Newton 2.x OS Q&A's

This document addresses Newton 2.x OS development issues that are not available in the currently printed documentation. Please note that this information is subject to change as the Newton technology and development environment evolve. If you have suggestions on how to improve the Newton DTS Q&As (or want the Q&As in additional formats), please email J. Christopher Bell at llama@apple.com. For the most recent version of the Q&As on the World Wide Web, check the URL:
http://devworld.apple.com/dev/newton/techinfo/NewtonQA.html

TABLE OF CONTENTS:

Desktop Connectivity (DILs)

Digital Books

Drawing and Graphics

Endpoints & Comm Tools

Hardware & OS

Localization

Miscellaneous

NewtApp

Newton C++ Tools

Newton ToolKit

NewtonScript

Pickers, Popups and Overviews

Recognition

Routing

Sound

Stationery

System Services, Find, and Filing

Text and Ink Input and Display

Transports

Utility Functions

Views

Application Design

How to Prevent an Application from Becoming a Backdrop (8/8/93)

Q: Is there a way an application can request not to be a backdrop application?

A: Yes, adding a non-`nil` `'noBackdrop` slot to the base view will stop an application from becoming a backdrop application.

The Newton Device Reboots When Turned On (8/9/93)

Q: My application has a really tight loop which can take more than a dozen seconds to finish. If the user turns off the Newton while my application is running, nothing happens at first, but finally the Newton turns off. The Newton device reboots when turned on. Why?

A: The reboot is happening because of a design goal. When the Newton OS learns the user wants to do a power off, the OS checks with the running application and says "Please get ready to shut down." If there is no response within about ten seconds the OS assumes that the process could be in an infinite loop. Since the user wants to turn off the Newton device, the OS terminates the application. When powering back up, there is no real clean state to return to, so the OS causes a reboot.

To work around this problem break up long processes so they can run in an `viewIdleScript`. In general, applications should release the CPU now and then so the OS can do clean up operations.

Optimizing Base View Functions (9/15/93)

Q: I've got this really tight loop that executes a "global" function. The function isn't really global, it's defined in my base view and the lookup time to find it slows down my code. Is there anything I can do to optimize it?

A: If the function does not use inheritance or "`self`", you can speed things up by doing the lookup explicitly once before executing the loop, and using the call statement to execute the function within the body of the loop.

Here's some code you can try inside the Inspector window:

```
f1 := {myFn: func() 42};
f2 := {_parent: f1};
f3 := {_parent: f2};
f4 := {_parent: f3};
f5 := {_parent: f4};
f5.test1 := func ()
   for i:=1 to 2000 do call myFn with ();
f5.test2 := func() begin
   local fn := myFn;
   for i:=1 to 2000 do call fn with ();
   end
```

```
/* executes with a noticeable delay */
f5:test1();

/* executes noticeably faster */
f5:test2();
```

Use this technique only for functions that don't use inheritance or the self keyword.

Note for MacOS programmers: this trick is analogous to the MacOS programming technique of using `GetTrapAddress` to get a trap's real address and calling it directly to avoid the overhead of trap dispatch.

Code Optimization (9/15/93)

Q: Does the compiler in the Newton Toolkit reorder expressions or fold floating point constants? Can the order of evaluation be forced (as with ANSI C)?

A: The current version of the compiler doesn't do any serious optimization, such as eliminating subexpressions, or reordering functions; however, this may change in future products. (Note: NTK 1.6 added constant folding, so for example `2+3` will be replaced with `5` by the compiler.) In the meantime, you need to write your code as clearly as possible without relying too heavily on the ordering of functions inside expressions.

The current version of the NTK compiler dead-strips conditional statements from your application code if the boolean expression is a simple constant. This feature allows you to compile your code conditionally.

For example, if you define a `kDebugMode` constant in your project and have in your application a statement conditioned by the value of `kDebugMode`, the NTK compiler removes the entire if/then statement from your application code when the value of `kDebugMode` is NIL.

```
constant kDebugMode := true;     // define in Project Data
if kDebugMode then Print(...);   // in application code
```

When you change the value of the `kDebugMode` constant to NIL, then the compiler strips out the entire if/then statement.

Global Name Scope (6/7/94)

Q: What is the scope of global variables and functions?

A: In NewtonScript, global functions and variables are true globals. This means that if you create global functions and global variables, you might get name clashes with other possible globals. As this system is dynamic, you can't do any pre-testing of existing global names.

Here are two recommended solutions in order to avoid name space problems:

Use your signature in any slot you create that is outside of the domain of your own application.

Unless you really want a true global function or variable, place the variable or function inside your base view template. You are actually able to call this function or access this variable from other applications, because the base view is declared to the root level.

If you really need to access the function or variable from a view that is not a descendent of your base view (like a floater that is a child of the root view), you might do something like:

```
if getroot().|MyBaseView:MySIG| then
   begin
      getroot().|MyBaseView:MySIG|:TestThisView();
      local s := getroot().|MyBaseView:MySIG|.BlahSize;
   end;
```

How to Prevent an Application From Opening (6/9/94)

Q: I do not want my application to open sometimes, for example because the screen size is too small, or because the Newton OS version is wrong. What's the best way to prevent it?

A: Check for whatever constraints or requirements you need early, if not in the `installScript`, then in the `viewSetupFormScript` for the application's base view. In your case, you can do some math on the frame returned from `GetAppParams` to see if the screen is large enough to support your application.

If you do not want the application to open, do the following:

- Call `Notify` to tell the user why your application cannot run.
- Set the base view's `viewBounds` so it does not appear, use `RelBounds(-10, -10, 0, 0)` so the view will be off-screen.
- Possibly set (and check) a flag so expensive startup things do not happen.
- Possibly set the base view's `viewChildren` and `stepChildren` slots to NIL.
- call `AddDeferredSend(self, 'Close, nil)` to close the view.

How to Create a Polite Backdrop Application (1/19/96)

Q: How do I get backdrop behavior in my application?

A: Backdrop behavior is given to you for free. If your applications close box is based on `protoCloseBox` or `protoLargeCloseBox` then your close box will automatically hide itself if your application is the backdrop application. If you also use `newtStatusBar` as your status bar proto, the appropriate buttons will shift to fill the gap left by the missing close box. Note that you do not have to use the NewtApp framework to use the `newtStatusBar` proto.

The system will automatically override the `Close` and `Hide` methods so your application cannot be closed.

If you need to know which application is the backdrop application, you can find the appSymbol of the current backdrop app with `GetUserConfig('blessedApp)`.

Here are some tips on being a polite backdrop application:
• Your application should be full-screen. (Set "Styles" as the backdrop to see why.)
• A polite backdrop application will also add the registered auxiliary buttons to its status bar. See the "Using Auxiliary Buttons" in the Newton Programmers Guide (Chapter 18.)

How to Respond to Changes From a Keyboard (2/6/96)

Q: I open a custom keyboard to edit my view. How can I tell that the keyboard has been closed so that I can process the potentially modified contents of the view?

A: The `viewChangedScript` for the view will be called each time the user does something to modify the view. For keyboards, this means the script is called each time the user taps a key. This is the only notification that is provided to indicate the view contents have changed.

There are no hooks you can use to tell you when standard keyboards have closed. If you implement your own keyboard, you could provide a `viewQuitScript` or other custom code to explicitly notify the target that the keyboard is going away, but we do not recommend this. (There may be a hardware keyboard attached, a system keyboard may be open, or the user may be writing into your view. It is a mistake to assume that the only way to modify your view is through your own keyboard.)

If the processing you need to do is lengthy and would interfere with normal typing on the keyboard, you can arrange it so the processing won't start for a few seconds. This usually gives the user time to type another key, which can then further delay the processing.

To make this "watchdog timer" happen, use the idle mechanism as your timer. Put the code to process the changes in the `viewIdleScript` (or call it from the `viewIdleScript`.) In the `viewChangedScript`, if the `'text` slot has changed, use `:SetupIdle(<delay>)` to arrange for the `viewIdleScript` to be called in a little while.

If `:SetupIdle(<delay>)` happens again before the first delay goes by (perhaps because the user typed another key,) the idle script will be called after the new delay. The older one is ignored. `SetupIdle` resets the timer each time it's called.

Don't forget to have the `viewIdleScript` return `NIL` so it won't be called repeatedly.

How to Test Your Application (2/7/96)

Q: Before I ship my application, what should I test?

A: Although there is no complete answer, the following is a quick outline of things that should be tested to ensure compatibility with the Newton OS. Items that are OS or Locale specific are noted. Also note that this list only covers current Apple MessagePad devices.

This is something to help you think of other areas to test. Covering the areas in this list should improve the stability of your application, but is not guaranteed to make it stable and fool-proof.

This list does not cover the functionality of the application itself. That is, it is not a test plan for your application.

1. Versions (Latest supported system updates)
See Current versions of MessagePad devices in the Misc. Q&A

2. Basic Functional Testing
2.1. Launch and use app from internal RAM, memory card, locked memory card, in rotated mode

3. Data Manipulation
3.1. Create and store data in internal RAM
3.2. Create and store data to memory card
3.3. Delete data from internal RAM
3.4. Delete data from memory card
3.5. Move data from internal RAM to memory card and vice versa
3.6. Duplicate data
3.7. Find data with app frontmost
3.8. Find data in app using Find All from paperroll
3.9. Find data in all editable fields
3.10. Check the app name in the Find slip when "Selected" is checked, and check that the app name is correct for the radio button in the Find slip
3.11. If the app implements custom find, make sure other types of find (selected and everywhere) still work.
3.12. Select and Copy data to and from clipboard
3.13. Backup to memory card and restore to different Newton device. Verify that data is intact.
3.14. Backup via NBU and restore to different Newton device. Verify that data is intact.
3.15. File data into folders (if supported.)

4. Communications
4.1. Print data to serial printer and network printer
4.2. Fax data
4.3. Beam data to another 2.x Newton device
4.4. Beam data to a 1.x Newton
4.5. Backup and restore data and app to memory card
4.6. Backup and restore data and app with NBU

5. Exception Testing (all of the following should cause exceptions)
5.1. Create new data to locked memory card
5.2. Delete data from locked memory card
5.3. Move data from internal memory to locked card
5.4. Beam data to a Newton device that does not have the expected application
5.5. With application running from memory card, unlock card with application open.
5.6. With application installed on memory card, unlock card with application closed.
5.7. Install application on memory card, run application, create data, close application, remove memory card.
5.8. Turn power off while application is running (PowerOff handler?)
5.9. Attempt to create new data with store memory full.
5.10. Run application with low frames heap (us HeapShow to reserve memory)
5.11. If appropriate, run application with low system heap.

6. Misc.

6.1. Does application work if soup is entirely deleted from Storage folder in Extras?
6.2. Delete application. Does any part stay behind? (icons? menus? etc.)
6.3. Check store memory and frames heap, install application, check store memory and frames heap. Do this several times and check for consistency
6.4. Do 6.3. and also check store and frames memory after removing application. Is all/most of the memory restored?
6.5. Check frames heap. Launch & use application. Check heap. Close application. Check heap.
6.6. Does the application add anything to the Preferences App?
6.7. Does the application add Prefs and Help to the "i" icon?
6.8. Does the application add anything to Assist, How Do I?
6.9. Launch with pager card installed
6.10. Check layout issues on MP100 vs. MP110 screen sizes (if application runs in 1.x.)
6.11. If multiple applications are bundled together, open all at the same time, check to see that the applications together aren't using too much frames heap.
6.12. Open, use, and close the application many times. Check frames heap afterward to check for leaks.
6.13. If application has multiple components and components can be removed separately, verify that application does the right thing when components are missing.
6.14. Test application immediately after cold resets and warm resets.

7. Compatibility
7.1. After application is installed and run, do the built-in applications work:
Names, Dates, To Do List, Connection, InBox, OutBox, Calls, Calculator, Formulas, Time Zones, Clock, Styles, Help, Prefs, Owner Info, Setup, Writing Practice.
7.2. If the application can be the backdrop (this is the default case)
7.2.1 Do the built-in applications continue to work? The list is as in 7.1. and Extras.
7.2.2 Do printing and faxing work?
7.2.3 Run through the other tests in this document with your application as backdrop.
7.3. If the application can operate in the rotated mode
7.3.1. Perform all tests with the application in rotated mode as well.
7.3.2. Check that screen layouts look correct.

How To Make An Application the Backdrop (8/23/96)

Q: Is there a way to programmatically change the backdrop application?

A: Yes, but only if the "new" backdrop application is one of your applications. Do not set another application to be the backdrop application. `GetRoot():BlessApp(appSymbol)` will close the current backdrop application and open the new backdrop application if necessary. Note that `appSymbol` must be a valid application symbol of a current installed and active application. `BlessApp` does NOT verify that an application can become the backdrop (for instance, it doesn't check the `'noBackdrop` flag for an application). For this reason, `BlessApp` must only be used on your own applications. See the Newton DTS Q&A, "How to Prevent an Application from Becoming a Backdrop" for more information about the `'noBackdrop` flag.

Do not call `BlessApp` from within a part's `installScript` or `removeScript`. If you want to do something like this, use a delayed call to use `BlessApp`.

Children of Exported Protos Are Missing (9/20/96)

Q: I have a template that is based on a user proto imported by my package. When the view based on my template opens, none of the exported proto's children show up. What is going on?

A: When you create children of templates in Newton Toolkit, they are collected in the `stepChildren` slot of the base view of the template file. If both the exported and importing template have children, they will both have a `stepChildren` slot. The result is that the `stepChildren` slot of the importing prototemplate is masking the one in the exported proto. The instantiated view does not collect all the children from the entire proto chain (though NTK does do this at compile time for user proto templates).

The solution for exported user protos with `stepChildren` is to add a `viewSetupChildrenScript` to either your exported proto template or the importer that collects all of the `stepChildren` into a runtime `stepChildren` array.

```
// AFTER setting up stepChildren, views which "import" this proto
// must call inherited:?viewSetupChildrenScript();
exporter.viewSetupChildrenScript := func()
  begin
    // get the current value of the "extra" kids ...unless
    // the importer added NO kids, in which case, these are OURS
    local extraKids := stepChildren;

    local items := clone(extraKids);
    local kids;

    local whichFrame := self;

    while (whichFrame) do
      begin
        // get kids, but NOT using inheritance
        kids := GetSlot(whichFrame, 'stepChildren);

        // copy any extra stepChildren (but if NO extra kids
        // are defined, don't copy twice!)
        if kids and kids <> extraKids then
          ArrayMunger(items, 0, 0, kids, 0, nil);

        // go deeper into the _proto chain (or set whichFrame to nil)
        whichFrame := whichFrame._proto;
      end;

    stepChildren := items;
  end;
```

Note that you will have similar problems with declared children. If you have declared children you will also need to collect the `stepAllocateContext` slot.

How to Override the Standard Button Bar (11/22/96)

Q: What's the proper way to override the button bar, especially to cover up the buttons on the Newton OS 2.1 devices?

A: We don't recommend that typical applications obscure, cover up, or otherwise modify the standard button bar. From a user interface standpoint, it's a bad idea, because it can make the unit look unfamiliar or (in extreme cases) unusable. Some applications, typically those created for vertical markets, are designed to "take over" the interface, in which case it may be permissible to cover or disable the button bar.

`GetRoot().buttons.soft` will be non-`nil` if there is a "soft" button bar, that is, the button bar is located on the drawable screen. GetRoot():LocalBox() returns the rectangle that encloses the screen and the tablet. `GetAppParams`() returns information about the useful area of the screen, excluding the soft or hard button bar. Used together, this information will allow you to implement any combination of button bar disabling and/or button bar obscuring.

(Some earlier documentation mentioned a button bar API called `KillStdButtonBar`. That API is designed for use when you want to actually remove the button bar so you can replace it - probably with a floating slip. It is a fairly expensive call and does a lot of things you don't need if all you want to do is take over the screen. We recommend avoiding that call if possible.)

If the goal is simply to maximize the visible area of the base view, then the button bar should be obscured only for devices with a soft button bar (for instance, a MessagePad 2000). On devices with a "hard" button bar (for instance, a MessagePad 130), the root view encompasses a larger area than the LCD display because the input tablet is larger to account for the "hard" button bar. Drawing is limited to the screen so applications wouldn't increase their visible area by covering the "hard" button bar.

The "soft" button bar can simply be covered by your application's base view. The only trick is properly detecting if there is a soft button bar, and finding out where on the screen it happens to be. This code will give you the largest drawable application box, covering any soft button bar but not covering any hard buttons.

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

If the goal is to to prevent users from accessing the buttons, then the button bar should be obscured regardless of whether or not it is on the LCD screen. On units with a hard button bar, you must take into account the fact that part of the base view will be off-screen. (For instance, don't place your close box under the silk-screened buttons.) A simple way to accomplish this is by having a child view whose bounds are the `appArea` and locating the rest of the application within that child.

Note that on some Newton devices (for instance, the eMate 300), the buttons are not located in a view at all. On these devices, covering the entire tablet does not prevent the user from accessing the buttons (for instance, opening up the Extras Drawer).

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

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

Creating Large Strings (1/3/97)

Q: What is the best way to create a really large string?

A: The best way is to create the string as a virtual binary object (VBO). VBOs are described in the chapter "Data Storage and Retrieval" in the Newton Programmer's Guide.

To create a string as a VBO, you first need to create a VBO of class `'string`. Next, you need to use the global function `BinaryMunger` to munge an empty string into the VBO. This will properly prepare the binary object to be used as a NewtonScript string.

Finally, use the global function `StrMunger` as often as needed to copy new string data into the VBO. Here is a code example:

```
// Prepare a VBO to be the string
local myString := GetDefaultStore():NewVBO( 'string, Length("") );
BinaryMunger( myString, 0, nil, "", 0, nil );

StrMunger(myString, StrLen( myString ), nil, "MyNewString", 0, nil );
// Repeat with more data if necessary...
```

Note that unlike the C language's `stdio` library function, the NewtonScript `StrLen` function does not need to traverse the string to determine the string length, so you probably don't need to worry about performance hits from its usage.

Not all NewtonScript routines will necessarily "preserve" the VBO nature of large strings. For instance, if you concatenate strings using the `Stringer` global function or the & or && operators, the result is currently a non-VBO string. Be aware that if you accidentally create a very large non-VBO string, the code may throw a "out of NewtonScript heap memory" `evt.ex.outofmem` exception.

NEW: Strategy for Saving Modified Data (4/8/97)

Q: What's the best way to save modified data to a soup? For example, if I try to save the contents of a `clEditView` every time some change is made, typing is very slow. So, when should it be saved?

A: The best way we've found is to start a timer every time a change is made, and save the data when the timer expires. If a change is made before the timer expires, you can reset the timer. This way, the longer operation of saving to a soup won't happen until after the user pauses for a few seconds, but data will still be protected from resets. The data should also be saved when the views that edit it are closed, or when switching data items.

The timer can be implemented several ways. If no view is available, `AddDelayedCall` or `AddDelayedSend` can be used. The OS also provides `AddProcrastinatedSend` and `AddProcrastinatedCall`, which more or less implement the timer-resetting feature for you.

The best way to implement the timer when views are available is using the `viewIdleScript`. The `viewIdleScript` is preferred over `AddProcrastinatedCall/Send` because of better management of the event queue by the OS. When you call `SetupIdle` to start an idle timer, any existing idle timer is reset. Procrastinated calls/sends aren't currently implemented by resetting an existing timer, but rather by creating a delayed event which fires for each call and then checking a flag when the timer expires to see if it's the last one.

Where these methods are implemented depends on what layer of your code manages the soup entry. With the NewtApp model, the Entry layer manages the data, and each view in the Data layer is responsible for stuffing the modified data in the `target` frame, which is usually a soup entry. The entry layer implements `StartFlush` to start the timer, and `EndFlush` is called when the timer expires and which should ensure that the data is saved to the soup.

Your `StartFlush` equivalent could be implemented something like this:

```
   StartFlush: func()
      begin
         self.entryDirty := TRUE;
         :SetupIdle(5000); // 5 second delay
      end;
```

Your `viewIdleScript` would look something like this:

```
   viewIdleScript: func()
      begin
         :EndFlush();
          nil;       // return NIL to stop the idler until next
StartFlush
      end;
```

And your `EndFlush` equivalent would look something like this:

```
   EndFlush: func()
      if self.entryDirty then
         begin
            // getting data from editView may not be
            // necessary at this point
            myEntry.editViewData := <editView/self>.viewChildren;
            EntryChangeXmit(myEntry, kAppSymbol);
            self.entryDirty := nil;
         end;
```

Implementing `EndFlush` as a separate method rather than just putting the contents in the `viewIdleScript` makes it easy to call the method from the `viewQuitScript` or `viewPostQuitScript`, to guarantee that changes are saved when the view is closed. (The `viewIdleScript` may not have been called if the user makes a change then immediately taps the close box or overview or whatever.)

Built-In Apps and System Data

There Is No ProtoFormulasPanel (2/5/96)

Q: The current documentation says to use `protoFormulasPanel` for `RegFormulas`, but there does not appear to be such a template.

A: You are correct, there is no such template. You use a `protoFloatNGo` as your base and add your formula elements to it. The only requirements are:

1. There must be an `overview` slot that contains the text to show in the formula's overview.

2. `viewbounds.bottom` must be the height of your panel.

3. There must be a `protoTitle` whose `title` slot is the name of the formula panel.

ProtoPrefsRollItem Undocumented Slots (2/6/96)

Q: When I try to open my own system preference, I get a -48204 error. The preference registers OK with the `RegPrefs` function.

A: The documentation on `protoPrefsRollItem` is incomplete. You must define an `overview` slot which is the text to show in the overview mode. You can optionally define an `icon` slot which is an icon for the title in the non-overview mode (a title icon). Note that title icons are much smaller than normal icons.

SetEntryAlarm Does Not Handle Events (2/6/96)

Q: I tried to set the alarm of an event using the `SetEntryAlarm` calendar message, but the alarm is not set.

A: It turns out that `SetEntryAlarm` will not find events. You need to use a new Calendar API called `SetEventAlarm`. This function is provided in the Newton 2.0 Platform File. See the Platform File Notes for more information.

How to Avoid CardFile Extensions "Still needs the card" (5/9/96)

Q: I have a package that registers a data definition and view definition for a new card type for the Names application. If the package is installed on a card and the card is removed, the user gets the following error message:

"The package <The package name> still needs the card you removed. Please insert it now, or information on the card may be damaged."

How can I avoid this problem?

A: Currently, the cardfile `AddLayout` method requires that the symbol in the layout is internal. This bug will be fixed in a future ROM. To work around this, do the following:

```
local newLayout := {_proto: GetLayout("A Test Layout")};
newLayout.symbol := EnsureInternal (newLayout.symbol);
GetRoot().cardfile:AddLayout(newLayout);
```

For more information about issues for applications running from a PCMCIA card, see the article "The Newton Still Needs the Card You Removed"

How to Find Distance Between Two Points on the Earth (6/7/96)

Q: Is there an API which calculates the distance between two points on the Earth?

