamaTalk:?MLlamaTalk_StdError("The connection
              has closed unexpectedly.", exception.data)
      else
        fLlamaTalk:?MLlamaTalk_StdError("A communications
            error has occured.", exception.data);

      AddDeferredAction(func(context) context:MDisconnect(),
                                                [fLlamaTalk]);
    end;

  return true;
 end,
MSetQuietDisconnect:
 func(quiet)   // returns true or nil
 begin
  fIsQuietDisconnect := if quiet then true else nil;
 end,
fEndPointAddress:
 nil  // contains the connection address frame, or nil if not required
 ,
 fWriteQueue:
 nil  // queue of outgoing transaction data items (entire queue is a transaction)
,
MIsConnecting:
 func()
 begin
  return fLlamaTalkState = kLlamaTalkState_Connecting;
 end,
MConnectCompProc:  // this routine is called from MConnectAction SELF is the endpoint frame
    func(ep, err)
 begin
  if not err then begin
      :MSetLlamaTalkState(kLlamaTalkState_Connected);
      :SetInputSpec(FInputHeaderComponent);  // kick-off the receive
    end;
  else if err <> -10039 then begin
      fLlamaTalk:MDisconnect();
        IF kDebugLlamaTalk THEN
          LT_PRINT("ERROR in MConnect Connect,
            error code =" && NumberStr(err));
      fLlamaTalk:?MLlamaTalk_StdError("Could not
            connect.", err);
    end;

  return err;
 end,
FInputDataComponent:
 {
  inputForm:      'raw,
  byteCount:      0,
  fDataBytesToGo: 0,
  fDataIndex:     0,
  fData:          nil,
  inputScript:   func(ep, data)
    begin
      IF kDebugLlamaTalk THEN LT_PRINT(
          "FInputDataComponent called...");
      BinaryMunger(fData, fDataIndex, byteCount,
                                        data, 0, byteCount);
      fDataIndex := fDataIndex + byteCount;
      fDataBytesToGo := fDataBytesToGo - byteCount;

      if fDataBytesToGo > 0 then begin
          byteCount := if fDataBytesToGo < 512
              then fDataBytesToGo else 512;
          ep:SetInputSpec(ep.FInputDataComponent);
        end
      else begin
          ep._parent.fInputQueue:MEnQueue(fData);
          ep:SetInputSpec(ep.FInputHeaderComponent);
          ep._parent:SetUpIdle(100);
        end;
    end,
 }
viewHideScript:
 func()
 begin
  fBusyShapes := nil;
 end,
viewclass: 74,
fOutputQueue:
 nil  // each entry holds an outgoing transaction (a queue) until it can be processed
,
MConnectAction:
   // this routine is called normally from MConnect, but as a deferred action from MConnectAsync
   // therefore, we cannot depend on the context of SELF, so give everything the ep frame context

 func(ep)
 begin
    local err := kLlamaTalkError_EndpointInUse;
    ep:MOpenNetStack();
 try
    err := ep:Instantiate(ep, NIL)
 onexception |evt.ex| do
    begin
      if HasSlot(CurrentException(), 'error) then
        err := CurrentException().error;
      ep:MCloseNetStack();
      IF kDebugLlamaTalk THEN
          LT_PRINT("ERROR in MConnectAction Instantiate,
            error code =" && NumberStr(err));
            ep:?MLlamaTalk_StdError
            (kLlamaTalkMessage_EndpointInUse, err);
            ep:MSetLlamaTalkState
            (kLlamaTalkState_Disconnected);
      return;
    end;

  ep:MSetLlamaTalkState(kLlamaTalkState_Connecting);
  err := ep:Connect(ep._parent.fEndPointAddress, nil);
  ep:MConnectCompProc(ep, err);
 end,
debug: "LlamaTalk",
MAbortTransaction:
 func()
 begin
  IF kDebugLlamaTalk THEN
      LT_PRINT("Aborting current transaction...");
  return :MBeginTransaction();
```

```
    end,
  fInputQueue: nil, // holds incoming data until it can be processed
  viewShowScript:
    func()
    begin
      local r := :LocalBox();
      local s := MakeOval(0, 0, r.right - 1, r.bottom - 1);
      fBusyShapes :=
        [
          { fShape:     [ MakeShape(r) ], // used to erase the view
            fStyle:   { transferMode:modeCopy,
                        penPattern:   vfNone,
                        fillPattern:    vfFillWhite,        }},