A: Yes. In the Newton 2.0 ROM there is a global function called `CircleDistance` which takes two longitude/latitude pairs and the units to use in reporting the distance, and `CircleDistance` returns the distance between the two points. NTK may give a warning about "`Unknown global function 'CircleDistance'`". This warning can be safely ignored so long as you're writing a package for a Newton 2.0 OS device.

`CircleDistance` (firstLong, firstLat, secondLong, secondLat, units)

Returns the distance between the two points. The distance is an integer. Currently `CircleDistance` rounds the distance to the nearest ten miles or ten kilometers.

`firstLong`: The longitude for the first point on the Earth.
`firstLat`: The latitude for the first point on the Earth.
`secondLong`: The longitude for the second point on the Earth.
`secondLat`: The latitude for the second point on the Earth.
`units`: A symbol specifying the units in which the distance will be calculated. Currently the options are `'miles` or `'kilometers`.

Note: the longitude and latitude arguments need to be integer values of the type used by `NewCity`. Check the section titled "Using the Time Zone Application" in the Built-In Applications and System Data chapter of the Newton Programmer's Guide for information on how to convert a longitude or latitude in degrees, minutes & seconds to an integer for `CircleDistance`.

Avoiding Query Bug in GetExtraIcons Call (8/2/96)

Q: Some calls to `GetExtraIcons` result in an undefined `Query` method exception. How can I fix this?

A: There is a bug in the implementation of `GetExtraIcons`. The code is not checking if the store has any extras information on it, so the `Query` message is getting sent to a `NIL` soup. The result is the exception.

At this time it is not clear if or when this bug will be fixed. I suggest you use the following workaround code when you call `GetExtraIcons`:

```
try
   GetExtraIcons(...)
   // do whatever you need to do here

onexception |evt.ex.fr.intrp;type.ref.frame| do
begin
   // check for a problem calling the Query function
   if currentException().data.errorCode = -48809 AND
         currentException().data.symbol = 'Query then
   begin
      // no extras drawer info on the store
   end ;
   else
      // a real error has occured, so let system handle it
      ReThrow() ;
end ;
```

How to Get Labels for Custom Names Fields (8/13/96)

Q: The Names application allows the user to add custom fields. If I have a specific entry, the cardfile method `bcCustomFields` returns the labels and values of the custom fields used in the entry. Is there a way to get a list of all the custom fields the user has defined?

A: Yes, you can pass `'customFields` to the names soup method `GetInfo`. This will return a frame of all the custom fields the user has defined. Each slot in this frame will have a frame with a 'label slot. Each 'label slot will be a string specified by the user. Here is an example:

```
GetStores()[0]:GetSoup(ROM_CardFileSoupName):GetInfo('customFields
)
```

...which returns:

```
{custom1: {label: "Customer Label"},
 custom2: {label: "Another label"}}
```

How to Add Confidential Owner Data (10/1/96)

Q: If I add confidential information to the Newton owner's card, all the information is beamed when the user beams the owner card. How can I keep confidential information from being sent?

A: Every owner entry has an `owner` slot, the value of which is a frame. This slot is removed from the entry before it is sent to another Newton device. You can add slots to this frame to store them, but keep them from being sent. Be sure to append your developer signature to any slot names you add to the `owner` frame.

Note that this only applies to the built-in Beam transport; any other transport or application can access all the slots in the Owner entry.

Adding Notes to Closed Notes Application (1/14/97)

Q: How do I add a note to the soup without having to have the Notepad application open? `MakeTextNote` doesn't work if Notes is closed.

A: You should use `MakeTextNote` to create the data, then add it to the soup entry using `soup:AddXmit` or `uSoup:AddToStoreXmit` (or one of the other soup functions.)

`MakeTextNote` always creates a frame with all the correct data that the Notes application requires. If the 2nd paramater (addit) is `TRUE`, it will add that frame to the Notes soup and show the note on the screen. If addIt is `NIL`, then the frame is returned.

It's the adding and showing that require the Notes app to be open, not the frame creation. For instance, to add a note to the default store, do something like:

```
   newNote := GetRoot().paperroll:MakeTextNote("Here is a sample
note", nil);
   GetUnionSoup("Notes"):AddToDefaultStoreXmit(newNote,
'|appSym:SIG|)
```

TapAction Slot Requires Text Slot to be Present (1/15/97)

Q: I have an autopart that I want to display an About slip when tapped. I added a `tapAction` slot but it does not work. What is missing?

A: The system will ignore the `tapAction` slot if it does not find a `'text` slot in the `partFrame` as well. The `'text` slot contains the name that will be displayed in the Extras drawer.

The following lines will correctly add a `tapAction` to a your part frame (in other words, your autopart):

```
DefineGlobalConstant('kTapActionFn,
func()
begin
   // your code goes here!
end);

// part MUST have a text slot for tapAction to be used
// text slot is the name seen by the user in Extras
SetPartFrameSlot('text, kAppName) ;
SetPartFrameSlot('tapAction, kTapActionFn) ;
```

NEW: Getting the Current Set of Multi-User Names (3/17/97)

Q: How can I get a list of all the students when a unit that supports it (for instance, the Apple eMate 300) is in multi-user mode?

A: The multi-user mode is implemented by the Home Page built-in application, and the list of users is stored in that application's preferences frame. Use `GetAppPrefs` to get the prefs

for that application for read-only purposes. Only the documented slots in that frame should be accessed. Other slots are neither documented nor supported, and their behavior may change. You should also check to ensure that the Home Page application exists on a particular unit before using any features. For example, here is a code snippet that evaluates to an array of user names, or NIL if the unit does not support multiple users or is not in multi-user mode.

```
if GetRoot().HomePage then
   begin
      local prefs := GetAppPrefs('HomePage, '{});
      if prefs.users and prefs.kMultipleUsers then
         foreach item in prefs.users collect item.name;
   end;
```

The Home Page preferences frame contains the following slots that may be accessed read only:

`kMultipleUsers`:	non-nil if multi-user mode is enabled
`kRequirePassword`	non-nil if passwords required in multi-user mode
`kDisallowNewUsers`	non-nil if new users can't be created at login
`users`	array of user frames or NIL.

A user frame contains the following slot that you may use as read-only data:

`name`	a string, the user-visible user's name

Keep in mind that new users could be created or existing users names may be changed at any time, and there is no notification when this happens. If necessary, you should check the set of users when your application launches. It is unlikely that new users will be created, deleted, or renamed while an application is open, unless this happens as a result of a new user being created at login. In this case, registering for change in the user configuration frame with `RegUserConfigChange` and watching for the `'kCurrentUser` slot to change will let you catch changes to the current set of multi-user names.

Controls and Other Protos

How to Set the Letter in AZTab Protos (3/26/96)

Q: How do I programatically select the letter in `protoAZTabs` or `protoAZVertTabs`?

A: You can use the `SetLetter` method of the AZTab protos:

```
protoAZTabs.SetLetter(newLetter, NIL)
```

Set the tab to the character specified by newLetter and update the hiliting. Note that this method does not send a `pickLetterScript` message.

Example:

```
// set myProtoAZTabs to the letter "C"
myProtoAZTabs:SetLetter($c, nil) ;

protoAZVertTabs.SetLetter...
see protoAZTabs.SetLetter
```

Return Value of ProtoSoupOverview's HitItem Message (1/6/97)

Q: What is the meaning of the return value of `protoSoupOverview:HitItem(...)`? I want to call the inherited method and use the return value to determine what action the system performed.

A: `ProtoSoupOverview:HitItem(...)` returns nil if it handled the tap and non-`nil` if it didn't handle the tap (the opposite meaning of the return value of `protoOverview`'s `HitItem` method).

Note that the final NPG documentation implies that `protoSoupOverview`'s `HitItem` is just like `protoOverview`'s `HitItem` method; this is a mistake in the documentation.

NEW: Don't Use ROM_UpArrowBitmap (3/28/97)

Q: I used the constant `ROM_UpArrowBitmap` in my application, and now my app appears partially invisible in the Newton 2.1 OS. What's wrong?

A: The constant `ROM_UpArrowBitmap` and the other directional arrow constants were not intended to be supported, and the value of the magic pointer has changed in Newton 2.1 OS. The change was made to better implement the (documented and supported) scrolling protos such as `protoUpDownScroller`.

`ROM_UpArrowBitmap` is named in the NTK Platform File defs file, and had mistakenly been mentioned in some public documentation from Apple, so you may have thought this was supported. If you have a reference to one of these magic pointers in the `icon` slot of a `clPictureView`, you'd have gotten an arrow graphic on the 2.0 and earlier releases of the OS, but on the Newton 2.1 OS, the changed value is not acceptable to the view system as a graphic. The result is that drawing is aborted when the OS tries to render the view with the arrow graphic, and views that would normally be drawn after the bad view will also fail to render, producing what appear to be invisible views that are otherwise functional.

You should use the documented protos to implement scrolling. If a custom scroller is needed, you can create your own graphic and include it in your NTK project.

Data Storage (Soups)

FrameDirty is Deep, But Can Be Fooled (8/19/94)

Q: Does the global function `FrameDirty` see changes to nested frames?

A: Yes. However, `FrameDirty` is fooled by changes to bytes within binary objects. Since strings are implemented as binary objects, this means that `FrameDirty` will not see changes to individual characters in a string. Since `clParagraphViews` try (as much as possible) to work by manipulating the characters in the string rather than by creating a new string, this means that `FrameDirty` can be easily fooled by normal editing of string `data`.

Here is an NTK Inspector-based example of the problem:

```
s := GetStores()[0]:CreateSoup("Test:DTS", []);
e := s:Add({slot: 'value, string: "A test entry", nested: {slot:
'notherValue}})
#4410B69  {slot: value,
           String: "A test entry",
           nested: {slot: notherValue},
           _uniqueID: 0}
FrameDirty(e)
#2        NIL

e.string[0] := $a; // modify the string w/out changing its reference
FrameDirty(e)
#2        NIL

EntryChange(e);
e.string := "A new string";   // change the string reference
FrameDirty(e)
#1A       TRUE

EntryChange(e);
e.nested.slot := 'newValue;   // nested change, FrameDirty is deep.
FrameDirty(e)
#1A       TRUE

s:RemoveFromStore()  // cleanup.
```

Limits on Soup Entry Size (2/12/96)

Q: How big can I make my soup entries?

A: In practice, entries larger than about 16K will significantly impact performance, and 8K should be considered a working limit for average entry size. No more than 32K of text (total of all strings, keeping in mind that one character is 2 bytes) can go in any soup entry.

There is no size limit built into the NewtonScript language; however, another practical limit is that there must be space in the NewtonScript heap to hold the entire soup entry.

There is a hard upper limit of 64K on Store object sizes for any store type. With SRAM-based stores there is a further block size limit of 32K. Trying to create an entry larger than this will result in `evt.ex.fr.store` exceptions. These limits are for the encoded form that the data takes when written to a soup, which varies from the object's size in the NS heap.

Newton Backup Utility and Newton Connection Utility cannot handle entries larger than 32K.

Note that Virtual Binary Objects (VBOs) in Newton 2.0 are no subject to the same restrictions. If you can store large objects as VBOs, you can store more information in your soup entries by referencing those VBOs.

Choosing EntryFlushXMit and EntryChangeXMit (4/17/96)

Q: What is the difference between the functions `EntryFlushXMit` and `EntryChangeXMit`?

A: The most important criterion when choosing between `EntryFlushXMit` and `EntryChangeXMit` is what will be done with the entry after the flush or change.

When an entry is added or changed, the system ensures that a cached entry frame exists in the NewtonScript heap. The system then writes the data in the frame to the store, skipping `_proto` slots. The result is that the data will be written to the store, and a cached frame will exist. Often, this is exactly what is desired because the entry is still needed since it will soon be accessed or modified.

In some cases, the data will be written to the soup with no immediate access afterwards. In other words, the data will not be used after being written to the soup. In these cases creating or keeping a cached entry frame in the NewtonScript heap is unnecessary and just wastes space and time. In these situations, `EntryFlushXMit` is a better option; it writes the data to the soup without creating the cached entry.

If any code accesses an entry that was just flushed, a new cached frame will be read in from the soup, just like when an existing entry is read for the first time.

The rule of thumb is: if an entry will be used soon after saving to the soup, then use `AddXMit` or `EntryChangeXMit`. If the entry will not soon be used again (so it doesn't need to take up heap space with the cached frame), then use `AddFlushedXmit` or `EntryFlushXMit`.

Some examples of good usage:

```
while entry do
begin
  entry.fooCount := entry.fooCount + 1;
  // nil appSymbol passed so don't broadcast
  EntryFlushXMit(entry, nil);
  entry := cursor:Next();
end;                          // Could broadcast now


foreach x in kInitialData do   // if new, may not need broadcast
   soup:AddFlushedXmit(Clone(x), nil);
```

Why Xmit Functions Seem to Leak Memory (10/31/96)

Q: I've noticed that the system seems to leak a little bit of memory every time I call an Xmit soup function, but not if I call the non-xmitting versions of the functions. For example, executing this code shows a little less free memory each time stats is called:

```
gc(); stats();
soup:AddToDefaultStoreXmit({ foo : "a test string"}, 'bar:SIG);
gc(); stats();
soup:AddToDefaultStoreXmit({ foo : "another test string"}, 'bar:SIG);
gc(); stats();
```

A: There is no leak. What's actually going on is that the Xmit versions of the soup methods do their broadcasting in deferred actions. That's good, because it means that broadcast handlers that might throw or have other side effects won't break your code. A little bit of heap memory is used to keep track of the deferred action and its arguments. This memory is released after the deferred action executes, which is typically immediately after control returns to the top level.

If you select all the test code in the inspector and press ENTER, NTK compiles the entire selection and executes it as a single operations, so control doesn't return to the top level (thus allowing the deferred actions to execute) until after the last operation. The deferred action created by each call to the Xmitting function will still be pending, so the space won't have been released yet, and stats reflects this. If the example is executed one line at a time, you'll see that no memory is actually leaked.

Limit on Soup Name Length (12/10/96)

Q: What is the maximum number of characters I can use for a soup name?

A: With the Newton OS 2.0 release, soup names, like index data, are limited to 39 Unicode characters. If you attempt to create a soup with a longer name, the OS will create a soup with only the first 39 characters of the longer name. We recommend you avoid this truncation, because typically the truncation removes some or all of your registered signature, and the resulting name will not be guaranteed unique.

You can still provide longer/prettier names for users, by using the `soupDef` mechanism and putting the long name (typically without appended signature) in the `'userName` slot of that data structure.

How to Use Begin and End Symbols with WhichEnd (1/8/97)

Q: The `WhichEnd` cursor method returns the symbols `'begin` or `'end`, depending on where the cursor is in a soup. Why does NTK complain when I try to check for these symbols?

A: Unfortunately, these are reserved words so NTK won't let you type them into normal code. The work-around is to enclose the symbol in vertical bars.

For instance, you can use code like:

```
if myCursor:WhichEnd() = '|begin| then
   :WeAreAtBeginning();
```

EntryChange on Modified Tags Array Throws -48022 (1/15/97)

Q: I added a tag to the array of tags in an entry. When I call `EntryChange` on the modified entry I get a -48022 error. What is wrong?

A: There is a known bug in Newton OS 2.0 (which is fixed in Newton OS 2.1) that can cause this problem. It can occur when you initially create an entry with an empty array as the

value of the tag index slot. The workaround is not to use the empty array. Use NIL instead. If you need to add an array of tags later, you can do so.

NEW: How to Avoid Resets When Using VBOs (2/27/97)

Q: When writing large amounts of information to virtual binary objects (VBOs), my Newton device sometimes resets. What is going wrong?

A: The problem happens because of how the Newton OS manages the memory for VBOs. Writing to VBOs in low memory conditions can sometimes cause the device to reset because no free pages are available for other OS operations.

To work around this problem, you can periodically call the global function `ClearVBOCache` while modifying VBOs. You can also work around the problem by putting the VBO in a soup entry and using `EntryChangeXmit` or `EntryFlushXmit`.

In all versions of the Newton 2.x OS released to date, VBOs (including packages) are managed in 1K pages. When you write to a VBO, the "dirty" pages can remain in the system heap, taking up space. `ClearVBOCache` takes a reference to a VBO as an argument, and moves the dirty pages for a given VBO to the store, freeing up the system memory. Note that this function does not commit the changes to the VBO, while `EntryChangeXmit` and `EntryFlushXmit` do commit the changes.

The likelihood of the problem depends on the amount of system memory currently available and how many pages of VBOs are modified. We recommend that you modify no more than 32 pages of VBOs before committing the changes or calling `ClearVBOCache`. For example, modifying 32K of contiguous data, or a single byte in 32 different pages of one VBO, or even a single byte in 32 different VBOs all modify 32 total pages of VBO data. Don't do this too often, though. Calling `ClearVBOCache` repeatedly for modifications to the same page of a VBO or when there are only a few modified pages will needlessly slow the machine.

If you are experiencing this problem, you should consider redesigning your application to minimize the amount of uncommited VBO data. When finished with a VBO, commit it to a soup entry as soon as possible or let it become unreferenced.

Desktop Connectivity (DILs)

Differences between MNP, Modem, Modem-MNP, and Real Modems (2/5/96)

Q: I want to just connect to a Newton device over a cable from a MacOS or Windows machine - what do I need to use to get reliable communications?
Q: I want to have the DILs answer an incoming call over a modem. How can I do that?
Q: What's the difference between the "Serial" and "Modem" Mac connection types?

A: In release 1.0 of the DILs, the best way to connect to a Newton device is by using a MNP connection over a serial cable. This is what you're using when you set connection type "Modem" on MacOS computers and "MNP" on Windows computers. This actually has nearly nothing to do with modems as such; it means you're connecting over a serial cable

using MNP error correction and compression. (And on Windows, it's the only supported option at this time.)

Currently you cannot use a true modem with the DILs to connect to a Newton device.

In general, you will never use the "Serial" connection type on a MacOS computer; that connects over a serial cable (like "Modem" does) but offers no error detection. Therefore, you would have to write your own code to check that data arrived safely.

CDPipeInit Returning -28102 on MacOS Computers (2/13/96)

Q: When I call the DILs function `CDPipeInit`, it returns a -28102 error (Communication tool not found). I've checked that the tool is installed properly, and the DIL sample application works fine. What's wrong?

A: A common cause of this error code is that the CSTR resources haven't been linked into your final executable. Those resources are used to find the filenames of the communications tools. Add the CSTR.rsrc file to your project and see if that fixes things.

Getting Serial Port Names on MacOS Computers (2/13/96)

Q: Different MacOS computers have different numbers of ports, different names for the ports, and the port names are translated into other languages in non-English MacOS System Software. How can I tell what serial ports are available?

A: You can use the Communications Toolbox to get the list of available serial ports. This code has been added to version 2 of the SoupDrink sample code - see the `SetupPortMenu` function in SoupDrink.c for an example.

Corruption of Some Binary Objects (5/13/96)

Q: Sometimes when I send a binary object (including a real) from the Newton device, it is corrupted when I read it with the FDILs on the desktop. What's going on?

A: When FDILs 1.0 receive a binary object, they must "guess" whether it is a string or not. This guessing algorithm has a flaw which can result in non-string binary objects being treated as strings, and thus the Unicode conversion process is performed on them, which results in corruption of the desktop binary object.

The easiest ways to avoid this problem are to either receive the data with the CDIL (in other words, don't include them in the frame), or else to ensure that either the first two or the last two bytes of the binary object are non-zero. This workaround will not be necessary in future versions of the DIL libraries.

Note: this has been fixed in the 1.0.2 Windows DILs.

Error -28801 or -28706 from FDget (5/13/96)

Q: Why does the `FDget` function return error -28801 (Out of heap memory) or -28706 (Invalid parameter)? I don't think I'm out of memory, and I don't always get this error code so my parameters must be right. What is wrong?

A: Sometimes these error codes are accurate and indicate that not enough memory could be allocated or that a parameter was invalid. Sometimes they are the result of a bug caused by having multiple copies of a rectangle slot inside a frame.

The protocol which is used to send frames can perform an optimization for certain rectangle frames, which transmits them in a compact form (5 bytes instead of up to 60). However, if a given frame holds the exact same rectangle frame in more than one slot, the data will not be handled correctly and will either result in one of these error codes, or alternatively it might substitute some other object in place of the frame, or might possibly crash.

This is a relatively uncommon problem, since all of the values in the frame must be between 0 and 255, and the frame must have the same rectangle in it twice - two frames with equivalent data would not trigger the problem. For example, frame "A" would cause the problem, but frames "B", "C" and "D" would not.

```
A:={first: {left:3, right: 30, top:10, bottom:90}};
A.second := A.first;        // triggers the problem

B:={first: {left:3, right: 30, top:10, bottom:90}};
B.second := clone(B.first);    // cloning avoids the problem

C:={first: {left:3, right: 30, top:10, bottom:90, foo: nil}};
C.second := C.first;           // no problem since C.foo exists

D:={first: {left:3, right: 30, top:10, bottom:1000}};
D.second := D.first;       // no problem since D.bottom is >255
```

To work around this problem, you can clone the frame (as in frame "B") or add another slot to the frame (as in frame "C") or ensure that the values are not between 0 and 255 (frame "D").

Note: this has been fixed in the 1.0.2 Windows DILs.

Using CDPipeListen Asynchronously in Windows Applications (7/15/96)

Q: I am passing in a callback function to CDPipeListen, but it never seems to be called. What is going wrong?

A: Due to a bug in CDPipeListen, the callback function never gets called in Windows applications. You will have to use a synchronous listen, then wait for the state of the DIL pipe to change before accepting the connection. The following code shows how to properly accept a connection.

```
anErr = CDPipeListen( gOurPipe, kDefaultTimeout, NULL, 0 );

if (!anErr)
{
   // This code doesn't need to be executed on MacOS, but
```

```
   // is currently required for Windows.  We need to loop,
   // waiting for the connection state to change to
   // kCDIL_ConnectPending.
   endTime = (GetTickCount()/1000) + 30;  // to timeout in 30 seconds
   while ((GetTickCount()/1000) < endTime )
   {
      if (CDGetPipeState( gOurPipe ) == kCDIL_ConnectPending) {
         anErr = CDPipeAccept( gOurPipe );
         break;
      } else
         CDIdle( gOurPipe );
   }
}
```

Note: this has been fixed in the 1.0.2 Windows DILs.

Unicode Strings and Memory Buffers (8/26/96)

Q: Sometimes when I use the DILs to get a string, some memory gets corrupted even though I'm sure I've allocated more memory than I have characters in the string. What's going on?

A: One common cause is that strings arriving from a Newton device are in Unicode - which takes two bytes per character. If you've only allocated one byte per character, you risk memory corruption because the data is converted to the one-byte form only after the whole buffer has arrived. This might be too late to prevent overrunning the buffer bounds. So, you need to allocate enough space for the Unicode version.