          { fShape:     [ s,            // 'DISCONNECTED phase
                        MakeLine(0,0,r.right-1, r.bottom-1),
                        MakeLine(0,r.bottom-1, r.right-1,0)
                          ]
            fStyle:   { transferMode:modeCopy,
                        penSize:      1,
                        penPattern:   vfBlack,
                        fillPattern:    vfFillWhite,        }},

          { fShape:     [ s ]           // 'IDLE phase
            fStyle:   { transferMode:modeCopy,
                        penSize:      1,
                        penPattern:   vfBlack,
                        fillPattern:    vfFillWhite,        }},

          { fShape:     [ s ]           // 'INPUT phase
            fStyle:   { transferMode:modeCopy,
                        penSize:      1,
                        penPattern:   vfBlack,
                        fillPattern:    vfFillGray,         }},

          { fShape:     [ s ]           // 'OUTPUT phase
            fStyle:   { transferMode:modeCopy,
                        penSize:      1,
                        penPattern:   vfNone,
                        fillPattern:    vfFillBlack,        }},
        ]
    end,
  MGetState:
    func()
    begin
      local state := -1;

      if fLlamaTalkState = kLlamaTalkState_Connected then
        try
          state := fEndPoint:State();
        onexception |evt.ex| do
          state := kLlamaTalkError_IllegalOperation;

      return state;
    end,
  fIsQuietDisconnect: nil,
}
```

Back in File LlamaTalkMain.t

```
vLlamaTalk := /* child of vDali */
  {viewBounds: {left: 8, top: 141, right: 24, bottom: 157},
   _proto: LlamaTalk,
   debug: "vLlamaTalk"
  }; // View vLlamaTalk is declared to vDali

vMessage := /* child of vDali */
  {text: "Ready for connect\u2026",
   viewBounds: {left: 9, top: 25, right: 229, bottom: 125},
   viewJustify: 0,
   viewFormat: 524624,
   _proto: protoStaticText,
   debug: "vMessage"
  }; // View vMessage is declared to vDali

vStatus := /* child of vDali */
  {text: "",
   viewBounds: {left: 32, top: 136, right: 224, bottom: 168},
   viewIdleScript:
     func()
     begin
       SetValue (vStatus, 'text,"In =" &&
         NumberStr(vLlamaTalk:MCountPendingInputTransactions())
         & "     " /* six blank spaces */ & "Out =" &&
         NumberStr(vLlamaTalk:MCountPendingOutputTransactions())
         & "\n" & "State =" && NumberStr(vLlamaTalk:MGetState())
         & "   " & (if fLlamaTalkState then
             NumberStr(fLlamaTalkState) else "nil") & "      " &
           "Ticks =" && NumberStr(Ticks()) );
           1000;  // 'idle one second': return the number of milliseconds
                  //  to wait before calling IdelScript again
     end,
   viewSetupDoneScript:
     func()
     begin
       :SetUpIdle(100);
     end,
   viewJustify: 0,
   _proto: protoStaticText,
   debug: "vStatus"
  }; // View vStatus is declared to vDali

vConnect := /* child of vDali */
  {text: "",
   buttonClickScript:
     func()
     begin
       if vLlamaTalk:MIsConnected() then
         :DoDisconnect()
       else
         :DoConnect();
       :DoUpdateButtons();
     end,
   viewBounds: {left: 122, top: 210, right: 222, bottom: 230},
   _proto: protoTextButton,
   debug: "vConnect"
  }; // View vConnect is declared to vDali

vOpenClose := /* child of vDali */
  {text: "",
   buttonClickScript:
     func()
     begin
       vLlamaTalk:Toggle();
       :DoUpdateButtons();
     end,
   viewBounds: {left: 122, top: 242, right: 166, bottom: 262},
   _proto: protoTextButton,
   debug: "vOpenClose"
  }; // View vOpenClose is declared to vDali

vShowHide := /* child of vDali */
  {text: "",
   buttonClickScript:
     func()
     begin
       vLlamaTalk:MToggle();
       :DoUpdateButtons();
     end;,
   viewBounds: {left: 178, top: 242, right: 222, bottom: 262},
   viewFlags: 515,
   _proto: protoTextButton,
   debug: "vShowHide"
  }; // View vShowHide is declared to vDali