For example, if you're expecting strings to be up to 50 characters long, you must allocate at least 100 bytes of memory in your buffer.

Digital Books

BookMaker Page Limitations? (11/19/93)

Q: Does the Newton BookMaker have limitations concerning the size of books or page count?

A: The current page limitation of BookMaker is 16 million pages, a very unlikely size to be exceeded. However, since the entire book is held in memory during the build process, you need to have enough application heap space allocated to the BookMaker desktop application. If there is not enough RAM available on your desktop computer to process a book, you can divide it into smaller parts and link them with the `.chain` command.

Drawing and Graphics

Drawing Text on a Slanted Baseline (9/15/93)

Q: Is it possible in the Newton OS to draw text on a slanted baseline? I don't mean italics, but actually drawing a word at a 45 or 60 degree angle and so on. For example, can text be drawn along a line that goes from 10,10 to 90,90 (45 degrees)?

A: The drawing package in the Newton OS supports no calls for rotating text. Note: this is like QuickDraw in the MacOS operating system. In MacOS, the workaround is to draw the text into a bitmap and then rotate the bits; you can do the same on a Newton device. In the Newton OS, we even provide calls to rotate a bitmap in 90 degree increments.

You might consider creating a font having characters that are pre-rotated to common angles (such as 30 or 45 degrees) so that applications could just draw characters rather than actually having to rotate a bitmap.

LCD Contrast and Grey Texture Drawing (11/10/93)

Q: An artist working with me did a wonderful job rendering a 3D look using several different dithered grey textures. The problem is that when her image is displayed on a Newton display everything on the screen dims. Is it possible that the image causes too much display current to maintain contrast?

A: What you're seeing is a well-known problem with LCD displays, and there's not a lot you can do about it. It's especially aggravated by large areas of 50% dithered gray (checkerboard) patterns, but the light gray and dark gray patterns also cause some of it.

The user interface of the Newton OS deliberately avoids 3-D styling and 50% dithered grays as much as possible for this reason. If you know your application is going to display large gray areas, you can adjust the contrast yourself on some hardware devices. There's a global function, `SetLCDContrast`, to do just that. However, changing the contrast with no end user control is not considered a good user-interface practice.

Destination Rectangles and ScaleShape (3/11/94)

Q: What is a valid destination rectangle for the 2nd argument to `ScaleShape`?

A: The destination rectangle must be at least 1 pixel wide and 1 pixel high. Each element of the bounds frame must have values that fit in 16 bits, -32768...32767. 0-width/height and negative width/height bounding boxes may appear to work in some cases, but are not supported.

How to Rotate Bitmaps Left (3/5/96)

Q: When I rotate a bitmap left using `MungeBitmap`, it sometimes shifts the data. How can I rotate left correctly?

A: There is a bug in the Newton 2.0 OS that manifests when the row size of the unrotated bitmap is not an even byte boundary. The result can be a shift of data up to 7 pixels.

You can work around this bug most efficiently by replacing the left rotation with three calls to `MungeBitmap` using these operations: `'flipHorizontal`, `'flipVertical`, and `'rotateRight`. (`'rotateRight` three times will work as well, but it is less efficient bacause flips are faster than rotates.)

Remember: "Three Rights (or Two Flips and a Right) Make a Left".

Newton Bitmap Formats (5/14/96)

Q: What is the format for bitmap binary objects in the Newton OS?

A: There are several bitmap formats used in the Newton OS. The Newton OS provides routines for creating and manipulating bitmaps at runtime, and uses other formats for displaying bitmaps from developer packages.

If you want to create a bitmap object at compile time, below is a description of the format of a simple bitmap object. If you want to create a bitmap at run time, we strongly encourage you to use `MakeBitmap` and copy data into the bitmap.

Simple Bitmaps

Normally, bitmaps are created at compile time using Newton Toolkit picture editors or functions (for example, GetPICTAsBits). If you want to create bitmaps dynamically at compile time, you can create a simple bitmap object with the following format.

Warning: Different formats may be used by images or functions in future ROMs. This format will still be supported for displaying images. This format does *not* describe images created by other applications nor any images provided or found in the Newton ROM. You can use the following format information to create and manipulate your own bitmaps -- preferably at compile time:

```
{
   bounds: <bounds frame>,
   bits:   <raw bitmap data>,
   mask:   <raw bitmap data for mask - optional>
}

 Binary object <raw bitmap data> - class 'bits

 bytes    data-type  descr
 0-3      long       ignored
 4-5      word       #bytes per row of the bitmap data
                     (must be a multiple of 4)
 6-7      word       ignored
 8-15     bitmap     rectangle - portion of bits to use--see IM I
 8-9      word       top
 10-11    word       left
 12-13    word       bottom
 14-15    word       right
 16-*     bits       pixel data, 1 for "on" pixel, 0 for off
```

The bitmap rectangle and bounds slot must be in agreement regarding the size of the bitmap.

MakeBitmap Shapes

If you want to create bitmap data at run time or extract bitmap data from a bitmap created with the `MakeBitmap` global function, use the `GetShapeInfo` function to get the bitmap and other slots required to interpret the meaning of the bitmap created by `MakeBitmap`.

Warning: the following information applies only to bitmaps of depth 1 (black and white bitmaps) created by your application with MakeBitmap. Do *not* rely on `GetShapeInfo` or the following slots for images created by other applications, images stored in the Newton ROM, images created with functions other than MakeBitmap, nor images with a depth other than 1.

If you created a bitmap using `MakeBitmap` of `depth` 1, the return value of `GetShapeInfo` contains frame with information you can use to interpret the bitmap data.

This frame includes a `bits` slot referencing the bitmap data for the bitmap. This bitmap data can be manipulated at run time (or copied for non-Newton use), using other slots in the return value of `GetShapeInfo` to interpret the bitmap binary object: `scanOffset`, `bitsBounds`, and `rowBytes`. For instance, the first bit of the image created with `MakeBitmap` can be obtained with code like:

```
bitmapInfo := GetShapeInfo(theBitmap);
firstByte := ExtractByte(bitmapInfo.bits, bitmapInfo.scanOffset);
firstBit := firstByte >> 7; // 1 or 0, representing on or off
```

Note that `rowBytes` will always be 32-bit aligned. For instance, for a bitmap with a `bitsBounds` having width 33 pixels, `rowBytes` will be 8 to indicate 8 bytes offsets per horizontal line and 31 bits of unused data at the end of every horizontal line.

Difference Between LockScreen and RefreshViews (1/15/97)

Q: In the NPG, it states that sending a view the `view:LockScreen(nil)` message forces an "immediate update". How is this different from calling `RefreshViews`?

A: When you post drawing commands (for example, `DrawShape`) the system normally renders the shape on the screen immediately. `:LockScreen(true)` provides a way to "batch up" the screen updates for multiple drawing calls. Sending `:LockScreen(nil)` "unplugs" the temporary block that has been placed on the screen updater, causing all the batched drawing changes to be rendered on the LCD.

`RefreshViews` tells the system to execute the commands needed to draw every view that has a dirty region. You can think of it as working at a level "above" the screen lock routines. When you send the message `Dirty`, it does not immediately cause the system to redraw the dirtied view, instead it adds the view to the dirty area for later redrawing.

You could lock the screen, dirty a view with a `SetValue`, call `RefreshViews` (and not see an update) draw a few shapes, and then, when you unlock the screen, the refreshes to the dirty regions and your shapes will all appear at once.

A bit more detail on the interaction between `LockScreen` and `RefreshViews`:
1. Does `LockScreen(nil)` result in a `RefreshViews`?

No. `LockScreen(true)` just stops the hardware screen from updating from the offscreen buffer. `LockScreen(nil)` releases that lock which usually causes the hardware screen to update soon thereafter.

2. While the screen is locked, will `SetValues` draw into the offscreen buffer?

`SetValue` doesn't draw. Otherwise, see 1.

3. While the screen is locked, what is the result of calling `RefreshViews`?

It will draw any dirty views into the offscreen buffer.

Drawing White Text on a Filled Background (1/15/97)

Q: I tried using `vfFillWhite` and `kRGB_0` but neither seems to work. How do I draw white text?

A: kRGB_White has some unusual behavior. If you want a white-on-black effect then you will need to use one of two workarounds:

For white text on a filled background using a style frame, set your background to the desired shade using either the fill color of the view or a filled object (in other words, do not do this just by setting the color fill of the text). Then use the `textPattern` slot in the style frame to make the text black (`kRGB_Black`) and set the `transferMode` to `modeBic`.

For white text on black using the color slot of the viewFont frame, use `kRGB_Gray1` for something that is as close to white as you can get.

Interaction Between Transfer Modes and Gray Patterns (1/15/97)

Q: How do the transfer modes interract with the new gray shades in Newton OS 2.1?

A: Here is how the transfer modes interact with images. Colors are determined by looking up the value in the color table. For instance, white means that the indexed pixel value is white in the color table. All the NOT modes operate on the values from the color table. In other words, the pixel value is looked up before the NOT is applied.

When the source and destination are different bit depths, the source is effectively expanded or shrunk to match the depth of the destination bitmap prior to drawing. When expanding, the index into the color table's bit pattern is repeated to fill the destination pixel. For instance, a 2-bit index of 0x1 (binary 01) is expanded to 0x3 (binary 0011) for 4 bits, while a 2-bit index of 0x2 (binary 10) is expanded to 0xC (binary 1100).

`modeOr` - Replaces pixels under the non-white part of the source image with source pixels. If the source pixel is white, the destination pixel is unchanged.

`modeXor` - Inverts pixels under the non-white part of the source image. Destination pixels under the white part of the source image are unchanged. This actually XORs the values in the source and destination pixels. For example, for destination of 0xA (75% grey), source 0x0 (white) produces result 0xA (unchanged). Source 0xF (black), produces result 0x5 (25% grey, or inverted). Source pixels of other values have less utility. For example, source 0x5 (25% grey) produces result 0xF (black), while source 0xA (75% grey) produces result 0x0 (white), and source 0x3 (50% grey) produces result 0x9 (slightly less than 75% grey).

`modeBic` - Erases screen pixels under the non-white part of the source image, making them all white. Destination pixels under the white part of the source image are unchanged. This actually, does a bitwise NOT, so it is really only useful when source pixels are either 0 (white) or 0xF (black). With other values, weird things happen, For example, destination 0xF with source 0xA produces result 0x5. Destination 0x0 with source 0xA produces result 0x0. Destination 0x3 with source 0xA produces result 0x1.

`modeNotCopy` - Replaces screen pixels under the black part of the source image with white pixels. Screen pixels under the white part of the source image are made black.

`modeNotOr` - Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are made black.

`modeNotXor` - Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are inverted.

NEW: Limitations of GrayShrink (3/4/97)

Q: Why isn't `GrayShrink` doing what I want it to when I use it with relatively small bitmaps?

A: `GrayShrink` was designed for rendering relatively large images such as received faxes into a moderately large part of a Newton display. It works by setting a flag in the bitmap that tells the imager to gather multiple bits from the source bitmap and turn them into a single gray pixel when drawing through a reducing transform.

If passed a bitmap that is more than one bit deep, the shrinking algorithm is not appropriate and so `GrayShrink` will not modify the bitmap. The end result will be a transformed (shrunk) image with the same bit depth as the original. That is, the shrinking will still happen, but the graying won't.

`GrayShrink` will not work with read-only bitmaps (it is unable to set the flag.) The result will still be a transformed (shrunk) image, but pixels will not be combined to gray. There is no way to clear the flag once it has been set. After `GrayShrink` has modified a bitmap, drawing it to the screen through any scaling transform that reduces the image will produce a pixel combined gray result.

The `GrayShrink` pixel gathering algorithm produces an anomaly along the righthand side of the reduced image. When rendering large bitmaps into a reasonably large destination, this is generally uunnoticeable. However, when used with small source bitmaps or when rendering into a small area, several columns along the right side of the result may not be drawn, and the anomaly is easily seen. We recommend using `GrayShrink` and the `'drawGrayScaled` setting for `protoImageView` only for large source images such as incoming faxes or scanned data.

NEW: Limitations of MungeBitmap (4/3/97)

Q: When I use `MungeBitmap` to flip or rotate a grayscale image, it gets corrupted. What's wrong?

A: `MungeBitmap` does not properly handle bitmaps with a depth greater than 1. You can work around this problem by using `kMungeBitmapFunc`, which has the same calling conventions and return value as `MungeBitmap`. `kMungeBitmapFunc` is provided in the Newton 2.1 Platform file, version 1.2b1 or later.

Calling `MungeBitmap` with the `'rotateLeft`, `'rotateRight`, or `'flipHorizontal` options will trigger the bug. The `'rotate180` and `'flipVertical` arguments to `MungeBitmap` work correctly with deeper bitmaps.

Note that with the 2.1 release of the Newton OS in the English Apple MessagePad 2000 and Apple eMate 300 products, the built-in Drawing stationery in NewtonWorks exhibits this bug when rotating gray bitmaps.

Endpoints & Comm Tools

What is Error Code -18003 (3/8/94)

Q: What is error code -18003?

A: This signal is also called SCC buffer overrun; it indicates that the internal serial chip buffer filled, and the NewtonScript part didn't have time to read the incoming information. You need to either introduce software (XON/XOFF) or hardware flow control, or make sure that you empty the buffer periodically.

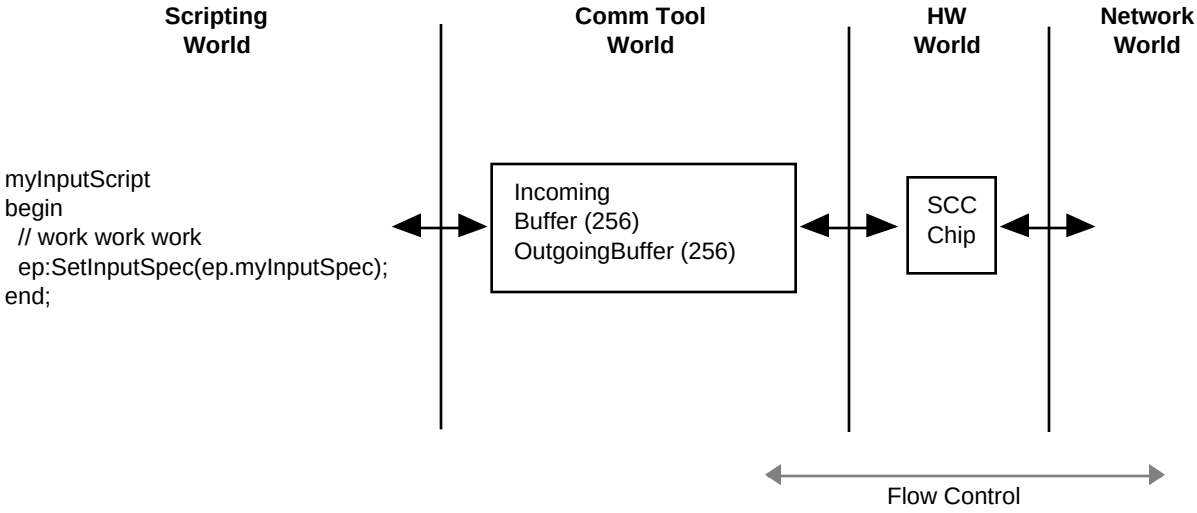
You will also get -18003 errors if the underlying comms tool encounters parity or frame errors. Note that there's no difference between parity errors, frame errors, or buffer overruns; all these errors are mapped to -18003.

See the diagram for an explanation of what is going on concerning the serial chip, the buffers and the scripting world.

The SCC chip gets incoming data, and stores it in a 3-byte buffer. An underlying interrupt handler purges the SCC buffer and moves it into a special tools buffer. The comms system uses this buffer to scan input for valid end conditions (the conditions which cause your inputSpec to trigger). Note that you don't lose data while you switch inputSpecs; it's always stored in the buffer during the switch.

Now, if there's no flow control (XON/XOFF, HW handshaking, MNP5), the network side will slowly fill the tool buffer, and depending on the speed the buffer is handled from the scripting world sooner or later the comms side will signal a buffer overrun. Even if flow control is enabled, you may still receive errors if the sending side does not react fast enough to the NewtonÕs plea to stop sending data. In the case of XON/XOFF, if you suspect that one side or the other is not reacting or sending flow control characters correctly, you may want to connect a line analyzer between the Newton and the remote entity to see what is really happening.

If you have inputScripts that take a long time to execute, you might end up with overrun problems. If possible, store the received data away somewhere, quickly terminate the inputSpec, then come back and process the data later. For instance, you could have an idleScript which updates a text view based on data stored in a soup or in a slot by your inputSpec.



Newton Remote Control IR (Infra-red) API (6/9/94)

NTK 1.0.1 and future NTK development kits contain the needed resources to build applications that control infrared receive systems, consumer electronics systems and similar constructs.

This development kit is fairly robust, and will produce send-only applications.

Note: The NTK 1.1 platforms file is required to produce code that will execute correctly on the MessagePad 100 upgrade units.

```
cookie := OpenRemoteControl();
```
Call this function once to initialize the remote control functions. It returns a magic cookie that must be passed to subsequent remote control calls, or nil if the initialization failed.

```
CloseRemoteControl(cookie);
```
Call this function once when all remote control operations are completed, passing cookie returned from `OpenRemoteControl`. Always returns `nil`. cookie is invalid after this call returns.

```
SendRemoteControlCode(cookie, command, count);
```
Given the cookie returned from `OpenRemoteControl`, this function sends the remote control command (see below for format of data). The command is sent count times. count must be at least 1. Returns after the command has been sent (or after the last loop for `count` > 1). (see diagram)

Each command code has the following structure:

```
struct IRCodeWord {
   unsigned long name;
   unsigned long timeBase;
   unsigned long leadIn;
   unsigned long repeat;
```

```
   unsigned long leadOut;
   unsigned long count;
   unsigned long transitions[];
};
```

`name`	identifies the command code; set to anything you like
`timeBase`	in microseconds; sets the bit time base
`leadIn`	duration in timeBase units of the lead bit cell
`repeat`	duration in timeBase units of the last bit cell for loop commands
`leadOut`	duration timeBase units of the last bit cell for non-loop commands
`count`	one-based count of transitions following
`transitions[]`	array of transition durations in timeBase units

Note that the repeat time is used only when the code is sent multiple times.

See Remote.π, Sony.r, RC5.r, and RemoteTypes.r files for examples. The .rsrc files have templates for ResEdit editing of the Philips and Sony resources. See Remote IR Sample code for more details.

Things To Know Before You Burn The Midnight Oil:

If the Newton goes to sleep, the IR circuits are powered down, and any subsequent sends will fail. If you want to override this, you need to have a powerOffhandler close the remote connection, and when Newton wakes up the application could re-open the connection.

If two applications are concurrently trying to use the IR port (beaming and remote control use for instance), this will cause a conflict.

Sample Code

The Remote IR Sample is part of the DTS Sample code distribution, you should find it on AppleLink and on the Internet ftp server (ftp.apple.com).

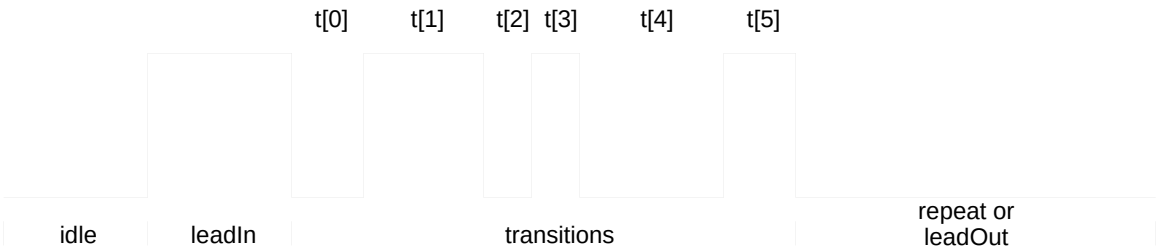
By way of a quick summary: the sample has an array of picker elements with the resource definitions bound to the index (ircode inside the application base view).

You specify the constant that is an index to the array, get the resource using the NTK function `GetNamedResource` and when you send data, use the constant as the resource used.

`OpenRemoteControl` is called in `viewSetupFormscript`, and `closeRemoteControl` is called in `viewQuitScript`. Note that these are methods, not global functions; same is true of `SendRemoteControlCode`.

More Information

Consult the IR samples available on ftp.apple.com (Internet) and on the Newton Developer CD-ROMs.



Communications With No Terminating Conditions (6/9/94)

Q: How do I handle input that has no terminating characters and/or variable sized packets?

A: Remember that input specs are specifically tied to the receive completion mechanism. To deal with the situations of no terminating characters or no set packet sizes, you need only realize that one receive completion is itself a complete packet. Set the `byteCount` slot of your input spec to the minimum packet size. In your input script, call `Partial` to read in the entire packet, and then call `FlushInput` to empty everything out for your next receive completion.

If this is time-delay-based input, you may be able to take advantage of `partialScripts` with `partialFrequencies`. Call the `Ticks` global function if necessary to determine the exact execution time of a `partialScript`.

What Really Happens During Instantiate & Connect (6/14/94)

Q: Does `Instantiate`, `Bind` or `Connect` touch the hardware?

A: Exactly what happens depends on the type of endpoint being used. In general:

The endpoint requests one or more communications services using endpoint options like this:

```
{
  type:      'service,
  label:     kCMSAsyncSerial,
  opCode:    opSetRequired
}

<see diagram section A>
```

The CommManager task creates the appropriate CommTool task(s) and replies to the communications service request. Each CommTool task initializes itself . In response to the `Bind` request the CommTool acquires access to any physical hardware it controls, such as powering up the device. The endpoint is ready-to-go.

```
<see diagram section B>
```

An endpoint may use multiple CommTool tasks, but there will be a single NewtonScript endpoint reference for them.