vSendPings := /* child of vDali */
  {text: "Start Ping Test",
   buttonClickScript:
     func()
     begin
       if fIsActiveIdle then begin
            fIsActiveIdle := nil;
            SetValue(self, 'text, "Start Ping Test");
            :SetUpIdle(0);
          end
       else begin
            fIsActiveIdle := true;
            SetValue(self, 'text, "Stop Ping Test");
            :SetUpIdle(100);
          end;
       end,
   viewBounds: {left: 122, top: 274, right: 222, bottom: 294},
   viewIdleScript:
     func()
     begin
       if vLlamaTalk:MCountPendingOutputTransactions() < 1 then
            // let's try to not explode our Newton, okay?
         if vLlamaTalk:MBeginTransaction() then begin
              vLlamaTalk:MWrite("How very like the future this place
```

```
might be.  It is a tiny world just big enough to support the
chemicals of one knowledge worker.  I feel a wave of loneliness
as I head back down.  Am I going too fast?");
            vLlamaTalk:MWrite("I plunge right on in through the
office door and into the bottomless sea below.  Suddenly I can't
remember how to stop.  Turn around.  Look.  Point behind myself.
Do I have to turn around and point?  I flip into a burning fit.");
            vLlamaTalk:MEndTransaction();
          end
        else begin
            fIsActiveIdle := nil;
            SetValue(self, 'text, "Start Ping Test");
            return nil;
          end;

      return 1000; // idle rate at which to send, in miliseconds, or nil to turn off idle handler
    end,
   fIsActiveIdle: nil,
   _proto: protoTextButton,
   debug: "vSendPings"
  }; // View vSendPings is declared to vDali

vProtocolLabel := /* child of vDali */
  {text: "Transport Protocol:",
   viewBounds: {left: 8, top: 176, right: 112, bottom: 192},
   viewJustify: 8388609,
   _proto: protoStaticText,
   debug: "vProtocolLabel"
  }; // View vProtocolLabel is declared to vDali

vProtocol := /* child of vDali */
  {viewBounds: {left: 120, top: 178, right: 216, bottom: 192},
   clusterValue: 1,
   _proto: protoRadioCluster,
   debug: "vProtocol"
  }; // View vProtocol is declared to vDali

vADSP := /* child of vProtocol */
  {buttonValue: 1,
   viewBounds: {left: 0, top: -5, right: 48, bottom: 11},
   text: "ADSP",
   _proto: protoRadioButton,
   debug: "vADSP"
  }; // View vADSP is declared to vDali

vMNP := /* child of vProtocol */
  {buttonValue: 2,
   viewBounds: {left: 48, top: -5, right: 96, bottom: 11},
   text: "MNP",
   _proto: protoRadioButton,
   debug: "vMNP"
  }; // View vMNP is declared to vDali

vS := /* child of vDali */
  {text: "Str",
   buttonClickScript:
    func()
    begin
     vLlamaTalk:MBeginTransaction();
     vLlamaTalk:MWriteObject("Hello");
     vLlamaTalk:MEndTransaction();
    end,
   viewBounds: {left: 66, top: 234, right: 94, bottom: 246},
   _proto: protoTextButton,
   debug: "vS"
  }; // View vS is declared to vDali

vC := /* child of vDali */
  {text: "Char",
   buttonClickScript:
    func()
    begin
     vLlamaTalk:MBeginTransaction();
     vLlamaTalk:MWriteObject($A);
     vLlamaTalk:MEndTransaction();
    end,
   viewBounds: {left: 26, top: 234, right: 54, bottom: 246},
   _proto: protoTextButton,
   debug: "vC"
  }; // View vC is declared to vDali

vT := /* child of vDali */
  {text: "True",
   buttonClickScript:
    func()
    begin
     vLlamaTalk:MBeginTransaction();
     vLlamaTalk:MWriteObject(true);
     vLlamaTalk:MEndTransaction();
    end,
   viewBounds: {left: 66, top: 210, right: 94, bottom: 222},
   _proto: protoTextButton,
   debug: "vT"
  }; // View vT is declared to vDali

vI := /* child of vDali */
  {text: "Int",
   buttonClickScript:
    func()
    begin
     vLlamaTalk:MBeginTransaction();
     vLlamaTalk:MWriteObject(1234567);
     vLlamaTalk:MEndTransaction();
    end,
   viewBounds: {left: 26, top: 258, right: 54, bottom: 270},
   _proto: protoTextButton,
   debug: "vI"
  }; // View vI is declared to vDali