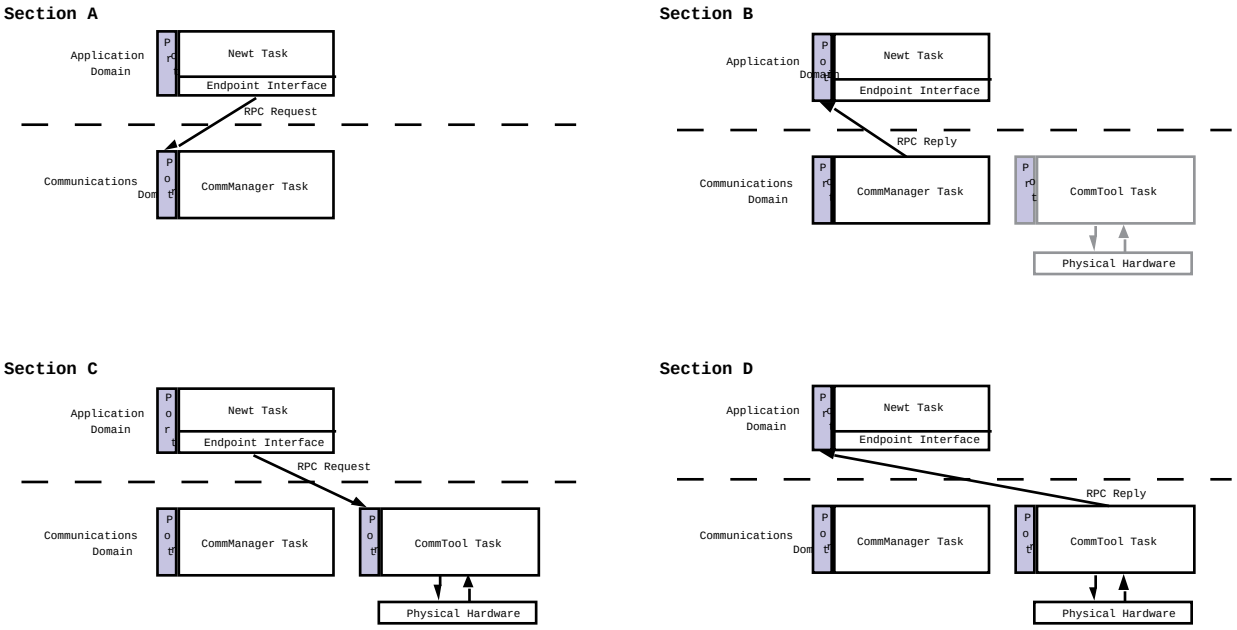
When the endpoint requests a connection, the CommTool interacts wih the physical hardware (or a lower level CommTool) as necessary to complete the connection, depending on the type of communications service. For example, ADSP will use the endpoint address frame to perform an NBP lookup and connection request. MNP will negotiate protocol specifications such as compression and error correction.

<see diagram section C>

The CommTool completes the connection and replies to the connection request. Note that if this is done asynchronously, the Newt task continues execution, giving the user an option to abort the connection request.

<see diagram section D>

`Disconnect` functions similarly to `Connect`, moving the endpoint into a disconnected state. `Unbind` releases any hardware controlled by the CommTool. `Dispose` deallocates the CommTool task.



Unicode-ASCII Translation Issues (6/16/94)

Q: How are out-of-range translations handled by the endpoints? For example, what happens if I try to output "\u033800AE\u Apple Computer, Inc."?

A: The first Unicode character (0338) is mapped to ASCII character 255 because is it out of the range of valid translations, and the second Unicode character (00AE) is mapped to ASCII character A8 because the Mac character set has a corresponding character equivalent in the upper-bit range.

All out-of-range translations, such as the 0338 diacritical mark above, are converted to ASCII character 255. However, the reverse is not true! ASCII character 255 is converted to

Unicode character 02C7. This means you will need to escape or strip all 02C7 characters in your strings before sending them if you want to use ASCII character 255 to detect out-of-range translations. Character 255 was picked over character 0 because 0 is often used as the C-string terminator character.

The built-in Newton Unicode-ASCII translation table is set up to handle the full 8-bit character set used by the MacOS operating system. Although `kMacRomanEncoding` is the default encoding system for strings on most Newtons, you can specify it explicitly by adding one of the following encoding slots to your endpoint:

```
encoding:  kMacRomanEncoding; // Unicode<->Mac translation
encoding:  kWizardEncoding ;    // Unicode<->Sharp Wizard translation
encoding:  kShiftJISEncoding ;       // Unicode<->Japanese ShiftJIS
translation
```

For `kMacRomanEncoding`, the upper 128 characters of the MacOS character encoding are sparse-mapped to/from their corresponding unicode equivalents. The map table can be found in Appendix B of the NewtonScript Programming Language reference. The upper-bit translation matrix is as follows:

```
short gASCIIToUnicode[128] = {
     0x00C4, 0x00C5, 0x00C7, 0x00C9, 0x00D1, 0x00D6, 0x00DC, 0x00E1,
     0x00E0, 0x00E2, 0x00E4, 0x00E3, 0x00E5, 0x00E7, 0x00E9, 0x00E8,
     0x00EA, 0x00EB, 0x00ED, 0x00EC, 0x00EE, 0x00EF, 0x00F1, 0x00F3,
     0x00F2, 0x00F4, 0x00F6, 0x00F5, 0x00FA, 0x00F9, 0x00FB, 0x00FC,
     0x2020, 0x00B0, 0x00A2, 0x00A3, 0x00A7, 0x2022, 0x00B6, 0x00DF,
     0x00AE, 0x00A9, 0x2122, 0x00B4, 0x00A8, 0x2260, 0x00C6, 0x00D8,
     0x221E, 0x00B1, 0x2264, 0x2265, 0x00A5, 0x00B5, 0x2202, 0x2211,
     0x220F, 0x03C0, 0x222B, 0x00AA, 0x00BA, 0x2126, 0x00E6, 0x00F8,
     0x00BF, 0x00A1, 0x00AC, 0x221A, 0x0192, 0x2248, 0x2206, 0x00AB,
     0x00BB, 0x2026, 0x00A0, 0x00C0, 0x00C3, 0x00D5, 0x0152, 0x0153,
     0x2013, 0x2014, 0x201C, 0x201D, 0x2018, 0x2019, 0x00F7, 0x25CA,
     0x00FF, 0x0178, 0x2044, 0x00A4, 0x2039, 0x203A, 0xFB01, 0xFB02,
     0x2021, 0x00B7, 0x201A, 0x201E, 0x2030, 0x00C2, 0x00CA, 0x00C1,
     0x00CB, 0x00C8, 0x00CD, 0x00CE, 0x00CF, 0x00CC, 0x00D3, 0x00D4,
     0xF7FF, 0x00D2, 0x00DA, 0x00DB, 0x00D9, 0x0131, 0x02C6, 0x02DC,
     0x00AF, 0x02D8, 0x02D9, 0x02DA, 0x00B8, 0x02DD, 0x02DB, 0x02C7
};
```

How To Specify No Connect/Listen Options (2/1/96)

Q: How do I specify that there are no options for the `Connect` and `Listen` methods of `protoBasicEndpoint`?

A: Different endpoint services use the options parameter differently. Some check for `nil` before attempting to access the array, while others assume they will always be passed an array of options. Some also assume that the array will always contain at least one element.

The correct work-around for this unspecified behaviour is to pass an array containing a single nil element. This works for all endpoint service types. For example:

```
ep:Connect([nil], nil);
```

Why Synchronous Comms Are Evil (2/1/96)

Q: Why does the following loop run slower and slower with each successive output? If the data variable contains a sufficiently large number of items, the endpoint times out or the Newton reboots before all the data is transmitted. For instance:

```
data := [....];
for item := 0 to Length(data) - 1 do
   ep:Output(data[ item ], nil, nil);
```

A: When `protoBasicEndpoint` performs a function synchronously, it creates a special kind of "sub-task" to perform the interprocess call to the comm tool task. The sub-task causes the main NewtonScript task to suspend execution until the sub-task receives the "operation completed" response from the comm tool task, at which time the sub-task returns control to the main NewtonScript task, and execution continues.

The sub-task, however, is not disposed of until control returns to the main NewtonScript event loop. In effect, each and every synchronous call is allocating memory and task execution time until control is returned to the main NewtonScript event loop! For a small number of sucessive synchronous operations, this is fine.

A fully asynchronous implementation, on the other hand, is faster, uses less machine resources, allows the user to interact at any point in the loop, and is generally very easy to implement. The above loop can be rewritten as follows:

```
ep.fData := [....];
ep.fIndex := 0;
ep.fOutSpec := {
   async:      true,
   completionScript:
      func(ep, options, error)
      if ep.fIndex >= Length(ep.fData) - 1 then
         // indicate we're done
      else
         ep:Output(ep.fData[ ep.fIndex := ep.fIndex + 1 ],
                   nil, ep.fOutSpec )
   };
ep:Output(ep.fData[ ep.fIndex ], nil, ep.fOutSpec );
```

Of course, you should always catch and handle any errors that may occur within the loop (`completionScript`) and exit gracefully. Such code is left as an excercise for the reader.

Maximum Speeds with the Serial Port (9/19/96)

Here are some rough estimates of the speeds attainable with the Newton serial port in combination with various kinds of flow control. These numbers are rough estimates, and depending on the protocol and amount of data (burst mode or not) you might get higher or lower transmission speeds. Experiment until you have found the optimal transmission speed.

- 0 to 38.4 Kbps

No handshaking necessary for short bursts, but long transmissions require flow control (either hardware or XON/XOFF).

- 38.4 Kbps to 115 Kbps

Require flow control, preferably hardware, but XON/XOFF should also work reasonably reliably.

- 115 Kbps +

You will encounter problems with latency and buffer sizes. Speeds in this range require an error correcting protocol.

Both hardware and `XON/XOFF` flow control can be set with the `kCMOInputFlowControlParms` and `kCMOOutputFlowControlParms` options. In the case of hardware handshaking `(RTS/CTS)` you should use the following options:

```
{  label:   kCMOInputFlowControlParms,
   type:    'option,
   opCode:  opSetRequired,
   data:    {  arglist: [
                  kDefaultXonChar,
                  kDefaultXoffChar,
                  NIL,
                  TRUE,
                  0,
                  0, ],
               typelist: ['struct,
                  'byte,
                  'byte,
                  'boolean,
                  'boolean,
                  'boolean,
                  'boolean,   ],
            },
},

{  label:   kCMOOutputFlowControlParms,
   type:    'option,
   opCode:  opSetRequired,
   data:    {  arglist: [
                  kDefaultXonChar,
                  kDefaultXoffChar,
                  NIL,
                  TRUE,
                  0,
                  0, ],
               typelist: ['struct,
                  'byte,
                  'byte,
                  'boolean,
                  'boolean,
                  'boolean,
                  'boolean,   ],
            },
}
```

XON/XOFF Software Flow Control Options Correction (9/19/96)

Q: XON/XOFF software flow control doesn't seem to work in `protoBasicEndpoint`'s asynchronous serial service. Why?

A: The endpoint option to specify flow control in the Newton Programmer's Guide is incorrect. The correct options are as follows:

```
{   label:      kCMOInputFlowControlParms,
    type:       'option,
    opCode:     opSetRequired,
    result:     nil,
    form:       'template,
    data:   {
        arglist:    [
            unicodeDC1,             // xonChar
            unicodeDC3,             // xoffChar
            true,                   // useSoftFlowControl
            nil,                    // useHardFlowControl
            0,                      // not needed; returned
            0,  ],                  // not needed; returned
        typelist:   ['struct,
            'byte,                  // XON character
            'byte,                  // XOFF character
            'boolean,               // software flow control
            'boolean,               // hardware flow control
            'boolean,               // hardware flow blocked
            'boolean,   ],  },  },  // software flow blocked

{   label:      kCMOOutputFlowControlParms,
    type:       'option,
    opCode:     opSetRequired,
    result:     nil,
    form:       'template,
    data:   {
        arglist:    [
            unicodeDC1,             // xonChar
            unicodeDC3,             // xoffChar
            true,                   // useSoftFlowControl
            nil,                    // useHardFlowControl
            0,                      // not needed; returned
            0,  ],                  // not needed; returned
        typelist:   ['struct,
            'byte,                  // XON character
            'byte,                  // XOFF character
            'boolean,               // software flow control
            'boolean,               // hardware flow control
            'boolean,               // hardware flow blocked
            'boolean,   ],  },  },  // software flow blocked
```

Why Are User Modem Settings Ingored (1/15/97)

Q: Our customers are complaining that modem preferences such as Ignore Dial Tone are getting ignored in our product. We are not doing anything special to set up the modem so why are the system settings ignored?

A: The user modem settings do not come for free. You must configure your endpoint based on the user settings. You can get these using the `MakeModemOption` call.

In general, we recommend that you always use `MakeModemOption` when setting up options to intiailize an endpoint. So you would call `MakeModemOption` to get your initial option array, then add your own custom options after that. `MakeModemOption` will return correct options based on the user settings for ignore DialTone, use PC Card Modem, etc.

Handling a -36006 Error When Disconnecting (1/17/97)

Q: Sometimes -36006 is thrown when I call my endpoint's disconnect method. What is happening?

A: This error will occur when Disconnect is called on a dropped connection. In fact, any time the endpoint state does not match the expected state of the calling method, a -36006 exception will be thrown.

To work around this problem, include an EventHandler method in your endpoint. When the connection drops, the EventHandler will be called and passed an event with an eventCode of 2. Simply add a delayed call to unbind and dispose of your endpoint. Do not use a deferred call to unbind and dispose of your endpoint: a bug in the deferred call mechanism can cause unpredictable results with communications code.

InputSpec Input Form 'Frame or 'Binary Buffer Bug (1/22/97)

Q: I have an `inputSpec` of form `'string`. When its `inputScript` triggers, I switch to an input form of `'binary`. When the binary `inputScript` triggers, the first few bytes of the data are garbage, and sometimes the `inputScript` doesn't trigger at all. The same behavior occurs when switching to the `'frame` input form. Why?

A: Binary and frame (B/F) input forms do not buffer incoming data the same way other input forms do. For maximum performance, the data is written directly into the destination object, rather than into an intermediate NewtonScript buffer for `endSequence` and `filter` processing.

Unfortunately, all data that has been buffered using a non-B/F input form is lost when switching to a B/F input form, resulting in corrupted data at the start of input, incorrect `byteCount`, or end-of-packet (`EOP`) detection failure.

The only workaround for this problem is to have the sender wait until the receiver has switched input forms and has flushed the input buffers before sending the binary data. In other words:

1. receive data using a non-B/F input form
2. flush the input buffer
3. switch to a B/F input form
4. tell the sender you're ready to receive more data

5. receive data

NEW: How to Debug Communication Endpoint Code (3/21/97)

Q: Is there any way I can use the NTK Inspector while running communications code? How do I debug my endpoint code?

A: If you are using a serial or MNP serial endpoint, you can use a serial PC Card to do your comms, freeing the standard serial port for the NTK Inspector. If you are using a serial or MNP serial endpoint, you must also modify your endpoint's instantiate options to use a PCMCIA slot instead of the built-in serial port. Here is the option you should add directly after the endpoint service option:

```
{
   type:          'option,
   label:          kCMOSerialHWChipLoc,
   opCode:         opSetRequired,
   form:           'template,
   result:         nil,
   data:           {
      argList:  [kHWLocPCMCIASlot1, 0],       // or kHWLocPCMCIASlot2
      typeList: ['struct, ['array, 'char, 4], 'uLong]
     }
}
```

If you are using Newton Internet Enabler (NIE) endpoints, you can use a PC Card Modem instead of a serial PC Card, but you do not have to add any special endpoint options. NIE will handle this automatically, provided you correctly set up your modem in the Modem preferences in the Prefs application.

This should allow your endpoint code to use the PC Card (serial card or modem card) instead of the built-in serial port. Connect the NTK Inspector to the built-in serial port as you normally would. If you are using an AppleTalk endpoint, you can simultaneously use the NTK inspector connected via AppleTalk.

NEW: XOn/XOff Software Flow Control Problem (4/3/97)

Q: XOn/XOff software flow control isn't working. I've included the documented `kCMOInputFlowControlParms` and `kCMOOutputFlowControlParms` options in my endpoint. What could I be doing wrong?

A: A quirk in the way Unicode characters are packed into `'char` fields in the endpoint option is preventing the correct flow control characters from being set in the serial driver. The solution is to use the `'byte` symbol rather than the `'char` symbol for these fields, thus avoiding the Unicode-to-ASCII conversion that would normally take place. The correct option frames are as follows:

```
{ label:    kCMOInputFlowControlParms,
  type:     'option,
  opCode:   opSetRequired,
  result:   nil,
  form:     'template,
```

```
  data: {
    arglist:  [
      unicodeDC1,           // xonChar
      unicodeDC3,           // xoffChar
      true,                 // useSoftFlowControl
      nil,                  // useHardFlowControl
      0,                    // not needed; returned
      0,  ],                // not needed; returned
    typelist: ['struct,
      'byte,                // XON character
      'byte,                // XOFF character
      'boolean,             // software flow control
      'boolean,             // hardware flow control
      'boolean,             // hardware flow blocked
      'boolean, ],  },  },  // software flow blocked

{ label:    kCMOOutputFlowControlParms,
  type:     'option,
  opCode:   opSetRequired,
  result:   nil,
  form:     'template,
  data: {
    arglist:  [
      unicodeDC1,           // xonChar
      unicodeDC3,           // xoffChar
      true,                 // useSoftFlowControl
      nil,                  // useHardFlowControl
      0,                    // not needed; returned
      0,  ],                // not needed; returned
    typelist: ['struct,
      'byte,                // XON character
      'byte,                // XOFF character
      'boolean,             // software flow control
      'boolean,             // hardware flow control
      'boolean,             // hardware flow blocked
      'boolean, ],  },  },  // software flow blocked
```

CHANGED: Sharp IR Protocol (4/9/97)

1 Serial Chip Settings

```
Baud rate    9600
Data bits    8
Stop bits    1
Parity       Odd
```

2 Hardware Restrictions

The IR hardware used in the Sharp Wizard series (as well as Newtons and other devices) requires a brief stablizing period when switching from transmitting mode to receiving mode. Specifically, it is not possible to receive data for two milliseconds after transmitting. Therefore, all devices should wait three milliseconds after completion of a receive before transmitting.

3 Packet Structure

There are two kinds of Packets: "Packet I" and "Packet II". Because the IR unit is unstable at the start of a data transmission, `DUMMY (5 bytes of null code (0x00))` and `START ID (0x96)` begin both packet types. At least two null bytes must be processed by the receiver as `DUMMY` before the `START ID` of a packet is considered. After this `(DUMMY, START ID)` sequence the `PACKET ID` is transmitted. Code `0x82` is the packet ID for a PACKET I transmission, and code `0x81` is the packet ID for a PACKET II transmission.

3.1 Packet I

This packet type is used to transmit the following control messages:

3.1.1	Request to send	ENQ (0x05)
3.1.2	Clear to send	SYN (0x16)
3.1.3	Completion of receiving data	ACK (0x06)
3.1.4	Failed to receive data	NAK (0x15)
3.1.5	Interruption of receiving data	CAN (0x18)

The format of this packet type is as follows:

	Byte length	Set value in transmission	Detection method in reception
DUMMY	5	0x00 * 5	Only 2 bytes are detected when received.
START ID	1	0x96	
PACKET ID	1	0x82	
DATA	1	above mentioned data	

Packet I example:

DUMMY	START ID	PACKET ID	DATA
0x00, 0x00, 0x00, 0x00	0x96	0x82	0x05

3.2 Packet II

This packet type is used to transmit data. The maximum amount of data that may be transmitted in one packet is 512 bytes. If more than 512 bytes are to be transmitted, they are sent as several consecutive 512-byte packets. The last packet need not be padded if it is less than 512 bytes and is distinguished by a `BLOCK NO` value of `0xFFFF`.

The format of this packet type is as follows:

	Byte length	Set value in transmission	Detection method in reception
DUMMY	5	0x00 * 5	Only 2 bytes are detected.
START ID	1	0x96	
PACKET ID	1	0x81	
VERSION	1	0x10	Judge only bits 7-4
BLOCK NO	2 (L/H)	0x0001 ~ 0xFFFF	
CTRL CODE	1	0x01	Don't judge
DEV CODE	1	0x40	Don't judge
ID CODE	1	0xFE	Don't judge
DLENGTH	2 (L/H)	0x0001 ~ 0x0200	
DATA	1 ~ 512		
CHKSUM	2 (L/H)		

BLOCK NO in last block must be set to 0xFFFF.

CHKSUM is the two-byte sum of all of the data bytes of DATA where any overflow or carry is discarded immediately.

Send all two-byte integers lower byte first and upper byte second.

Packet II example:

```
DUMMY                        START ID    PACKET ID    VERSION    BLOCK
NO      CTRL CODE
0x00, 0x00, 0x00, 0x00     0x96        0x81         0x10       Low
High    0x01

DEV CODE    ID CODE    DLENGTH            data    CHECKSUM
0x40        0xFE       Low        High    ????    Low         High
```

4 Protocol

Data will be divided into several blocks of up to 512 bytes each. These blocks are transmitted using type I and II packets as follows:

4.1 Transmission Protocol

4.1.1 The initiating device (A) begins a session by sending an ENQ (type I) packet. The receiving device (B) will acknowledge the ENQ by transmitting a SYN packet.

4.1.2 When (A) receives a SYN packet, it goes to step 4.1.4 below.

4.1.3 When (A) receives a CAN packet, or when 6 minutes have elapsed without a SYN packet reply to an ENQ packet, (A) terminates the session. If (A) receives any other packet, no packet, or an incomplete packet, it begins sending ENQ packets every 0.5 seconds.

4.1.4 When (A) receives a SYN packet, it transmits a single type II data packet, then awaits an ACK packet from (B).

4.1.5 When (A) receives an ACK packet, the transmission is considered successful.

4.1.6 If no ACK packet is received within 1 second from completion of step 4.1.4, or if any other packet is received, (A) goes to step 4.1.1 and transmits the data again. Retransmission is attempted once. The session is terminated if the second transmission is unsuccessful.

4.2 Reception Protocol

4.2.1 The receiving device (B) begins a session by waiting for an ENQ (type I) packet. If no ENQ packet is received after 6 minutes (B) terminates the session.

4.2.2 When (B) receives an ENQ packet, (B) transmits either a SYN packet to continue the session or a CAN packet to terminate the session.

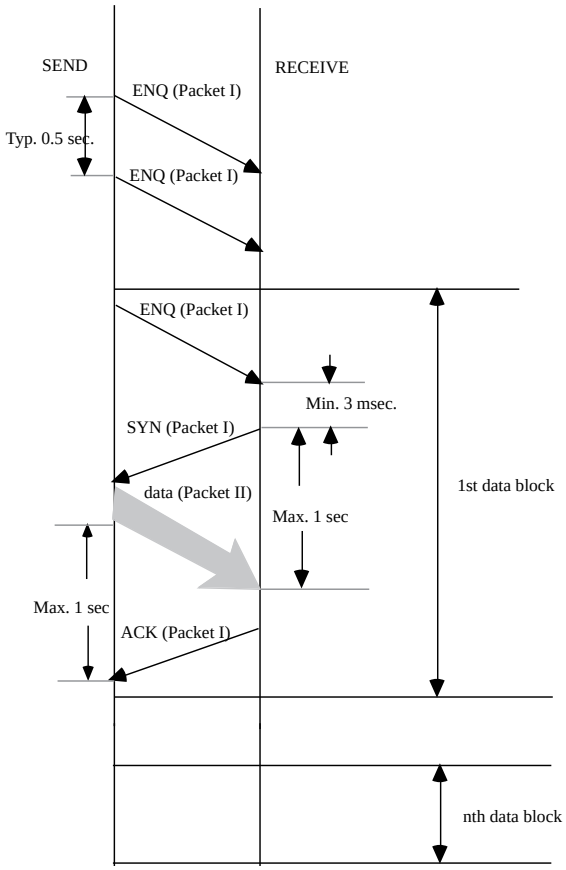
4.2.3 When (B) receives a valid type II packet (for example, the checksum and all header fields appear to be correct), (B) transmits an ACK packet.

4.2.4 If one or more header fields of the data packet are not correct, or if the time between data bytes is more than 1 second, (B) goes to step 4.2.1 and does not transmit the ACK packet (this will cause (A) to retransmit the packet after a one second delay).

4.2.5 If the header fields of the data packet appear to be correct but the checksum is incorrect, (B) transmits a NAK packet (this will cause (A) to retransmit the packet immediately).

Because of the restriction in hardware mentioned in item 2 above, it is not possible to receive data for two milliseconds after a data transmission. Please wait three milliseconds before transmitting a response to the other device.

(see diagram)



Hardware & OS

IR Port Hardware Specs (6/15/94)

Q: What are the hardware specifications for the Newton IR port?

A: In the Apple MessagePad 100, 110, and 120, the Sharp ExpertPad, and the Motorola Marco, the IR transmitter/receiver is a Sharp Infrared Data Communication Unit model RY5BD11 connected to channel B of a Zilog 85C30 SCC. Data is communicated along a 500 KHz carrier frequency at 9600 or 19200 baud, 8 data bits, 1 stop bit, odd parity. The IR hardware

requires a minimum of 5 milliseconds settling time when transitioning between sending and receiving. Sharp's CE-IR2 wireless interface unit may be used to connect the Newton to MacOS or DOS machines, with the appropriate software.

The Newton supports four IR software data modes:
Sharp encoding, NewtIR protocol (specifications are NOT releaseable)
Sharp encoding, SharpIR protocol
Plain Serial
38 KHz encoding ("TV Remote Control")

Serial Cable Specs (8/9/94)

Q: I want to make my own serial cable. Which wires and which connector pins do I use?

A: To create a hardware flow control capable cable for Mac-to-Newton or Newton-to-Newton communications (also called a "null-modem" cable) all you need are two mini-din-8 connectors and seven wires connected as follows:

```
Ground (4) -> Ground (4)  (also connect to connectors' shrouds)
Transmit+ (6) -> Receive+ (8)
Transmit- (3) -> Receive- (5)
Receive+ (8) -> Transmit+ (6)
Receive- (5) -> Transmit- (3)
Data Term Ready (1) -> Clear To Send (2)
Clear To Send (2) -> Data Term Ready (1)
```

You should use twisted pairs for 6/3, 8/5, and 1/2, to improve signal quality and reduce attenuation, especially in long cables. You can use side-by-side pairs, as in telephone hookup cable, for short cable runs.

Remember that because RS-422 uses a differential signal for transmit and receive, you always need two transmit and two receive pairs, and a break of either wire will cause communications in that direction to fail. The advantage, however, is significantly longer and more reliable cable runs than RS-232.

If you don't use hardware flow control, you can eliminate the 1/2 pair, but that's not recommended unless you know this cable will be used only in software flow control situations.

Q: What's the pin mapping on the Newton-to-PC (DIN-to-DB9) cable?

A: Here it is:

Note that the pin numbers shown are as defined above.

PC (DB9)	Newton (DIN)
1	1
2	3
3	5
4	7,2
5	4,8
6	1

```
7           N/C
8           N/C
9           N/C

N/C=not connected.
```

IR Hardware Info (9/6/94)

Q: How does the Newton send "Remote Control" codes?

A: This information is hardware dependent, and is only valid for the Original Message Pad, Message Pad 100, and Message Pad 110 products.

The IR transmitter/reciever is a Sharp IR Data Communication Unit connected to the second channel of a built-in SCC. When in "Remote Control" mode, the SCC is not used. Instead, a carrier frequency of 38KHz is transmitted, and the CPU toggles a register to generate the data pattern.

How Much Power Can a PCMCIA Card Draw (3/31/95)

Q: How much power can I draw through the PCMCIA slot?

A: The current rating depends on which Newton you are using and the type of batteries in use. Alkaline batteries provide less current than NiCad due to higher internal resistance. There is also a 'semi' artifical limit in the ROM. Currently any card who's CIS indicates more than 200 mA current draw will be rejected by the CardHandler. Other than that, here's the run down by hardware:

```
   Apple MessagePad 100:     50 mA
   Apple MessagePad 110:   ~160 mA
   Apple MessagePad 120:   ~300 mA
   Apple MessagePad 130:   ~300 mA (with backlight off)
   Apple MessagePad 130:   (with backlight on, the maximum has not
been characterized)
```

Do-it-Yourself Package Installation (8/26/96)

Q: I want to have a newer version of my package downloaded over an endpoint, and replace the older version. How do I do this?

A: There are a few steps, but they're fairly straightforward.

First, you need to remove the old package if the new version has the same unique name. (See code below which you can use if you don't know whether the new package has the same name or not.) Then call `SafeRemovePackage()`.

Second, you need to get the new package to the Newton device. Use the endpoint method `SuckPackageFromEndpoint()`, or the store method `SuckPackageFromBinary()` depending on where the package is coming from.

In some cases, you don't want to remove the old package until you're sure the new one works. If you're in this situation, the new package will have to have a different unique name. Just defer the call to `SafeRemovePackage` until after you verify (most likely with a deferred call) that the `SuckPackageFromEndpoint` or `SuckPackageFromBinary` has succeeded.

Also note that you can't call `SafeRemovePackage` from a function that's in the target package. You'll need to create a small function which does nothing but remove the old package, and then TotalClone that small function before executing it via a deferred call. Otherwise you'd be chopping your package's legs out from under itself, causing no end of havoc!

In some cases, it is appropriate to have a "loader" package which has a small amount of code to check whether or not to install the real package. This is accomplished by writing a small auto part which has the "Auto Remove Package" flag turned on, and the real package in a binary object within itself. This auto part `installscript` performs whatever checks are necessary, and then conditionally calls `SuckPackageFromBinary`, providing the binary object which holds the real package.

To create a binary object from a package, you need to move the data from the `.pkg` file that NTK produces into an object in the NewtonScript environment in NTK. On Windows NTK, `LoadDataFile` does this. On Macintosh NTK, the easiest thing to do is use a utility such as Clipboard Magician to copy the data from the `.pkg` file into a resource, then use `GetNamedResource` to get the data in your installer package. `GetNamedResource` and `LoadDataFile` are documented in the Newton Toolkit User's Guide. The MonacoTest sample code is a working example of a package installer that uses this technique.

To get the unique name for a package inside a binary object, you can use the following NewtonScript code. It takes the package object as its argument, and will return the string holding the unique name.

```
func(pkgRef)
begin
 local thelen:=extractword(pkgRef,26) div 2 -1;
 local s:="             ";

 while strlen(s)<thelen do
   s:=s&s;
  s:=substr(s,0,thelen);

 BinaryMunger(s, 0, thelen*2, pkgRef,
      52+(extractlong(pkgRef,48)*32)+extractword(pkgRef,24),
thelen*2);
 s;
end
```

Q: What are the hardware specifications for the serial port?

A: In the Apple MessagePad 100, 110, 120, 130, 2000, the eMate 300, the Sharp ExpertPad, and the Motorola Marco, the serial port is an EIA standard RS-422 port with the following pinout (as viewed looking at the female Mini-DIN-8 socket on the side of the Newton device, or looking at the female Mini-DIN-9 on the Newton Serial Adapter):

<see diagram>

```
Pin 1 HSKo  /DTR
Pin 2 HSKi  /CTS
Pin 3 TxD-  /TD
Pin 4 GND   Signal ground connected to both logic and chassis ground.
Pin 5 RxD-  /RD
Pin 6 TxD+  (see below)
Pin 7 GPi   General purpose input received at SCC's DCD pin.
Pin 8 RxD+  (see below)
Pin 9 Power out - 5V/100ma.  This pin only exists on the eMate 300
and the Newton Serial Adapter.
```

All inputs are:
Ri 12K ohms
minimum Vih 0.2v, Vil -0.2V
maximum tolerance Vih 15V, Vil -15V

All outputs are:
Rl 450 ohms
minimum Voh 3.6V, Vol -3.6V
maximum Voh 5.5V, Vol -5.5V

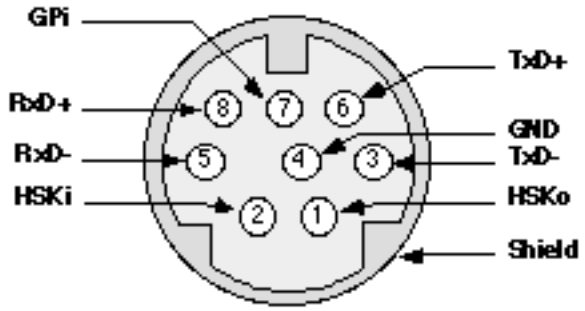
No more than 40mA total can be drawn from all pins on the serial port. Pins 3 & 6 tri-state when SCC's /RTS is not asserted.

The EIA RS-422 standard modulates its data signal against an inverted (negative) copy of the same signal on another wire (twisted pairs 3/6 & 5/8 above). This differential signal is compatable with older RS-232 standards by converting to EIA standard RS-423, which involves grounding the positive side of the RS-422 receiver, and leaving the positive side of the RS-422 transmitter unconnected. Doing so, however, limits the usable cable distance to approximately 50 feet, and is somewhat less reliable.

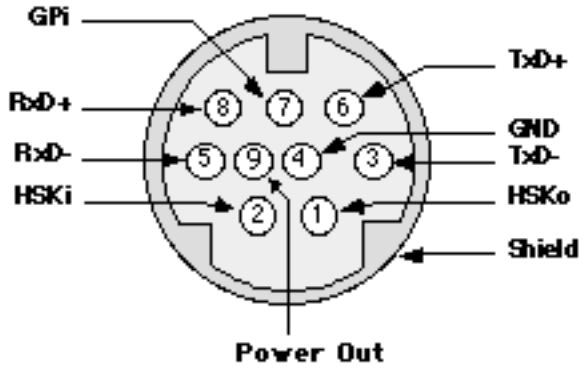
The MessagePad 120 and the MessagePad 130 use a Linear Technology LTC902 serial line driver. This part drives +5/-5 nominally for the RS422 signals, and you can use just one half to interconnect with RS232 compatible signal levels.

The MessagePad 2000 and the eMate 300 use a Linear Technology LTC1323 serial line driver. This part drives +5/-5 nominally for the RS422 signals, and you can use just one half to interconnect with RS232 compatible signal levels.

MessagePad 100, 110, 120, 130, Sharp ExpertPad, Motorola Marco



Newton Serial Adapter and eMate 300



Localization

StringToDateFrame & StringToTime Don't Use Seconds (5/9/96)

Q: When passed a string with seconds, for example "12:23:34", `StringToDateFrame` and `StringToTime` don't seem to work. `StringToDateFrame` returns a frame with NIL for all the time & day slots, and `StringToTime` returns NIL.

A: To correctly handle strings with seconds, seconds must be stripped from the string. If the application might be used outside the US, check for the Locale time delimiter. Here is a function which prepares a string for `StringToDateFrame` and `StringToTime`:

```
PrepareStringForDateTime := func (str)
begin   // str is just a time string, nothing else belongs
   local newStr := clone (str);
   local tf:= GetLocale().timeFormat;
   local startMin := StrPos (str, tf.timeSepStr1, 0);
   local startSec := StrPos (str, tf.timeSepStr2, startMin+1);
   // If a time seperator for seconds, then strip out seconds
   if startSec then
   begin
         local skipSecSep := startSec + StrLen (tf.timeSepStr2);
         local remainderStr := SubStr (
            str, skipSecSep, StrLen (str) - skipSecSep);
         local appendStr := StringFilter (
            remainderStr, "1234567890", 'rejectBeginning);
         newStr := SubStr (str, 0, startSec) & appendStr;
```

```
    end;
    return newStr;
end;
```

NEW: How GetDateStringSpec Uses Its Element Array (3/31/97)

Q: It appears that the compile-time function `GetDateStringSpec` formats the supplied date elements in reverse order. Is this is a bug?

A: This is the defined behavior. The Newton Programmer's Guide implies that the order does not matter, but that is not correct. `GetDateStringSpec` uses the elements in reverse order, although some functions that use dateStringSpecs may not observe the order defined by the dateStringSpec. For instance, some functions may use the elements of the dateStringSpec, but use the element ordering defined by the locale bundle.

For instance, you can use this call to define a dateStringSpec for use in a locale bundle with the order Day/Month/Year:

```
GetDateStringSpec([[kElementYear, kFormatNumeric],[kElementMonth,
   kFormatNumeric],[kElementDay, kFormatNumeric] ]);
```

Miscellaneous

Unicode Character Information (9/15/93)

Q: Where can I find more about Unicode tables?

A: The following book provides a full listing of the world wide (non-Kanji) Unicode characters:

The Unicode Standard
WorldWide Character Encoding
Version 1.0 Volume 1
ISBN-0-201-56788-1

Current Versions of MessagePad Devices (5/15/96)

Q: What are the versions of the Apple Newton MessagePad device?

A: This answer will change as product versions are released. To find the version number, open the Extras Drawer. In the Newton 1.x OS, open the Prefs application and look at the number in the bottom middle of the screen. In the Newton 2.0 OS, choose Memory Info from the Info button.

As of Sep 25, 1996 the latest versions are:

English Newton 2.0 OS

MessagePad 120	2.0 (516205)
MessagePad 130	2.0 (526205)

German Newton 2.0 OS

MessagePad 120	D-2.0 (536030)

English Newton 1.x OS

MessagePad	1.05
MessagePad	1.11
MessagePad 100	1.3 (415333)
MessagePad 110	1.3 (345333)
MessagePad 120	1.3 (465333)

German Newton 1.x

MessagePad	D 1.11
MessagePad 100	D 1.3 (435334)
MessagePad 120	D 1.3 (435334)

French Newton 1.x

MessagePad 100	F 1.3 (424112)
MessagePad 110	F 1.3 (424112)
MessagePad 120	F 1.3 (455334)

NEW: Using the Icon Editor in NTK 1.6.4 (4/18/97)

Q: In the icon editor in the Project Settings slip in NTK 1.6.4 for MacOS, I don't always see all the images. Once I choose an image in any depth, the list is much shorter for the other depths, and I can't find the image I need! What's wrong?

A: All of the images in a multiple-depth icon must be exactly the same size. To help ensure that this is the case, once any image is selected in any depth, NTK limits the available choices in the other depths to images that are exactly the same size. Images from any resource files that match in size will be shown, and images of different sizes will not appear. You must edit your images "by hand" to ensure that all the images you want for your icon family are exactly the same size, padding smaller images with white pixels as necessary.

NewtApp

Creating Preferences in a NewtApp-based Application (1/31/96)

Q: How do I create and use my own preferences slip in a NewtApp-based application?

A: In your application's base view create a slot called `prefsView` and place a reference to the template for your preferences slip there (probably using the NTK `GetLayout` function.) When the user selects "Prefs" from the Info button in your application, the NewtApp framework will create and open a view based on the template in the `prefsView` slot.

When your preferences view opens, a reference to your application's base view is stored in a slot called `theApp` in the preferences view. Use this reference to call the application's `GetAppPreferences` method. This method will return a frame containing your application's preferences. `GetAppPreferences` is a method provided by NewtApp and should not be overidden.

When adding slots to the preferences frame, you must either append your developer signature to the name of the preference (for example, `'|Pref1:SIG|`) or create a slot in the preferences frame using your developer signature and save all preferences in that frame. This will guarantee that you don't overwrite slots used by the NewtApp framework.

Here is an example of how to get the preferences frame and add your data:

```
preferencesSlip.viewSetupFormScript := func()
begin
   prefs := theApp:GetAppPreferences();
   if NOT HasSlot(prefs, kAppSymbol) then
      prefs.(kAppSymbol) := {myPref1: nil, myPref2: nil};
end;
```

To save the preferences, call the application's `SaveAppState` method:

```
preferencesSlip.viewQuitScript := func()
   theApp:SaveAppState(); // save prefs
```

NewtApp currently provides one built-in preference for where to save new items. In the preferences frame there will be a slot called `internalStore`. Setting this slot to `true` will force the NewtApp framework to save all new items on the internal store.

Creating an About Slip in a NewtApp-based Application (1/31/96)

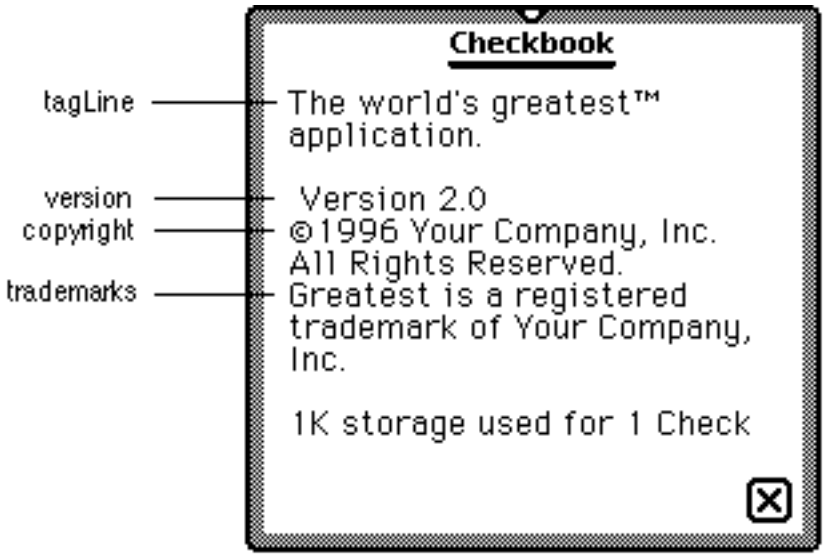
Q: How do I create my own About slip in a NewtApp-based application?

A: Depending on how much control you want, there are two ways to do this. For the least amount of control, create a slot in your application's base view called `aboutInfo`. Place a frame in that slot with the following slots:

```
{tagLine:  "",    // A tagline for your application
 version: "",     // The version number for the application
 copyright: "",   // Copyright information
 trademarks: "",  // Trademark information
}
```

The information found in this frame will be displayed by the NewtApp framework when the user selects "About" from the Info button's popup. See the picture below for an example of what the user will see.

Alternatively, you can create your own About view. If you do this, create a slot in your application's base view called `aboutView` containing a reference to a template for your about view (probably using the NTK `GetLayout` function.) A view will be created from that template and opened when the user selects "About" from the Info button's popup.



NewtSoup FillNewSoup Uses Only Internal Store (2/5/96)

Q: My `NewtSoup` continues to get the `FillNewSoup` message, even when the soup already exists. Am I doing something wrong?

A: The NewtApp framework only checks for entries on the internal store to determine if the `FillNewSoup` message needs to be sent. Check the "Setting the UserVisible Name With NewtSoup" Q&A for more details and a description of how to work around the problem.

Setting the User Visible Name With NewtSoup (2/6/96)

Q: How can I make the user visible name for my NewtApp's soup be something besides the internal soup name, as I can do with `RegUnionSoup`?

A: There is a method of `newtSoup` called `MakeSoup` which you can override. The `MakeSoup` method is responsible for calling `RegUnionSoup` (or otherwise making a soup) and then calling the `FillNewSoup` method if the soup is new/empty.

`MakeSoup` is called normally as part of initializing the `newtSoup` object. Here is a sample `MakeSoup` method that will use a newly defined slot (from the `newtSoup` based template) for the user name.

The current documentation doesn't tell you everything you need to do to properly override the `MakeSoup` method. In particular, `MakeSoup` is used by the `newtSoup` implementation to initialize the object, so it needs to set up other internal slots. It's vital that the `'appSymbol` slot in the message context be set to the passed argument, and that the `'theSoup` slot be set to the soup or unionSoup that `MakeSoup` creates or gets. (Recall that `RegUnionSoup` returns the union soup, whether it previously existed or not.)

The `GetSoupList` method of union soups used in this code snippet returns an array with the member soups. It should be considered documented and supported. A newly created union will have no members, so `FillNewSoup` should be called. This is an improvement

over the default `MakeSoup` method, which always calls `FillNewSoup` if the soup on the internal store is empty.

The user visible name is supplied via the newtSoup `'userName` slot, which is looked up in the current context. As with `soupName`, `soupDescr`, etc, you should set a new `userName` slot in the frame in the `allSoups` frame in the `newtApplication` template.

```
MakeSoup: func(appSymbol)
   begin
      self.appSymbol := appSymbol;    // just do it...
      self.theSoup := RegUnionSoup(appSymbol, {
         name: soupName,
         userName: userName,
         ownerApp: appSymbol,
         userDescr: soupDescr,
         indexes: soupIndices,
      });
      if Length(theSoup:GetSoupList()) = 0 then
         :FillNewSoup();
   end;
```

How to Control Sort Order in NewtApp (5/10/96)

Q: While a NewtApp application is running, can I change the order in which soup items appear?

A: Yes, the key to changing the sort order is to modify the query spec in the `allSoups` frame, and then cause the application to refresh. The cursor that controls the sort order for the layout is built from the `masterSoupSlot` slot. Both the default and the overview layouts have a `masterSoupSlot` which points back to the relevant `allSoups` slot in the app base view.

Here are the basic steps:

1) Ensure `newtAppBase.allSoups` & `newtAppBase.allSoups.mySoup` are writeable. (Since the frames reside in the package, they are in protected memory.)
2) Modify the query spec to the new sort order.
3) Now send `newtAppBase.allSoups.mySoup:SetupCursor()` to create a new cursor using the new query spec.
4) Then do a `newtAppBase:RedoChildren()` to display the items in the new sort order.

The code would look something like:

```
if IsReadOnly (newtAppBase.allSoups) then
  newtAppBase.allSoups := {_proto: newtAppBase.allSoups};
if IsReadOnly (newtAppBase.allSoups.mySoup) then
   newtAppBase.allSoups.mySoup :={
      _proto: newtAppBase.allSoups.mySoup};
   newtAppBase.allSoups.mySoup.soupQuery :=
      {indexpath: newKey};       // new sort order!
   newtAppBase.allSoups.mySoup:SetupCursor();
   newtAppBase:RedoChildren();
```

How to Avoid NewtApp "Please insert the card" errors (5/10/96)

Q: If a NewtApp-based application is on a PC card and the card is removed, the user gets the following error message:

"The package <package name> still needs the card you removed. Please insert it now, or information on the card may be damaged."

How can I avoid this problem?