vR := /* child of vDali */
  {text: "Real",
   buttonClickScript:
    func()
    begin
     vLlamaTalk:MBeginTransaction();
     vLlamaTalk:MWriteObject(1234567.89);
     vLlamaTalk:MEndTransaction();
    end,
   viewBounds: {left: 66, top: 258, right: 94, bottom: 270},
   _proto: protoTextButton,
   debug: "vR"
  }; // View vR is declared to vDali

vN := /* child of vDali */
  {text: "Nil",
   buttonClickScript:
    func()
    begin
     vLlamaTalk:MBeginTransaction();
     vLlamaTalk:MWriteObject(nil);
     vLlamaTalk:MEndTransaction();
    end,
   viewBounds: {left: 26, top: 210, right: 54, bottom: 222},
   _proto: protoTextButton,
   debug: "vN"
  }; // View vN is declared to vDali

vB := /* child of vDali */
  {text: "Binary",
   buttonClickScript:
    func()
    begin
     vLlamaTalk:MBeginTransaction();
     local b := SetLength(SetClass(Clone(""), 'binary), 4);
     StuffLong(b, 0, 1234567);
     vLlamaTalk:MWriteObject(b);
     vLlamaTalk:MEndTransaction();
    end,
   viewBounds: {left: 34, top: 282, right: 86, bottom: 294},
   _proto: protoTextButton,
   debug: "vB"
  }; // View vB is declared to vDali

vAsync := /* child of vDali */
  {indent: 4,
   text: "Connect Asynchronously",
   viewBounds: {left: 112, top: 192, right: 234, bottom: 208},
   viewSetupFormScript:
    func()
    begin
     self.indent := self.indent + StrWidth(self.text);
     inherited:?viewSetupFormScript();
    end,
   viewValue: nil,
   _proto: protoRCheckbox,
   debug: "vAsync"
  }; // View vAsync is declared to vDali