A: While a card is unmounting, if an object on the card is still referenced, then the user will get the above error message asking them to reinsert the card. For more information about issues for applications running from a PC card see the article "The Newton Still Needs the Card You Removed"

The `newtApplication` method `NewtInstallScript` is normally called in the part's `InstallScript` function. One thing the `NewtInstallScript` does is register the viewDefs in the NewtApp base view allViewDefs slot using the global function `RegisterViewDef`.

Currently, `RegisterViewDef` requires that the data definition symbol be internal. If the symbol is on the card, then when the `NewtRemoveScript` tries to unregister the viewDef a reference to data on the card is encountered and the above error message will be shown. This bug will be fixed in a future ROM.

To work around this bug for any 2.0 based ROM, add the following code to your part's `InstallScript` before calling `NewtInstallScript`:

```
local mainLayout := partFrame.theForm;
if mainLayout.allViewDefs then
   foreach dataDefSym,viewDefsFrame in mainLayout.allViewDefs do
      foreach viewDef in viewDefsFrame do
         RegisterViewDef (
            viewDef, EnsureInternal (dataDefSym) );
partFrame.removeFrame :=
   mainLayout:NewtInstallScript(mainLayout);
```

Note that it is OK to call `RegisterViewDef` more than once with the same view definition. `RegisterViewDef` will do nothing (and return NIL) if the template is already registered.

Customizing Filters with Labelled Input Lines (9/4/96)

Q: I need to open a slot view on a slot that isn't a standard data type (int, string, etc). How do I translate the data from the soup format to and from a string?

A: Here is some interim documentation on the filter objects that newtLabelInputLines (and their variants) use to accomplish their work.

A filter is an object, specified in the 'flavor slot of the newtLabelInputLine set of protos, which acts as a translator between the target data frame (or more typically a slot in that frame) and the text field which is visible to the user. For example, it's the filter for

newtDateInputLines which translates the time-in-minutes value to a string for display, and translates the string into a time-in-minutes for the target data.

You can create your own custom filters by protoing to newtFilter or one of the other specialized filters described in Chapter 4 of the Newton Programmer's Guide.

When a newtLabelInputLine is opened, a new filter object is instantiated from the template found in the 'flavor slot for that input line. The instantiated filter can then be found in the `filter` slot of the view itself. The `_parent` slot of the instantiated filter will be set to the input line itself, which allows methods in the filter to get data from the current environment.

Here are the slots which are of interest. The first four are simply values that you specify which give you control over the recognition settings of the inputLine part of the field, and the rest are methods which you can override or call as appropriate.

Settings:
`recFlags`
Works like entryFlags in protoLableInputLine. This provides the 'viewFlags settings for the inputLine part of the proto -- the field the user interacts with.

`recTextFlags`
Provides the 'textFlags settings for the inputLine part of the proto.

`recConfig`
Provides the 'recConfig settings for the inputLine part of the proto.

`dictionaries`
Like the 'dictionaries slot used in recognition, Provides custom dictionaries if `vCustomDictionaries` is on in the `recFlags` slot.

Methods:
`PathToText()`
Called when the inputLine needs to be updated. The function should read data out of the appropriate slot in the 'target data frame (usually specified in the '`path` slot) and return a user-visible string form of that data. For example, for numbers the function might look like
`func() NumberStr(target.(path))`

`TextToPath(str)`
Called when the inputLine value changes. The result will be written into the appropriate slot in the 'target data frame. The string argument is the one the user has modified from the inputLine part of the proto. For example, for numbers the function might look like
`func(str) if StrFilled(str) then StringToNumber(str)`

`Picker()`
An optional function. If present, this method is called when the user taps on the label part of the item. It should create and display an appropriate picker for the data type. For the pre-defined filters, you may also wish to call this method to open the picker. You should store a reference to the filter in the picker view. Then if the user picks an item, send the filter instance a `PickActionScript` message. If the picker is cancelled, send a `PickCancelledScript` message.

NOTE: If this method is defined, a pick separator line and the text "Other..." will be added to the labelCommands array.

`PickActionScript( newValue )`
An optional function. This method should be called when the user selects something from the picker opened through the filter's `Picker` method. If you override this method be sure to call the inherited `PickActionScript` method.

`PickCancelledScript()`
An optional function. This method should be called when the user cancels the picker opened through the filter's `Picker` method. If you override this method be sure to call the inherited `PickCancelledScript` method.

`InitFilter()`
Optional. This method is called when an inputLine that uses this filter is first opened. This method can be used to get data from the current environment (for example, the 'path slot of the inputLine) and adjust other settings as appropriate.

Dynamically Changing the Height of Stationery (11/19/96)

Q: How can I dynamically change the height of my roll-style stationery?

A: To dynamically change the height of roll-style stationery you will need to change the target's `height` slot, flush the data, and then do a re-target. For instance, you might have the following method in your stationery:

```
DoResize: func( newHeight )
   begin
      target.height := newHeight;
      :FlushData();
      :DoRetarget();
    end;
```

Using Custom Help Books in a NewtApp-based Application (12/2/96)

Q: I have created a help book for my NewtApp-based application. Can I make my application open the help book when the user chooses "Help" from the info button?

A: Yes, there is a `newtApplication` slot called `helpManual`. You should store a reference to your help book in this slot.

There is also a slot called `viewHelpTopic` which you can use to dynamically change the location the help book is opened to. This slot should store the name of the topic to open to.

See the DTS Sample Code project "Beyond Help" for an example of a help book.

Creating a Large newtEditView/newtROEditView (12/2/96)

Q: When I use `newtEditView` or `newtROEditView`, I cannot scroll through all the text of a large note. After a few pages it stops scrolling. What is going wrong?

A: Both `newtEditView` and `newtROEditView` have a default scroll height of 2,000 pixels. To work around this limitation, you will need to add a slot called `noteSize` to your `newt(RO)editView`. This slot should hold an array of two elements. The first element is the scroll width. If you do not want horizontal scrolling, the scroll width should equal the view width. The second element is the `scrollHeight`.

Here is an example `noteSize` slot that you would use to create a `newt(RO)EditView` with a scroll height of 20,000 pixels.

```
{
   _proto:     newtEditView,
   noteSize:   [viewWidth, 20000],
   ...
}
```

How to Use ForceNewEntry with NewtApp (12/2/96)

Q: I have a `newtApp`-based application which does not use stationery. If I set the `forceNewEntry` slot to `nil` in my layout and open the application with a `nil` target, I can still see the entry view. How can I avoid this?

A: You will need to check for the existence of a target frame. If one does not exist then close the entry view.

You will not have this problem if you use stationery because the `newtApp` framework will not open the stationery if a target does not exist.

How to Programatically Open the Header Slip (1/3/97)

Q: I like the way the Newton Works application (for Newton 2.1 OS) automatically opens the header slip each time a new entry is created. Can I make my newtApp-based application do this?

A: Yes! The `newtEntry(Roll/Page)Header` proto has a `PopIt` method which opens the header. You will need to override the `StatScript` of your `newtNewStationeryButton` and send the header a `PopIt` message. Because `PopIt` is not defined prior to Newton 2.1 OS, you will need to check for its existence before calling it. Here is a code example:

```
newtNewStationeryButton.   StatScript: func( theStat )
      begin
         // Keep a copy of the inherited return value for use below
         local result := inherited:?StatScript( theStat );

         // Pass self as a parameter for the closure.
         // This gives us a reference
         // to the application so we can get the entry view.
         AddDeferredCall( func( context )
            begin
               local entryView := context:GetTargetView();
```

```
                // This code assumes that your header is declared to
                // the entry view with the name theHeaderView
                if entryView.theHeaderView.popIt then
                   entryView.theHeaderView:Popit( true );
             end,
          [self] );

          result;
       end;
```

Programatically Changing the Default ViewDef (1/3/97)

Q: I want to be able to programatically change which viewDef is shown when I tap the "New" button in my newtApp-based application. How can I do this?

A: By default, the viewDef which is shown when you create a new entry is the one with the value `'default` in its `symbol` slot. To change this behavior at run-time, you will need to override the `StatScript` method of your `newtNewStationeryButton`.

In the `StatScript` method, you will set two slots in your application. The first slot is the `preferredViewDef` slot in your application's base view. The second is the `viewDef` slot of the current layout. Both of these slots should be set to the symbol of the viewDef that you want displayed. For instance, you might have the following `StatScript`:

```
StatScript := func( theStat )
begin
   preferredViewDef := 'myNewDefaultStationery;
   layout.viewDef:= 'myNewDefaultStationery;

   // Make sure we call the inherited method
   inherited:?StatScript( theStat );
end;
```

Note: you must not modify either the application's `preferredViewDef` slot or the layout's `viewDef` slot at any other time. Doing so could cause your application to not work on future versions of the Newton OS.

How to Properly Declare NewtApp Views (1/6/97)

Q: I have a `newtEntryPageHeader` which is declared to my `newtLayout` view. Each time I change entries in my application, the header does not get properly updated. What's going wrong?

A: If you declare your `newtApplication` views, they need to be declared to their parent. Declaring `newtApplication` views to a grandparent can cause undefined behavior.

Because of how the declare mechanism works, you must be careful when you declare a view to a grandparent view. In some circumstances, you could try to access a view which has been closed.

As an example, pretend you have three views called viewA, viewB, and viewC. They have the following heirarchy.:

```
ViewA                (grandparent)
    ViewB            (parent)
        ViewC        (child)
```

ViewC is a child of viewB and viewB is a child of viewA; ViewC is declared to viewA. If you close viewB, viewC will also be closed because it is a child of viewB. Since ViewC was declared to ViewA, ViewA will still have a reference to viewC which has been closed. Sending view messages to viewC will throw.

For more information on the Newton OS declare mechanism, see the "Declaring Multiple Levels" Q&A, and the "The Inside Story on Declare" appendix in the Newton Programmer's Guide.

How to Create Custom Overviews with NewtApp (1/8/97)

Q: My NewtApp-based application can print successfully, but I want a custom format that can print multiple items on one page or handle different transports than the default overview supports. How do I do this?

A: Add an `overviewTargetClass` slot to your application (or any other layout that is a descendent of your `newtOverLayout`). Set this slot's value to be the symbol that represents your data class (for example, `'|myData:SIG|`), which must match the data class you use when registering your print format. The NewtApp overview will use `overviewTargetClass` instead of the default overview class (`'newtOverview`) supplied by `newtOverLayout`.

You must still register your print format, but you must set its `usesCursors` slot to `true` indicating that it will use the value target as a multiple item target and it will iterate over it using `GetTargetCursor(target)`. For an example of a print format that can handle multiple items, see the MultiRoute DTS sample.

For more information about the default overview class (`'newtOverview`), see the Q&A "Limitations with NewtOverview Data Class".

How to Store Prefs in a NewtApp-based Application (1/17/97)

Q: I want to save application-specific preferences and state information before my application is closed. What is the best way to do this?

A: You can save application-specific information in the frame in the `prefsCache` slot of your NewtApp-based application. This slot is defined in your application's base view by the NewtApp framework. This frame will be saved to the system soup when the application closes.

When you add to the `prefsCache` frame, you must use your registered signature to avoid conflicting with slots that the framework may use. You can name each of your preferences

with your signature, or we recommend adding a subframe in a slot named with your signature. For instance, you might have the following code:

```
prefsCache.('|MyPrefs:MySIG|) := {pref1: 1, pref2: 2};
```

NEW: A CheckAll Button for NewtApp Overviews (3/4/97)

Q: What do I have to do to get the Check All button to appear in my overview? What's the compatible way to do this so that the application works on Newton 2.0 OS as well?

A: In Newton 2.1 OS, there is a proto called `newtCheckAllButton` (`@872`) which you can use. This proto sends the `CheckAll` method to the layout. In Newton 2.1 OS, `newtOverLayouts` have two new methods, `CheckAll` and `UncheckAll`, which implement this behavior. However, none of this is present in Newton 2.0 OS .

To create a check all button that works on the Newton 2.0 OS, you will need to create the button yourself and implement the `CheckAll` and `UncheckAll` methods for your overview layout (or any other layout you wish to implement check all for.)

Older versions of the DTS sample code (either "Checkbook-7" or "WhoOwesWhom-3") do have a `protoCheckAllButton`. These samples implement an earlier (and less useful) flavor of Check All. The old samples check all the items which are currently visible in the overview, while the Newton 2.1 OS checks all the items that are present in the currently selected folder/card filter. "Checkbook-8" or "WhoOwesWhom" (version 3 or later) will reflect the Newton 2.1 behavior.

Until the updated samples are available, start with the `protoCheckAllButton` from the older sample code, since that gives the correct look and button bounds, and modify it as follows:

The check all button's `buttonClickScript` should look something like this:

```
func()
   if newtAppBase.currentLayout = 'overView then
      begin
         if layout.checkAllPrimed then
            layout:UnCheckAll()
         else
            layout:CheckAll();
         layout.checkAllPrimed := NOT layout.checkAllPrimed;
      end;
```

The overview layout's CheckAll and UncheckAll methods should look something like this:

```
CheckAll:
   func()
      begin
         local curse := dataCursor:Clone();
         curse:Reset();
         hilitedIndex := nil;
         selected := MapCursor(curse, func(e) MakeEntryAlias(e));
         AddUndoSend(layout, 'UnCheckAll, []);
```

```
            layout:DoRetarget();
        end;

    UncheckAll:
        func()
            begin
                hilitedIndex := nil;
                selected := nil;
                layout:DoRetarget();
            end
```

Note that these methods make use of two undocumented slots: `hilitedIndex` and `selected`. `hilitedIndex` is used internally by `newtOverLayout` to track the tapped item. You may set it to `NIL` (as above) to clear the value, but do not set it to some other value or rely on its current value. `selected` contains an array of aliases to soup entries representing the currently selected items, and will be used by the routing and filing buttons for processing entries. It is important to clear `hilitedIndex` when modifying the `selected` array in any way.

The resulting CheckAll button should be included in the `menuRightButtons` array for the status bar. The older sample code puts it on the left, however user interface discussions as part of the Newton 2.1 OS effort resulted in the decision to place the button on the right.

CHANGED: Creating a Simple NewtApp (4/7/97)

Q: What are the basic steps to create a simple NewtApp-based application?

A: The following steps will create a basic NewtApp-based application:

Basic Setup

1) Create a project.
2) In NTK's Project Settings dialog, set Platform to "Newton 2.0" or "Newton 2.1".

Create the NewtApp base view

1) Create a layout file.
2) Drag out a `newtApplication`.
3) Set the following slots to the following values:

```
allLayouts: {
        default: GetLayout("default.t"), // see step 9 in the next section
        overview:   GetLayout("Overview.t"),    // set step 4, overview
section
    }
allSoups:       {
    mySoup: {
        _proto: newtSoup,
        soupName: "SoupName:SIG",
        soupIndices: [],
        soupQuery: {}  } }
title: kAppName
```

4) Draw a `newtClockFolderTab` or `newtFolderTab` as a child of the `newtApp`.
5) Draw a `newtStatusBar` as a child of the `newtApp`.
6) For the `newtStatusBar` set the following slots:

```
menuLeftButtons:  [newtInfoButton]
```

```
menuRightButtons: [newtActionButton, newtFilingButton]
```
7) Save the layout file as "main.t" and add it to the project.

Create the default view:
1) Create another layout file.
2) Draw a newtLayout in the new layout file.
3) Add a viewJustify slot to the newtLayout and set it to parentRelativeFull horizontal and vertical.
4) Set the viewBounds of the newtLayout to:
```
{top: 20, // leave room for the folder tab
bottom: -25,  // leave room for the status bar
left: 0,
right: 0}
```
5) Draw a newtEntryView as a child of the newtLayout.
6) Add a viewJustify slot and set it to parentRelativeFull horizontal and vertical (necessary only until platform file is updated).
7) Set the viewBounds of the newtEntryView to:
```
{top: 0, bottom: 0, right: 0, left: 0};
```
8) Draw slot views as children of the entry view to display slots from the soup entry.

For example:
a) Draw a newtLabelInputLine as a child of the newtEntryView.
b) Set the following slots:
```
label:  "My Label"
path:   'myTextSlot
```
c) Draw a newtLabelNumInputLine as a child of the newtEntryView.
d) Set the following slots:
```
label:  "Number"
path:   'myNumberSlot
```

9) Save the layout file as "default.t" and add it to the project. Move it so that it is compiled before the main layout (use the Process Earlier menu item).

Add Overview support
1) Create another layout file.
2) Draw a newtOverLayout in the new layout file.
3) Add the Abstract slot to the newtOverLayout, for example:
```
Abstract := func(item, bbox )
begin
   local t := item.myTextSlot & ",";
   if item.myNumberSlot then
      t := t && NumberStr(item.myNumberSlot);
   MakeText(t, bbox.left+18, bbox.top,
      bbox.right, bbox.bottom - 18);
end;
```
4) Save the layout file as "overview.t" and add it to the project. Move it so that it is compiled before the main layout (use the Process Earlier menu item).

Add InstallScript and RemoveScript
1) Create a text file and add the following to it:
```
InstallScript := func(partFrame) begin
partFrame.removeFrame :=
   (partFrame.theForm):NewtInstallScript(partFrame.theForm);
end;

RemoveScript := func(partFrame) begin
```

```
    (partFrame.removeFrame):
  NewtRemoveScript(partFrame.removeFrame);
end;
```

2) Save the text file and add it to the project.

Newton C++ Tools

NEW: Packed Structures in C++ Tools (2/28/97)

Q: Do the Newton C++ Tools support a concept of packed structures? Some compilers provide a keyword (`packed`) to prevent aligning of the fields by the compiler. I tried to use the keyword and got an error. How can I ensure that a structure is packed?

A: While there is no supported way to do this in the current tools, the ARM compiler does have an experimental directive, `__packed`, which is probably worth a try. Using this directive may cause the compiler to stop with an internal error in some circumstances, so be prepared. You should ensure that the structure produced has the correct alignment by using the C `sizeof` and `offsetof` functions, since this directive does introduce a compiler dependency. Accessing elements in `__packed` structures can be considerably less efficient than using non-packed structures: use them only when necessary. For example:

```
__packed struct T { char c; int i; };
```

produces a structure that is 5 bytes wide. Without the `__packed` directive, it would be 8 bytes wide and the integer field would begin 4 bytes from the structure start, so that it was word aligned.

We believe the internal error in the compiler can be avoided by taking the sizeof the structure before using it. An easy way to do this is to add a dummy function right after the structure is declared. For example:

```
inline void dummyT() { (void)sizeof(T); }
```

Primitive types can also be declared `__packed`, which means that the compiler will not make assumptions about the alignment of pointers to them. That is, if you know an int starts two bytes into a word-aligned data structure, the wrong thing will happen if you simply cast the pointer to `int`. Instead, you can used an `unaligned int` type. This generates considerably less efficient code than is needed for working with aligned values, but it's still more efficient that trying to extract the proper bytes and shift/add them into an integer youself. For example:

```
typedef __packed int UNALIGNED_INT;
int IntAt(UNALIGNED_INT* p) { return *p; }
```

This directive does not work properly with bitfield specifiers. For example:

```
__packed struct Foo {
   unsigned flag1 : 1;
   unsigned flag2 : 1;
   unsigned data1 : 6;
   unsigned short data2;
}
```

will not produce what you expect. Instead, avoid the bitfield specifiers and take advantage of C++ inline functions to access the partial bytes:

```
__packed struct Foo {
   char stuff;
   unsigned short data2;
```

```
    int Flag1() { return (stuff & 0x80) != 0; }
    int Flag2() { return (stuff & 0x40) != 0; }
    int Data1() { return stuff & 0x3F; }
    int Data2() { return data2; }
};
inline void dummyFoo() { (void)sizeof(Foo); }
```

The result is a 3-byte wide data structure with the bitfields easily accessible.

Note that the ProtocolGen tool (part of the DDKs) does not understand the `__packed` directive. ProtocolGen does not make use of structure sizes, so it's OK to NOP out the `__packed` keyword for that tool. Here's an easy way to do that:

```
#ifdef PROTOCOLGEN
#define __packed
#endif
```

Newton ToolKit

NTK, Picture Slots and ROM PICTs (12/19/93)

Q: How can I use a PICT in ROM from a picture slot editor in NTK?

A: You must use an NTK `AfterScript` to set the appropriate slot in the view to point to the ROM based PICT (assuming that the constant for the PICT is defined in the NTK definitions file AND documented in the Newton Programmers Guide). Use something like this in the `AfterScript`:

```
thisView.icon := ROM_RouteDeleteIcon;
```

Recognition Problems with the Inspector Window Open (3/8/94)

Q: When I have the Inspector window open in NTK and I debug my application, recognition does not work properly and the Newton complains about lack of memory. However, when I disconnect the Inspector, recognition works fine. What is going on?

A: The NTK inspector window uses system memory on the Newton side; the Toolkit App itself makes use of MNP (a compression and error correction protocol) in the Newton, which uses a buffer shared with the recognition working memory.

Different releases of the Newton OS have different amounts of memory allocated for this shared area, so the problem may not be apparent on some units. However, if this happens you have several options:

- Disconnect the Inspector when testing the recognition side.
- Use the keyboard for text input while testing the code.
- Write shorter text items.

Accessing Views Between Layout Windows (6/7/94)

Q: I have problems setting a `protoStaticText` text slot that is in one linked layout window from a button that is in another linked layout window. I tried to allow access to the base view from both linked layouts, but this didn't help. I even tried to allow access from the base view to both layouts, but this didn't help, either. What should I do?

A: There is no way to declare views across the artifical boundary imposed by the linked layouts. Until this feature of NTK is implemented, you must either create the link yourself at run time, or declare the button to the top level of the linked layout, and then declare the link.

For example, consider a view called `textThatChanges` which a child of a view called `changingContainer` and is declared to `changingContainer` with the name `textThatChanges`. `ChangingContainer` is the base view for a layout which is linked into the main layout, and the link (in the main layout) is declared as `changingContainerLink`. Code in the main layout can change the text of the `textThatChange` view like so:

```
SetValue(containerLink.whatToDo, 'text, "Turn and face the...")
```

To do the equivalent of the declare yourself:

1) In the `viewSetupFormScript` script of the `'buttonThatChanges` button, set the value of the base view's slot `'theTextView` to `self`, as in the following code fragment:

```
func()
begin
   base.theTextView := self;
end
```

2) In the `buttonClickScript` script of the `'buttonThatSetsText` button, use the global function `SetValue` to store new text in the text slot of the `'buttonThatChanges` button, as in the following code fragment:

```
func()
begin
   SetValue(base.theTextView, 'text, "Now something happened!");
end
```

Note that this example assumes the self-declared view called `base`. In your application, you may access your base view in a different way.

Dangers of StrCompare, StrEqual at Compile Time (6/9/94)

Q: I've noticed that `StrCompare` can return different results at compile time than it does at run time. What gives?

A: While most functions documented in the NewtonScript Reference are available at run time and at compile time (within the NTK environment), some functions have different behaviors.

In this case, the sort order for strings within the NTK NewtonScript environment is different from the ordering used on the Newton (and different from other commonly used desktop machine sort orders.) The differences are only apparent if you use characters outside the ASCII range, for instance, accented characters.

If it is necessary to pre-sort accented strings at compile time, you can write your own function that will return the same results as `StrCompare` on an given Newton unit. Here is one such function for English releases of the Newton OS (which assumes strings using only page 0 of the unicode table):

```
constant kNSortTable :=
'[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,
47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,
69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,
91,92,93,94,95,96,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
81,82,83,84,85,86,87,88,89,90,97,98,99,100,101,102,103,104,105,106,
107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,
123,124,125,126,127,128,129,130,131,132,133,161,157,135,136,165,
149,138,137,143,141,152,159,158,144,140,170,134,146,147,148,142,
150,138,168,171,151,153,160,153,154,155,156,174,174,174,174,65,
65,145,67,175,69,175,175,176,176,176,176,162,78,177,177,177,79,
79,164,79,178,178,178,85,166,167,139,65,65,65,65,65,65,145,67,69,
69,69,69,73,73,73,73,169,78,79,79,79,79,79,163,79,85,85,85,85,172,
173,89];

// function to compare strings (only page 0 characters)
// with the same order as the Newton ROM does.
DefConst('kNewtonStrCompare, func(s1, s2)
   begin
      local l1 := StrLen(s1);
      local l2 := StrLen(s2);
      local l := Min(l1, l2);
      local i := 0;
      while i < l and
         (r := kNSortTable[ord(s1[i])] - kNSortTable[ord(s2[i])]) = 0
do
            i := i + 1;
      if i = l then
         l1-l2
      else
         r;
   end);
```

Note that just because you might find a particular function to be defined at compile time, do not assume that it behaves in exactly the same way as the like-named run-time function, unless the documentation explicitly says it does. (And, of course, it might not always be defined in the compile-time environment of future NTK products if it isn't documented that way.)

Profiler and Frames of Functions (7/10/95)

Q: Using the profiler with a large frame of functions gives confusing results. The profiler labels each function by the name of the frame and a number, but the numbers don't seem to correspond to the order in which I defined the functions. Moving the functions around doesn't change the profiler labels. How can I figure out which function is which?

A: If frames have less than than a certain number of slots (20 in the current release), the slots are kept in the order they were defined or added. If there are more than 20 slots in the

frame, the slots are reordered. (This improves slot lookup operations.) The profiler in NTK 1.5 and NTK 1.6labels the functions by their position in the final, possibly reordered, frame.

To determine which function is in which position, you need to look at the frame after the reordering has occurred. You can do this by printing the frame after it's been defined. At compile time you can use a print statement in the slot editor or afterScript. After the package has been downloaded you can use the inspector. Then count (starting from one) through the slots to find your function.

Here's a little inspector snippet that will print the slots in a frame in order with their numbers:

```
call func(theFrame) begin
   local i := 0;
   foreach slot, value in theFrame do begin
      print(i && ': && slot);
      i := i + 1;
   end
end with (<the reordered frame>)
```

NTK 1.6 Heap/Partition Memory Issues (11/24/95)

Q: How do I set the build heap, main heap, and multifinder partition sizes in NTK 1.6 so I can build my package without running out of memory?

A: Here is an explanation of how NTK makes uses of the various heaps. Understanding this will allow you to set your sizes for optimal performance.

Main Heap

The Main heap holds your frame data while you're working in NTK. Its size is set through the Toolkit Preference dialog. You must quit and restart NTK for changes to take effect.

The Main heap is allocated when NTK starts up. It is not disposed off until you quit NTK. If NTK can't allocate the Main heap it reports the problem and quits. As a result, if you can start NTK, Main heap allocation has completed.

We have no rule of thumb for setting the Main heap size. You need to experiment keeping the following in mind:

1) If the Main heap is insufficient, NTK will tell you so.
2) Reducing the Main heap size reduces overall RAM requirements.
3) The Main heap is garbage collected (GC). Increasing its size may improve performance by reducing GC activity. This will affect build time, and to a lesser degree the time it takes to open a project. Please note that the gains in build time are nonlinear and quickly reach a plateau, as shown in the following example:

```
   Main     Build time
heap size      (+/- 0.5 sec)

   1250K                     Main heap ran out of memory...
```

```
1275K          32.7 sec
1300K             26.4 sec
1400K       22.3 sec
1500K       19.2 sec
1600K       17.5 sec
2000K       16.0 sec
3000K       15.2 sec
```

Experiment with Main heap size by measuring build time until you find a reasonable compromise between build time and memory requirements for your particular project.

If you are curious about GC activity, do the following:

1) Add the following line to your `GlobalData` file (in the NTK folder) and restart NTK:

```
protoEditor:DefineKey({key: 65}, 'EvaluateSelection);
```

This allows you to use the period key on the numeric keypad to evaluate selected text in the Inspector window or any text file in the NTK build-time environment. (Normally the text is compiled by NTK and then evaluated by the Newton device when you hit the Enter key.) See the NTK User's Guide for details on the `GlobalData` file.

2) Type `VerboseGC(TRUE)` in the Inspector window, select, and hit the keypad-period key. Each time the GC kicks in, a line will be displayed in the Inspector window. By watching the frequency of GCs, you can get some idea of how your main heap is being used.

3) Use `VerboseGC(FALSE)` to turn this feature off. Please note that `VerboseGC` is available only in the NTK build-time environment. The function does not exist on the Newton device itself. It should be used only for debugging and optimization.

Build Heap

The Build heap holds your package frame data during the last part of the build. Its size is set through the Toolkit Preference dialog. Changes take effect immediately.

The Build heap is allocated only when the Build Package command is issued. It is released as soon as the resulting file is written to disk. As a result Build heap allocation is a recurring issue.

The rule of thumb is to set the Build heap to the size of your package (on the MacOS computer hard disk, not on the Newton device). If the Build heap is insufficient, NTK will tell you so.

There is nothing to be gained by setting the Build heap larger than necessary.

NTK first attempts to allocate the Build heap from MultiFinder memory. If that fails, NTK tries to allocate the Build heap from NTK's partition.

To verify that you have enough memory for the Build heap you need to look at the "About This Macintosh" dialog in the Finder application just prior to issuing the build command.

1) If the "Largest Unused Block" exceeds the Build heap requested size, the Build heap will be allocated from MultiFinder memory.

2) If 1 failed and NTK's partition bar shows enough free memory to accommodate the request, the Build heap will be allocated in NTK's partition.

3) If both 1 and 2 failed, the build will fail. Try to increase MultiFinder free memory by quitting any other open application, or increase the free memory in NTK's partition by closing some or all of NTK's open windows. Then try building again.

To prevent fragmentation of MultiFinder memory launch NTK first, and DocViewer, ResEdit, etc. afterwards. Whenever possible, quit those other applications in the reverse order .

Note: You can use Balloon help to see how much memory an application is actually using. Simply select the Show Balloons menu item and position the cursor on the application partition bar in the About Macintosh dialog. This feature is missing from PowerPC-based MacOS computers.

NTK Partition Size

For NTK 1.6 the rule of thumb for the "smallest useful" partition size for small projects is:
(3500K + Main heap size) for a 680x0 MacOS computer
(5500K + Main heap size) for a PowerPC MacOS computer with Virtual Memory off.

These rules do not include space for the Build heap.

The "smallest useful" partition size is defined by the following example: Using NTK default Main and Build heaps, open the Checkbook sample. Open one browser and one layout window for each file in the project, connect the Inspector, build and download. Perform a global search on "Check" (case insensitive) producing slightly more than 200 matches. Double click on several of these matches displayed in the search results window. Build and download again.

For serious work, increase the partition size by at least 256K for small projects, more for large ones. If you routinely perform global searches that produces many matches, see the next section.

On a PowerPC-based MacOS computer with Virtual Memory on, NTK's 2.7 Meg of code (the exact number is shown in the Finder Info dialog) stays on the hard disk, reducing memory requirements at the expense of performance.

NTK Search and Memory Hoarding (11/24/95)

Q: I sometimes run out space after working with a project for a while. How can I avoid this?

A: NTK 1.6 is built with the MacApp application framework, which brings with it certain memory requirements. Understanding the way NTK uses memory can help avoid running out of memory.

Most of user interface elements you see when using NTK are pointer-based MacApp objects. Allocating a large number of pointers in the application heap causes fragmentation. To prevent that, MacApp has its own private heap where it manages all these pointers.

This heap expands when necessary, but in the current implementation it never shrinks. This memory is not lost, but it may be wasted, effectively reducing free memory in the application partition.

During a single NTK session, build requirements are relatively constant. Partition size requirements will thus be mostly affected by the maximum number of NTK windows open at the same time. If you keep this number reasonable, relative to the partition size you can afford, there should be no problem.

The fact that MacApp's objects heap never shrinks can, however, become an issue when performing searches. The problem is not the search itself, but the number of matches. Each line you see in the Search Results window is a MacApp object occupying 500 to 800 bytes. If your search results in a large number of matches, you may run out of memory.

To reduce such occurrences:
1) Perform more focused searches to keep the number of matches per search reasonable.
2) Close the Search Results window as soon as you are done with it, preferably before doing another search.

NTK Stack Overflow During Compilation (11/24/95)

Q: When I build my project that has very deeply nested statements, NTK runs out of memory and quits. What's going wrong?

A: The deep nesting in your project is causing the compiler to overflow the stack space available in NTK. NTK 1.6 is more likely than than NTK 1.5 to suffer this problem due to new compiler code which nests deeper while parsing if-then-else statements, causing the stack to overflow into the application heap.

If you see an inadvertent crash in NTK during a save operation or a package build:

1) If you are familiar with MacsBug, examine the stack. This particular case will show up in the stack as several calls to the same function before the actual crash.
2) Otherwise, temporarily reduce the number of "else" branches and rebuild the package. If the problem disappears, stack overflow is the prime suspect.

There are at least three ways to avoid this problem and possibly improve performance at the same time:
1) Re-arrange the 'else' statements to resemble a balanced tree
2) Instead of If-then-else statements use:
   An array of functions (with integers as selectors)
   A frame of functions (with symbols as selectors)
3) Finally, as a temporary work around, you can increase the stack size using the ResEdit application.

Re-arrange the 'else' statements to resemble a balanced tree

This solution is the simplest to implement if you need to change existing code. It accommodates non-contiguous integer selectors, and in most cases is faster.

For example, the following code:

```
if x = 1 then
    dosomething
else
    if x = 2 then
        doSomethingElse
    else
        if x = 3 then
            doYetAnotherThing
        else
            if x = 4 then
                doOneMoreThing
            else
                if x = 5 then
                    doSomethingSimple
                else
                    if x = 6 then
                        doThatThing
                    else
                        if x = 7 then
                            doThisThing
                        else // x = 8
                            doTheOtherThing
```

...can be rewritten like this:

```
if x <= 4 then
    if x <= 2 then
        if x = 1 then
            doSomething
        else // x = 2
            doSomethingElse
    else
        if x = 3 then
            doYetAnotherThing
        else // x = 4
            doOneMoreThing
else
    if x <= 6 then
        if x = 5 then
            doSomethingSimple
        else // x = 6
            doThatThing
    else
       if x = 7 then
            doThisThing
       else // x = 8
            doTheOtherThing;
```

Note that the if/then/else statement nesting is "unusual" to illustrate the nesting that the compiler must make each statement is nested as the compiler would process it.

Use an array of functions with integer selectors

Replace a long if-then-else statement with an array of functions. The code is more compact and readable. For a large set of alternatives, the faster direct lookup should compensate for the extra function call. This approach is most useful for a contiguous range of selector values

(e.g., 11 to 65). It can accommodate a few "holes" (for example, 11 to 32, 34 to 56, 58 to 65). It is not practical for non-contiguous selectors (e.g., 31, 77, 256, 1038...)

For example, the following code:

```
if x = 1 then
   dosuchandsuch;
else
   if x = 2 then
      dosomethingelse;
   else
      if x = 3 then
         andsoon;
```

...can be rewritten like this:

```
cmdArray := [func() dosuchandsuch,
   func() dosomethingelse,
      func() andsoon];

   call cmdArray[x] with ();
```

Use a frame of functions with symbols for selectors

This alternative provides the flexibility of using symbols for selecting the outcome.

For example, the following code:

```
if x = 'foo then
   dosuchandsuch;
else
   if x = 'bar then
      dosomethingelse;
   else
      if x = 'baz then
         andsoon;
```

...can be rewritten like this:

```
cmdFrame := {foo: func() dosuchandsuch,
             bar: func() dosomethingelse,
             baz: func() andsoon};

call cmdFrame.(x) with ();
```

Increase NTK's stack size using the ResEdit application

Open the Newton Toolkit application with ResEdit.

Double-click on the "`mem!`" resource icon

Double-click on resource ID `1000` named "Additional NTK Memory Requirements"

Change the fifth (and last) value. This is an hexadecimal number. In NTK 1.6, you should see "`0001 8000`" which is `98304` bytes (or `96k`) to add to the total stack size. For example, to increase this value to `128k` = `131072` bytes change the hexadecimal value to "`0002 0000`".

Unit Import/Export and Interpackage References (11/25/95)

Q: How can I reference information in one part or package from another (different) part or package?

A: Newton 2.0 OS provides the ability for packages to share informations by exporting or importing units. Units are similar to shared libraries in other systems.

A unit provides a collection of NS objects (unit members.) Units are identified by a name, major version number, and minor version number. Any frame part can export or import zero or more units.

A unit must be declared, using `DeclareUnit`, before it's used (imported or exported.) See the docs on `DeclareUnit` below for details.

To export a unit, call `DefineUnit` and specify the NS objects that are exported.

To import from a unit, simply reference its members using `UnitReference` (or UR for short.)

Unit Usage Notes

- Units can also be used to share objects among parts within a single package. This avoids the need to resort to global variables or similar undesirable techniques.

- A part can export multiple units. To achieve some degree of privacy, you can partition your objects into private and public units. Privacy is achieved by not providing the declaration for a unit.

- References to units are resolved dynamically whenever a package is activated or deactivated. For example, a package can be loaded before the package providing the units it imports is loaded. There will be no problem as long as the provider is loaded prior to actually using the imported members.

Conversely, it's possible for the provider to be deactived while its units are in use. The part frame methods, RemovalApproval and ImportDisabled, provide a way to deal with this situation.

Robust code should ensure that the units it imports are available before attempting to use their members. It should also gracefully handle the situation of units being removed while in use. See the DTS sample "MooUnit" for an example.

Unit Build-Time Functions

These functions are available in NTK at build-time only:

`DeclareUnit(unitName, majorVersion, minorVersion, memberIndexes)`
`unitName` - symbol - name of the unit
`majorVersion` - integer - major version number of the unit
`minorVersion` - integer - minor version number of the unit
`memberIndexes` - frame - unit member name/index pairs (slot/value)
return value - unspecified

A unit must be declared by `DeclareUnit` before it's used (imported or exported.) The declaration maps the member names to their indexes. A typical declaration looks like:

```
DeclareUnit('|FastFourierTransforms:MathMagiks|, 1, 0, {
   ProtoGraph:     0,
   ProtoDataSet:   1,
});
```

Typically, the declarations for a unit are provided in a file, such as "FastFourierTransforms.unit", that is added to an NTK project (similar to `.h` files in C.)

When resolving imports, the name and major version specified by the importer and exporter must match exactly. The minor version does not have to match exactly. If there are units differing only in minor version, the one with the largest minor version is used.

Typically, the first version of a unit will have major version 1 and minor version 0. As bug fixes releases are made, the minor version is incremented. If a major (incompatible) change is made, then the major version number is incremented.

Note: When a unit is modified, the indexes of the existing members must remain the same. In other words, adding new members is safe as long as the indexes of the existing members don't change. If you change a member's index it will be incompatible with any existing clients (until they're recompiled with the new declaration.)

`DefineUnit(unitName, members)`
`unitName` - symbol - name of the unit
`members` - frame - unit member name/value pairs (slot/value)
return value - unspecified

`DefineUnit` exports a unit and specifies the value of each member. Immediates and symbols are not allowed as member values. A typical definition looks like:

```
DefineUnit('|FastFourierTransforms:MathMagiks|, {
   ProtoGraph:     GetLayout("foo.layout"),
   ProtoDataSet:   { ... },
});
```

A unit must be declared before it's defined. The declaration used when exporting a unit with `n` members must contain `n` slots with indexes `0..n-1`. The definition must specify a value for every declared member (this is important.)

`UnitReference(unitName, memberName)`
or
`UR(unitName, memberName)`
`unitName` - symbol - name of a unit
`memberName` - symbol - name of a member of unit

return value - a reference to the specified member

To use a unit member call `UnitReference` (`UR` for short) with the unit and member name.

The unit name `'ROM` can be used to refer to obects in the base ROM. For example: `UR('ROM, 'ProtoLabelInputLine)`.

Note: references to objects in the base ROM are sometimes called "magic pointers" and have traditionally been provided in NTK by constants like `ProtoLabelInputLine` or `ROM_SystemSoupName`.

In Newton 2.0 OS, there may also be packages in the ROM. These ROM packages may provide units. Their members are referenced just like any other unit, using `UR`, the unitName, and the memberName. This is the mechanism by which licensees can provide product-specific functionality.

`AliasUnit(alias, unitName)`
`alias` - symbol - alternate name for unit
`unitName` - symbol - name of a unit
return value - unspecified

`AliasUnit` provides a way to specify an alternate name for a unit. Since unit names must be unique, they tend to be long and cumbersome. For example:

```
AliasUnit('FFT, '|FastFourierTransforms:MathMagiks|);
```

...so that you could write:

```
local data := UR('FFT, 'ProtoDataSet):New(points);
```

...instead of:

```
local data := UR('|FastFourierTransforms:MathMagiks|,
'ProtoDataSet):New(points);
```

`AliasUnitSubset(alias, unitName, memberNames)`
`alias` - symbol - alternate name for unit
`unitName` - symbol - name of a unit
`memberNames` - array of symbols - list of unit member names
return value - unspecified

`AliasUnitSubset` is similar to `AliasUnit`, except that it additionally specifies a subset of the units members which can be used. This helps restrict code to using only certain members of a unit.

Unit Part Frame Methods

These methods can optionally be defined in a part frame to handle units becoming unavailable.

`RemovalApproval(unitName, majorVersion, minorVersion)`
`unitName` - symbol - name of the unit
`majorVersion` - integer - major version number of the unit
`minorVersion` - integer - minor version number of the unit
return value - `nil` or string

This message is sent to a part frame when an imported unit is about to be deactivated. It may a return a string to be shown to the user as a warning about the consequences of deactivating the package in use. For example:

```
"This operation will cause your connection to fooWorld to be
dropped."
```

Note: do not assume that the user is removing the package. Other operations such as moving a package between stores also cause package deactivation.

This message is only a warning. The user may decide to proceed and suffer the consequences. If the user proceeds, the `ImportDisabled` message (see below) will be sent.

If the removing the unit is not a problem (for example, your application is closed), then `RemovalApproval` can return `nil` and the user will not be bothered.

`ImportDisabled(unitName, majorVersion, minorVersion)`
`unitName` - symbol - name of the unit
`majorVersion` - integer - major version number of the unit
`minorVersion` - integer - minor version number of the unit
return value - unspecified

This message is sent to a part frame after an imported unit has been deactivated. The part should deal with the situation as gracefully as possible. For example, use alternative data or put up a Notify and/or close your application.

Unit-Related Glue Functions

These functions are available in the Newton 2.0 Platform file.

`MissingImports(pkgRef)`
return value - `nil` or an array of frames (see below)
glue name - `kMissingImportsFunc`

`MissingImports` lists the units used by the specified package that are not currently available. `MissingImports` returns either nil, indicating there are no missing units, or an an array of frames of the form:

```
{
   name: symbol  - name of unit desired
   major: integer - major version number
   minor: integer - minor version number

   <other slots undocumented>
}
```

Store parts and PowerPC-native NTK (5/15/96)

Q: When I build a store part with NTK 1.6 or 1.6.2 on my PowerPC MacOS computer, text searches (for example `mySoup:Query({words: "pizza"})` don't sucessfully find the entries. Why?

A: On PowerPC MacOS computers only, there is a bug in 1.6 and 1.6.2 wherein building store parts will cause this behavior. The workaround is building the store part on a 680x0-based MacOS computer.

If you don't have a 680x0 machine available, you might try any of various third-party applications which remove the PowerPC-native code from an application which contains 680x0 code and PowerPC code, thus forcing it to run the 680x0 code instead. Before doing this, be sure to backup your copy of NTK!

Using Strings as Hex Data and Windows NTK (12/10/96)

Q: When I use `SetClass(SetLength("\u<hex data>"), theLength), theClass)` in Windows NTK , the binary object is not what I expect. It seems to be byte-swapped. How can I create binary objects with data in them in Windows NTK?

A: In Windows NTK (and other Windows NS environments), strings are stored in byte swapped order, that is, low byte first. This is because strings are basically arrays of 16-bit Unicode characters, and on the Intel platform 16-bit values are most usefully stored low byte first. Technically, changing the class and length of a string relies on the internal representation of strings, which isn't documented or supported, though it works fine on Newton OS and Mac OS platforms.

By the time Win NTK is final, a new build-time function will be added that will correctly create a binary object of a given class with data passed as hex bytes in a string. Until then, you can continue to use the hack - just byte swap your source data.

NewtonScript

Nested Frames and Inheritance (10/9/93)

Unlike C++ and other object oriented languages, NewtonScript does not have the notion of nested frames obtaining the same inheritance scope as the enclosing frame.

This is an important design issue, because sometimes you want to enclose a frame inside a frame for name scoping or other reasons. If you do so you have to explicitly state the messages sent as well as explicitly state the path to the variable:

Here's an example that shows the problems:

```
myEncloser := {
   importantSlot: 42,
   GetImportantSlot := func()
      return importantSlot,

   nestedSlot := {
      myInternalValue: 99,

      getTheValue := func()
         begin
         local foo;
```

```
         foo := :GetImportantSlot();         // WON'T WORK; can't
find function
         foo := myEncloser:GetImportantSlot();  // MAY WORK

         importantSlot := 12;       // WON'T WORK; will create new
slot in nestedSlot
         myEncloser.importantSlot := 12;     // MAY WORK
         end
   }
};

myEncloser.nestedSlot:GetTheValue();
```

The proper way to accomplish this is to give the nested frame a `_parent` or `_proto` slot that references the enclosing frame. Nesting the frame is not strictly necessary in this case, only the `_proto` or `_parent` references are used.