// After Script for "vDali"
thisView := vDali;
```

NTJ

continued from page 1

Welcome to the Desktop Integration Libraries

(API) and the documentation for the platform-specific desktop communications code. Developers could let Newton devices communicate with other devices using either the serial port, modem, infrared transmitter, or AppleTalk/ADSP connections. With the Newton Toolkit (NTK) application, developers can design Newton applications to talk to the outside world, but that does not mean that it is easy for desktop applications to talk to a Newton device.

Differing communications APIs between Windows and MacOS, as well as complex communication APIs on each platform force a steep learning curve on many desktop application developers who hope for Newton connectivity. Some of these solutions are still incomplete without custom error-correction, like that required for infrared connections to a desktop receiver. To compound the problem, objects in the unified data model of the NewtonScript language – called "frames" – are often too richly defined to translate easily to the rigid memory structures found in traditional programming languages on the desktop. Desktop developers interested in a robust cross-platform solution must implement multiple types of communication mechanisms for both MacOS and Windows; developers must also deal with the perils of translating frames between the Newton world and the desktop world.

The Newton team has thus created libraries for both MacOS and Windows computers that will enable third-party developers to more quickly integrate Newton connectivity into their desktop applications . These libraries are called the Desktop Integration Libraries.

About the Newton Desktop Integration Libraries

The Newton Desktop Integration Libraries (DILs) come in several layers, with each layer building upon the functionality of the other layers. These are:

- the communications layer
  The Communications DILs (the CDILs)
- the Newton object layer
  The Frames DILs (the FDILs)
  – This layer contains the CDILs
- the synchronization & package layer
  The Protocol DILs (the PDILs)
  – This layer contains the FDILs and CDILs

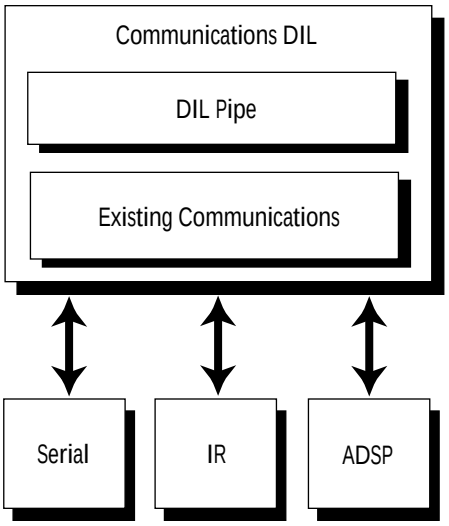
The communications layer is the basic layer of the Desktop Integration Libraries. The accompanying functions and classes allow a desktop computer and Newton to easily configure a transport-independent connection. On top of that connection, the frames layer translates complex Newton frames to the desktop world. The highest layer, the Protocol DIL, consists of libraries intended to reproduce the essential parts of the "connection protocol" used by the Newton Connection Kit . This layer currently requires that the desktop-stored data be stored on disk in a format similar to the NCK synchronization files, so that synchronization can proceed almost automatically.

All of these libraries are designed to work on the MacOS platform in the major C and C++ development environments and on the Windows platforms via Dynamic Linked Libraries (DLLs). Note that the calling conventions in C and C++ versions differ, with the C versions containing an extra argument representing the target object, encapsulated with the behavior in the C++ version. In this article, examples will be shown using the C++ syntax.

Making the Connection

The Communication DILs provide a simple cross-platform transport-independent communications API on the desktop. Current connection types implemented are serial, modem/MNP, AppleTalk/ADSP, and infrared. (Straight serial and ADSP connections are not yet available for Windows DILs.) When other local or network transports are supported on the Newton, all that will be required to support them on the desktop is a new version of the DILs and modifications to the desktop user interface for choosing the new connection type.

From the Newton world, using the Communications DILs is simply a matter of configuring a standard Newton communications endpoint. You must set the endpoint's `configOptions` appropriately to determine the connection type. For instance, this could be serial, serial-MNP, modem, infrared, or AppleTalk. All the standard read and write functions are supported through the Newton endpoint.



Platform Communications Tools

On the desktop side, you communicate to the Newton device through a connection object called a "pipe". A pipe is a virtual bi-directional stream of data that automatically handles data conversions such as byte-swapping (a touchy issue for cross-platform integer manipulation), ASCII-to-Unicode conversion, low-level handshaking, and encryption. Note that encryption algorithms are not built in to the CDIL libraries. The hooks are available to the CDIL developer for implementing encryption on the data stream itself, so that the higher level CDIL, FDIL, and PDIL interfaces work independent

of encryption.

You will initialize the CDILs once in your application via `CDInitCDILs()` and then obtain a virtual pipe via `CreateCDILObject()`. This pipe is not yet tied to a particular connection type or port; to tell the CDILs how to find the Newton, use the `CDPipeInit` routine. Here is an example of how to use this function to initialize a pipe with a standard serial connection.

```
fErr = CDInitCDILs();
if (fErr) return(fErr);
ourPipe = CreateCDILObject();
fErr = ourPipe->CDPipeInit( "Serial","", "Baud 38400
  dataBits 8 Parity None Port ", Port);
```

Note that the connection-type-specific arguments are presented in a string so that each connection type can define as many arguments for initialization as necessary. In this example, the `Port` variable is used to specify which serial port the user has selected. If `CDPipeInit` returns no error, you should complete the pipe connection using `PipeListen` and `PipeAccept`, which are required, but are only interesting for transports like ADSP and MNP that demand bilateral arbitration of the connection before data can flow. Note that you must define your own time-outs for completing the connection.

Once connected, you can send data using the straightforward `CDPipeRead` and `CDPipeWrite` functions. If you need to know the details of the pipe, functions are available to check the pipe state, to return the number of bytes in the pipe, and to flush the outgoing data from the pipe. Even if you are using the higher level DILs, you will still use the CDILs to instantiate the pipe, to send simple data, and to receive simple data. Note that from the Newton side, normal endpoint states and the `endpoint:Output()` function are used to send simple data and commands, perhaps as a prelude to frame sending/receiving. For instance, your desktop code might initiate its conversation like:

```
fErr =  ourPipe->CDPipeWrite("LLMA\4", &len, true);
if (fErr) return(kNoDataSent);
fErr = ourPipe->CDPipeRead(str, &length, &eom, 0, 0);
if (!fErr && strcmp(str, "OK") == 0)
  StartGettingMyData(ourPipe);
```

Sending Newton Frames

Desktop developers who want to communicate with Newton devices must eventually deal with the complexity of the Newton data model. While structure design and size in C are specified at compile time, the dynamic NewtonScript language can accommodate complex objects (frames) whose data and field names (slots) are created at run time. For instance, a Newton application might receive the following object without prior knowledge of its structure:

```
    // 1. frame within a frame
    // 2. array of symbols
    // 3. simple string
    // 4. integer
t :=  {name:  { first: "J. Christopher",  last: "Bell", },
    neighbors: ['Jim, 'Bruce],
    company: "Apple Computer"
    RDbuilding: 5,
    }
```

This level of structure is common for a NewtonScript object. Even if a desktop application is designed to work only with built-in Newton applications, the desktop application must expect the unexpected – particularly user- or developer-created objects stored in new slots. For this reason, the Frames DILs are designed to accommodate data that is not bound to a predictable structure. If the incoming data is not predictable by the desktop application, it is called "unbound data".

For some applications, all data is in a predictable format that is stored on the desktop side as a simple C struct and stored on the Newton in simple frames like:

```
e := {height: 345, notes: "Reg"};
```

For this class of desktop applications, the primary concern is high-performance data transfer to and from the Newton via its native C structs. In these applications, developers probably want to specify the mapping between the Newton world and the C world, because this will boost performance. For this type of application, high performance is achieved by specifying the structure as "bound data".

When describing Newton data on the desktop, several basic distinctions apply. The basic class, `DILObject`, is used to represent basic object behavior. The simplest instances of this class are objects defined on the Newton in 32 bits, objects that are known as "immediates". These include `nil`, `true`, single characters with automatic ASCII-Unicode translation, and NewtonScript's signed 30-bit integers. Other classes that build on this behavior are `DILArray` and `DILFrame`, both of which implement a list whose elements map to different desktop memory locations. Based on top of `DILObject` are binary objects and strings; strings use automatic Unicode-ASCII conversion if specified as bound data.

Bound Data – The Basics

Here's an example of how to bind a slot in a frame so that you can send Newton frames or receive frames as fast as possible. For each slot in the frame, which corresponds to a C structure, you can create a binding specifying the type, size, and the corresponding memory location. For instance, a desktop application could specify that it is expecting a slot called `height` that should contain an integer; this integer should be stored in memory location `&myStruct.height`. To do this using the FDILs, you would create an `DILFrame` (in C++) using the code `myFrame = new DILFrame;`. Binding is done through the `FDbindSlot` method using code like:

```
// tell the FDILs how to map the Newton slot 'height.
//  args = slot name, memAddress,  object type, length (not used
//         for immediates like integers)
myFrame->FDbindSlot("height", &myStruct.height, kDILInteger, 0)
myFrame->FDbindSlot("notes", &myStruct.note, kDILString, 256)
```

Binding: any type, anywhere

Note that in the above example, the target frame is flat in the sense that it contains no imbedded structures. You can send or receive much more complex objects if you know their structure and data types. For instance, if you wanted to bind the data in the Newton frame `{name: {...}, ...}` – a frame within a frame – you could bind it with code like:

```
outerFrame = new DILFrame;
subFrame = new DILFrame;
  // define the sub frame slots
subFrame ->FDbindSlot( [...] );

// the subframe is within outerFrame
outerFrame->FDbindSlot("name",
   subFrame , kDILFrame, 0);
```

Note that this structure-within-a-structure ability also applies to arrays. To define an array of three elements, create a `DILArray` object and use `FDbindSlot` to bind three "slots" (in this case, array positions) within it. For example, let's use the name example and bind all its data for highest performance when receiving:

```
/* prepare to receive a frame like:
t := {name: { first: "J. Christopher", last: "Bell"},
     neighbors: ['Jim, 'Bruce],
     company: "Apple Computer"
     RDbuilding: 5,
     }
*/
mainF = new DILFrame;
subF = new DILFrame;
subA = new DILArray;

// Bind the "substructures" (1 subframe, 1 subarray)
subF ->FDbindSlot("first", &Fname, kDILString, len);
subF ->FDbindSlot("last", &Lname, kDILString, len2);
subA ->FDbindSlot(NULL,&symString1, kDILSymbol, 0);
subA ->FDbindSlot(NULL,&symString2, kDILSymbol, 0);

// Bind slots in outer frame
mainF->FDbindSlot("name", subF, kDILFrame, 0);
mainF->FDbindSlot("neighbors", subA, kDILArray, 0);
mainF->FDbindSlot("company",  &comp, kDILString, 0);
mainF->FDbindSlot("RDbuilding", &b_num, kDILInteger, 0);
```

Once you have defined your bindings, you can send and receive data using `FDPut` and `FDGet` with code like the following:

```
outFrame->FDput(myCDILPipe);
myFrame->FDget (myCDILPipe,
  0 /* for byte swapping */,
  0 /* for char encoding */,
  15 /* for timeouts */,
  NULL /* for a completionHook */);
```

Note that the argument for the `completionHook` callback function permits you to implement your frame reading asynchronously or synchronously, depending on the value of that argument.

You might be asking, "But what if I don't know the exact structure of the data beforehand?" Since NewtonScript is so dynamic, the FDILs are designed to handle this without much trouble. After using `FDGet` to receive a frame, calling `myFrame->FDGetUnboundList()` will return a tree structure that you can navigate to obtain information about unexpected slots, array data, subframes, or any data not predictable enough to bind explicitly. Note that you can obtain this tree of information whether or not any slots are bound. If you bind some slots, but extra slots are received, they are banished to the unbound list – which you are not required to check. On the other hand, you could decide to bind no data, and then recursively investigate all incoming data solely through the unbound list.

You can traverse this tree via references like `arrayOrFrame->children[i]` and you can get data from entries by checking their `var` and `varType` fields. Entries in this structure are tagged with a type; if their `varType` is `kDILFrame` or `kDILArray`, you will probably want to recursively check their children in the tree, copying the contents of the frame or array. For other objects, the data is stored in each entry's `var` field. For instance, the following code looks for a string to copy from the current entry:

```
if (entry->varType == kDILString)
  strcpy(destString, entry->var);
```

A sample included with the DILs shows how to recursively print out one of these trees of unbound data. You can capture and store all this information if you wish or you can disregard it, depending on the type of application. After you are done with the unbound list, call `FDFreeUnboundList()` to dispose of the structure before receiving the next frame.

On the Newton side, the transfer of frames is relatively simple. After establishing the Newton endpoint, use endpoint `inputForms` with the `'frame` value to receive frames and the `endpoint:OutputFrame()` method to send frames from the Newton. This might seem surprisingly easy in comparison to the desktop version. Most of the energy expended on the desktop side is to manipulate the robust data structures and the run-time data tagging that are taken for granted in NewtonScript.

Synchronization and Packages

The Protocol DIL (PDIL) libraries are a higher level of connectivity between Newton devices and the desktop. One feature that appeals to desktop application developers wishing to manipulate built-in soups is the ability to connect to the desktop via the Connection icon in the Extras Drawer; there is no need for an additional Extras Drawer icon or a Newton installer. This is because the PDILs can permit your desktop application to speak the protocol similar to the one used by the Newton Connection Kit and the Newton Package Installer.

Although package installation is an option in both of those applications, automatic soup synchronization facilities in the PDILs are appealing to many desktop developers. To do this, the PDILs emulate the basics of the Newton Object Store on the local desktop file system. This facilitates new applications wishing to maintain API consistency across platforms, but is less likely to be used for new applications because of the dependency on the PDIL's native "store" file format for "soup" storage. We will not go into detail in this article on the API for the Protocol DILs, but it is an extension of the FDILs and will be released soon after the FDILs. As part of these libraries, desktop developers will receive suites of data type translation functions.

Powerful and Easy to Port

With the APIs described above, you have the basic knowledge of how to:

- Set up virtual connections between a desktop application and a Newton.
- Port your communications code easily between MacOS or Windows.
- Enable other connection types easily after writing code for one of them. (current connection types are serial, modem/MNP, infrared, and AppleTalk)
- Send and receive simple data over these basic connections.
- Send and receive complex NewtonScript frames from a desktop application.

These are the basic skills necessary to take a desktop application and turn it into an integrated desktop solution using the DILs. Whether you plan to upload a single Newton user's data, or to distribute marketing data to a team of mobile Newtons, the Desktop Integration Libraries open the doors to high-performance desktop-to-Newton applications. Most important, you can port your communications code to new platforms and new connection types easily. In addition to this flexibility, the robust DIL APIs will lighten the burden of learning the MacOS CommToolbox and Windows communication tools, and let you focus instead on the data you want to send and receive. Good luck with your desktop-to-Newton solution and welcome to the DILs!

NTJ