Symbol Hacking (11/11/93)

Q: I would like to be able to build frames dynamically and have my application create the name of the slot in the frame dynamically as well. For instance, something like this: `MyFrame:= {}; theSlotName := "Slot_1";`

At this point is there a way to then create the following?... `MyFrame.Slot_1`

A: The function `Intern` takes a string and returns a symbol. There is also a mechanism called path expressions (see the NewtonScript Reference), that allows you to specify an expression or variable to evaluate, in order to get the slot name. You can use these things to access the slots you want:

```
MyFrame := {x: 4};
theXSlotString := "x" ;

MyFrame.(Intern(theXSlotString)) := 6

theSlotName := "Slot_1";
MyFrame.(Intern(theSlotName)) := 7;

// myFrame is now {x: 6, Slot_1: 7}
```

Check for Application Base View Slots (3/6/94)

Here's a simple function that will print out all the slots and the slot values in an application base view. This function is handy if you want to check for unnecessary slots stored in the application base view; these eat up the NewtonScript heap and eventually cause problems with external PCMCIA RAM cards.

```
call func()
begin
   local s,v;
   local root := GetRoot();
```

```
      local base := root.|YourApp:YourSIG|; // name of app
      local prot := base._proto;

      foreach s,v in base do
      begin
         if v and v <> root AND v <> base AND v <> prot then
         begin
            Write ("Slot:" && s & ", Value: ");
            Print(v);
         end;
      end;
   end with ()
```

The debugging function `TrueSize` can also be a valuable tool to determine the heap used by your applications. See the NTK User Guide for more information about `TrueSize`.

Performance of Exceptions vs Return Codes (6/9/94)

Q: What are the performance tradeoffs in writing code that uses try/onexception vs returning and checking error results?

A: We did a few trials to weight the relative performance. Consider the following two functions:

```
thrower: func(x) begin
   if x then
      throw('|evt.ex.msg;my.exception|, "Some error occurred");
   end;

returner: func(x) begin
   if x then
      return -1;  // some random error code,
   0; // nil, true, whatever.
   end;
```

Code to throw and and handle an exception:

```
local s;
for i := 1 to kIterations do
   try
      call thrower with (nil);
   onexception |evt.ex.msg;my.exception| do
      s := CurrentException().data.message;
```

Code to check the return value and handle an error:

```
local result;
local s;
for i := 1 to kIterations do
   if (result := call returner with (nil)) < 0 then
      s := ErrorMessageTable[-result];
```

Running the above loops 1000 times took about 45 ticks for the exception loop, and about 15 ticks for the check the return value loop. From this you might conclude that exception handling is a waste of time. However, you can often write better code if you use exceptions.

A large part of the time spent in the loop is setting up the exception handler. Since we commonly want to stop processing when exceptions occur, we can rewrite the function to set up the exception handler once, like this:

```
local s;
try
   for i := 1 to kIterations do
      call thrower with (nil);
   onexception |evt.ex.msg;my.exception| do
      s := CurrentException().data.message;
```

This code takes only 11 ticks for 1000 iterations, an improvement over the return value case, where we'd have to check the result after each call to the function and stop the loop if an error occurred.

Running the same loops, but passing `TRUE` instead of `NIL` so the "error" occurs every time was interesting. The return value loop takes about 60 ticks, mostly due to the time needed to look up the error message. The exception loop takes a whopping 850 ticks, mostly because of the overhead in the `CurrentException`() call.

With exceptions, you can handle the error at any level up the call chain, without having to worry about each function checking for and returning error results for every sub-function it uses. This will produce code that performs much better, and will be easier to maintain as well.

With exceptions, you do not have to worry about the return value for successful function completion. It is occasionally very difficult to write functions that both have a return value and generate an error code. The C/C++ solution is to pass a pointer to a variable that is modified with what should otherwise be the return value of the function, which is a technique best avoided.

As in the above example, you can attach data to exceptions, so there's no need to maintain an error code to string (or whatever) mapping table, which is another boon to maintainability. (You can still use string constants and so on to aid localization efforts. Just put the constant in the throw call.)

Finally, every time an exception occurs you have an opportunity to intercept it with the NTK inspector. This is also a boon to debugging, because you know something about what's going wrong, and you can set the `breakOnThrows` global to stop your code and look at why there's a problem. With result codes you have a tougher time setting break points. With a good debugger it could be argued that you can set conditional break points on the "check the return value" code, but even when you do this you'll have lost the stack frame of the function that actually had the problem. With exceptions and `breakOnThrows`, all the local context at the time the exception occurred is still available for you to look at, which is an immense aid.

Conclusion: Use exceptions. The only good reason not to would be if your error handler is very local and if you expect it to be used a lot, and if that's true you should consider rewriting the function.

These desciptions document current OS formats only, we reserve the right to extend or change the implementation in future releases.

Generic
NewtonScript objects are objects that reside either in the read-write NewtonScript memory, in pseudo-ROM memory, inside the package, or in ROM. In earlier MessagePad platforms, these objects are aligned to 8-byte boundaries. In Newton 2.0 OS, objects in the NewtonScript memory are aligned to 4-byte boundaries. Inside Newton 2.0 packages, you can optionally align objects to 4-byte boundaries (with NTK's "tighter object packing" checkbox). Alignment causes a very small amount of memory to be wasted, usually less than 2%.

The Newton Object System has four built-in primitive classes that describe an object's basic type: immediates, binary objects, arrays, and frames. The NewtonScript function `PrimClassOf` will return an object's primitive type.

Immediates and Magic Pointers
Immediates (integers, characters, TRUE and NIL) and magic pointers are stored in a 4-byte structure containing up to 30 bits of data and 2 bits of primitive class identification.

Referenced Objects
Binaries, arrays and frames are stored as larger separate objects and managed through references. A reference is a four- byte object. The binary objects, frames, or arrays themselves are stored separately as objects containing a so-called Object Header.

Object Header
Every referenced object has a 12-byte header that contains information concerning size, flags, class, lock count and so on. This information is implementation-specific.

Symbols
A symbol is a binary object that contains a four-byte hash value and a name, which is a null-terminated ASCII string. Each symbol uses 12 (header) + 4 (hash value) + length of name + 1 (null terminator) bytes.

Binary Objects
A binary object contains a 12- byte header plus space for the actual data (allocated in 8 -byte chunks.)

Strings
Strings are binary objects of class (or a subclass of) `String`. A string object contains a 12-byte header plus the Unicode strings plus a null termination character. Note that Unicode characters are two-byte values. Here's an example:

```
"Hello World!"
```

This string contains 12 characters, in other words it has 24 bytes. In addition we have a null termination character (24 + 2 bytes) and an object header (24 + 2 + 12 bytes), all in all the object is 38 bytes big. Note that we have not taken into account any possible savings if the string was compressed (using the NTK compression flags).

Rich Strings
Rich strings extend the string object class by embedding ink information within the object. Within the unicode, a special character `kInkChar` is used to mark the position of an ink

word. The ink data is stored after the null termination character. Ink size varies depending on stroke complexity.

Array Objects
Array objects have an object header (12 bytes) and additional four bytes per element which hold either the immediate value or a reference to a referenced object. To calculate the total space used by an array, you need to take into account the memory used by any referenced objects in the array.

Here's an example:

```
[12, $a, "Hello World!", "foo"]
```

We have a header (12 bytes) plus four bytes per element (12 + (4 * 4) bytes). The integer and character are immediates, so no additional space is used, but we have 2 string objects that we refer to, so the total is (12 + (4*4) + 38 + 20 bytes) 86 bytes. We have not taken into account savings concerning compression. Note that the string objects could be referred by other arrays and frames as well, so the 38 and 20 byte structures are stored only once per package.

Frame Objects
We have two kinds of frames: frames that don't have a shared map object; and frames that do have a shared map object. We take the simple case first (no shared map object).

The frame is maintained as two array-like objects. One, called the frame map, contains the slot names, and the other contains the actual slot values. A frame map has one entry per symbol, plus one additional 4 -byte value.

The frame map uses a minimum of 16 bytes. If we add the frame's object header to this, the minimal size of a frame is 28 bytes. Each slot adds 8 bytes to the storage used by the frame (two array entries.) Here's an example:

```
{Slot1: 42, Slot2: "hello"}
```

We have a header of 28 bytes, and in addition we have two slots, for a total of (28 + (2 * 8)) 48 bytes. This does not take into account the space used for each of the slot name symbols or for the string object. (The integer is an immediate, and so is stored in the array.)

Multiple similar frames (having the same slots) could share a frame map. This will save space, reducing the space used per frame (for many frames all sharing the same map) to the same as used for an array with the same number of slots. (If just a few frames share the frame map, we need to take into account the amortized map size that the frames share. So the total space for N frames sharing a map is N*28 bytes of header per frame, plus the size of the frame map, plus the size of the values for the N frames.

Here's an example of a frame that could share a map with the previous example:

```
{Slot1: 56, Slot2: "world"}
```

We have a header of 12 bytes. In addition, we have two slots (2 * 4), and additional 16 bytes for the size of a map with no slots Ñ all in all, 36 bytes. We should also take into account the shared map, which is 16 bytes, plus the space for the two symbols.

When do frames share maps?

1. When a frame is cloned, both the copy and the original frame will share the map of the original frame. A trick to make use of this is to create a common template frame, and clone this template when duplicate frames are needed.

2. Two frames created from the same frame constructor (that is, the same line of NewtonScript code) will share a frame map. This is a reason to use `RelBounds` to create the `viewBounds` frame, and it means there will be a single `viewBounds` frame map in the part produced.

Note**:** These figures are for objects in their run-time state, ready for fast access. Objects in transit or in storage (packages) are compressed into smaller stream formats. Different formats are used (and different sizes apply) to objects stored in soups and to objects being streamed over a communications protocol.

Symbols vs Path Expressions and Equality (7/11/94)

Q: While trying to write code that tests for the existance of an index, I tried the following, which did not work. How can I compare path expressions?

```
if value.path = '|name.first| then ... // WRONG
```

A: There are several concerns. `'|name.first|` is not a path expression, it is a symbol with an escaped period. A proper path expression is either `'name.first` or `[pathExpr: 'name, 'first]`. The vertical bars escape everything between them to be a single NewtonScript symbol.

The test `value.path = 'name.first` will always fail, because path expressions are deep objects (essentially arrays) the equal comparison will compare references rather than contents. You will have to write your own code to deeply compare path expressions.

This code is further complicated by the fact that symbols are allowed in place of path expressions that contain only one element, but the two syntaxes produce different NewtonScript objects with different meanings. That is, `'name = [pathExpr: 'name]` will always fail, as the objects are different.

A general test is probably unnecessary in most circumstances, since you will be able to make assumptions about what you are looking for. For example, here is some code that will check if a given path value from a soup index is equivalent to `'name.first`:

```
if ClassOf(value.path) = 'pathExpr and Length(value.path) = 2
   and value.path[0] = 'name and value.path[1] = 'first then ...
```

Function Size and "Closed Over" Environment (7/18/94)

Q: I want to create several frames (for soup entries) that all share a single function, but when I try to store one of these frames to a soup, I run out of memory. Can several frames share a function and still be written to a soup? My code looks like this:

```
...
local myFunc := func(...) ...;
local futureSoupEntries := Array(10, nil);
```

```
for i := 0 to 9 do
   futureSoupEntries[i] := {
      someSlots: ...,
      aFunction: myFunc,
   };
...
```

A: When a function is defined within another function, the lexically enclosing scope (locals and paramaters) and message context (self) are "closed over" into the function body. When NewtonScript searches for a variable to match a symbol in a function, it first searches the local scope, then any lexically enclosing scopes, then the message context (self), then the _proto and _parent chains from the message context, then finally the global variables.

Functions constructed within another function, as in your example, will have this enclosing lexical scope, which is the locals and parameters of the function currently being executed, plus the message context (self) when the function is created. Depending on the size of this function and how it's constructed, this could be very large. (Self might be the application's base view, for example.)

A `TotalClone` is made during the process of adding an entry to a soup, and this includes the function body, lexical scopes, and message context bound up within any functions in the frame. All this can take up a lot of space.

If you create the function at compile time (perhaps with `DefConst('kMyFunc, func(...) ...)`) it will not have the lexically enclosing scope, and the message context at compile time is defined to be an empty frame, and so cloning such a function will take less space. You can use the constant `kMyFunc` within the initializer for the frame, and each frame will still reference the same function body. (Additionally, the symbol `kMyFunc` will not be included in the package, since it is only needed at compile time.)

If the soup entries are only useful when your package is installed, you might consider instead replacing the function body with a symbol when you write the entry to the soup. When the entry is read from the soup, replace the symbol with the function itself, or use a `_proto` based scheme instead. Each soup entry will necessarily contain a complete copy of the function, but if you can guarantee that the function body will always be available within your application's package, it might be unnecessarily redundant to store a copy with each soup entry.

TrueSize Incorrect for Soup Entries (2/6/96)

Q: When I use `TrueSize` to get the size of a soup entry I get results like 24K or even 40K for the size. That can't be right. What's going on?

A: `TrueSize` "knows" about the underlying implementation of soup entries. A soup entry is really a special object (a fault block) that contains information about how to get an entry and can contain a cached entry frame. In the information about how to get an entry, there is a reference to the soup, and various caches in a soup contain references to the cursors, the store, and other (large) NewtonScript objects. `TrueSize` is reporting the space taken up by all of these objects. (Note: calling `TrueSize` on a soup entry will force the entry to be faulted in, even if it was not previously taking up space in the NewtonScript heap.)

The result is that `TrueSize` is not very useful when trying to find out how much space the cached frame for an entry is using. A good way to find the space used for a cached entry frame is to call `gc(); stats();` record the result, then call `EntryUndoChanges(entry); gc(); stats()`. The difference between the two free space reports will be the space used by the cached frame for a given entry.

`EntryUndoChanges(entry)` will cause any cached frame to be removed and the entry to return to the unfaulted state. Gc() then collects the space previouly used by the cached entry frame.

If you want the `TrueSize` breakdown of the types of objects used, you can `Clone` the entry and call `TrueSize` on the copy. This works because the copy is not a fault block, and so it does not reference the soups/cursors/stores.

NEW: Floating Point Numbers Are Approximations (3/28/97)

Q: The functions `Floor` and `Ceiling` seem broken. For instance, `Floor(12.2900 * 10000)` returns 122899, not 122900. What's going on?

A: This is not a bug in `Floor` or `Ceiling`. This happens because of the way floating point numbers are stored, and the limitation is common to many real number representations. In the same way that 1/3 cannot accurately be represented in a finite number of digits in base 10 (it is .3333333333...), likewise 1/10 cannot be exactly represented as a fractional part in base 2. Because number printers typically round to a small number of significant digits, you don't normally notice this. The NTK inspector, for example, displays only 5 significant figures in floating point numbers. However, if you display the number with enough precision, you'll see the representation error, where the real is actually slightly larger or smaller than the intended value.

```
FormattedNumberStr(0.1, "%.18f")  ->  "0.10000000000000010"
FormattedNumberStr(0.3, "%.18f")  ->  "0.29999999999999990"
```

The functions `Floor` and `Ceiling` are strict, and do not attempt to take this error into account. In the example, `12.29` is actually `12.2899999999999990`, which multiplied by `10000` is `122,899.999999999990`. The largest integer less than this number (`Floor`) is correctly `122899`.

There are usually ways to work around this problem, depending on what you are trying to accomplish. To convert a floating point number to an integer, use `RIntToL`, which rounds to the nearest integer avoiding the problems caused with round-off error and `Floor` or `Ceiling`. `RIntToL(x)` produces the same result that `Floor(Round(x))` would produce.

```
RIntToL(12.29*10000)  ->  122900
```

If you need to format a number for display, use a formatting function such as `FormattedNumberStr`. These functions typically round to the nearest displayable value. To display 2 decimal digits, use "%.2f":

```
FormattedNumberStr(12.29, "%.2f")  ->  "12.29"
```

If you're working with fixed point numbers such as dollar amounts, consider using integers instead of reals. By representing the value in pennies (or mils, or whatever) you can avoid the imprecision of reals. For example, represent `$29.95` as the integer `2995` or `29950`, then divide by `100` or `1000` to display the number. If you do this, keep in mind that there is a

maximum representable integer value, `0x1FFFFFFF` or `536870911`, which is sufficient to track over 5 million dollars as pennies, but can't go much over that.

If you really need to find the greatest integer less than a certain number and can't tolerate how `Floor` deals with round off errors, you'll need to do some extra work keeping track of the precision of the number and the magnitude of the round off error. It's worthwhile to read a good numeric methods reference. Floating point numbers in NewtonScript are represented by IEEE 64-bit reals, which are accurate to around 15 decimal digits. The function `NextAfterD` provides a handy way to see how 'close together' floating point numbers are.

```
FormattedNumberStr(NextAfterD(0.3, kInfinity), "%.18f");
                                        ->  "0.300000000000000040"
```

NEW: Real Numbers in NewtonScript (3/28/97)

Q: How are real numbers represented as floating point in NewtonScript? How accurate are they? What about infinities, NANs, and other exceptions?

A: Real numbers in NewtonScript are represented as IEEE 64-bit floating point numbers, which are accurate to about 15 decimal digits. You can read more about the IEEE floating point numbers in "Inside Macintosh: PowerPC Numerics" available online at the URL:

`http://gemma.apple.com/dev/techsupport/insidemac/PPCNumerics/PPCNumerics-2.html`

The Newton floating point environment is not as rich in features as the PowerPC environment, and the PowerPC numerics document is only mentioned as a useful resource for understanding floating point issues. It in no way documents API or features of the Newton floating point environment.

Briefly, numbers are represented by 1 bit of sign ("on" is negative), 11 bits of exponent, and 52 bits of fractional part. The exponent bits are stored in excess 0x3FF, that is, 0x3FF is the representation for 0, values greater than 0x3FF are positive exponents, and values less than 0x3FF are negative exponents. The 52 bits of fractional part actually provide 53 bits of accuracy, because the initial 1 bit is dropped.

For example, suppose that we want to convert 9 97/128 into IEEE 64 bit format:

1) convert to base 2
   `1001.1100001`
2) shift number to the form of 1.yyyyyy * 2^Z
   `1.0011100001 * 2^3`
3) add 0x3FF (excess 0x3FF) to exponent field, convert to binary.
   `3+0x3FF = 0x402 = 100 0000 0010`
4) now put the numbers together, using only the fractional part of the number represented above, in the form of yyyyyy
   `0 10000000010`
   `00111000010000000000000000000000000000000000000000000000`
   in hex representation, this is `0x4023840000000000`
5) Just to verify, try it: `StrHexDump(9+97/128, 16) -> "4023840000000000"`

The IEEE standard also allows for non-normal numbers. Here are the exceptions:

infinity	e = 7FF, f = 0 (+ or - depending on sign bit)
NaN	e = 7FF, f <> 0 (also overflow, error, etc.)

zero e = 0, f = 0 (+ or -, depending on sign bit)
subnormal e = 0, f <> 0 (these are less precise numbers, smaller than the smallest normal number)

Note that there is more than one not a number value. In fact, there are quite a large number. The IEEE spec assigns meaning to various NaN values, as well as defining signalling and quiet NaNs. NewtonScript does not distinguish between NaN values. One NaN is as good as another.

In NewtonScript, real numbers are 8-byte binary objects of class 'real. In addition to the NewtonScript floating point literal syntax, you can use the compile time function MakeBinaryFromHex to construct real numbers, and you must use this style for custom NaN values. The most recent platform files for Newton 2.0 and Newton 2.1 provide constants for negative zero (kNegativeZero), positive and negative infinity (kInfinity, kNegativeInfinity), and a canonical NaN (kNaN).

```
MakeBinaryFromHex("4023840000000000", 'real) -> 9.7578125 // = 9+97/128
```

Pickers, Popups and Overviews

Determining Which ProtoSoupOverview Item Is Hit (2/5/96)

Q: How do I determine which item is hit in a protoSoupOverview?

A: There is a method called HitItem that gets called whenever an item is tapped. The method is defined by the overview and you should call the inherited one. Also note that HitItem gets called regardless of where in the line a tap occurs. If the tap occurs in the checkbox, you should do nothing, otherwise you should do something.

The method is passed the index of the hit item. The index is relative to the item displayed at the top of the displayed list. This item is always the current entry of the cursor used by protoSoupOverview. So, you can find the actual soup entry by cloning the cursor and moving it.

Here is an example of a HitItem method. If the item is selected (the checkbox is not tapped) then the code will set an inherited cursor (called myCursor) to the entry that was tapped on:

```
func(itemIndex, x, y)
begin
   // MUST call the inherited method for bookeeping
   inherited:HitItem(itemIndex, x, y);

   if x > selectIndent then
   begin
  // get a temporary cursor based on the cursor used
  // by soup overview
      local tCursor := cursor:Clone();

  // move it to the selected item
      tCursor:Move(itemIndex) ;
```

```
  // move the inherited cursor to the selected entry
      myCursor:Goto(tCursor:Entry());

  // usually you will close the overview and switch to
  // some other view
      self:Close();
   end;
    // otherwise, just let them check/uncheck
 // which is the default behavior
end
```

Displaying the ProtoSoupOverview Vertical Divider (2/5/96)

Q: How can I display the vertical divider in a `protoSoupOverview`?

A: The mechanism for bringing up the vertical divider line was not correctly implemented in `protoSoupOverview`. You can draw one in a `viewDrawScript` as follows:

```
// setup a cached shape for efficiency
mySoupOverview.cachedLine := nil;

mySoupOverview.viewSetupDoneScript := func()
begin
   inherited:?viewSetupDoneScript();

   local bounds := :LocalBox();
   cachedLine := MakeRect(selectIndent - 2, 0,
        selectIndent - 1, bounds.bottom);
end;

mySoupOverview.viewDrawScript := func()
begin
   // MUST call inherited script
   inherited:?viewDrawScript();

   :DrawShape(cachedLine,
      {penPattern: vfNone, fillPattern: vfGray});
end;
```

Validation and Editing in ProtoListPicker (4/1/96)

Q: I am trying to use the `ValidationFrame` to validate and edit entries in a `protoListPicker`. When I edit certains slots I get an error that a path failed. All the failures occur on items that are nested frames in my soup entry. What is going on?

A: The built-in validation mechanism is not designed to deal with nested soup information. In general, you gain better flexibility by not using a `validationFrame` in your pickerDef, even if you have no nested entries. Instead, you can provide your own validation mechanism and editors:

Define a `Validate` method in your picker definition
Define an `OpenEditor` method in your picker definition
Draw a layout for each editor you require

`pickerDef.Validate(nameRef, pathArray)`
`nameRef` - nameRef to validate
`pathArray` - array of paths to validate in the nameRef
returns an array of paths that failed, or an empty array

Validate each path in `pathArray` in the given nameRef. Accumulate a list of paths that are not valid and return them.

The following example assumes that `pickerDef.ValidateName` and `pickerDef.ValidatePager` have been implemented:

```
pickerDef.Validate := func(nameRef, pathArray)
begin
   // keep track of any paths that fail
   local failedPaths := [];

   foreach index, path in pathArray do
   begin
      if path = 'name then
      begin
         // check if name validation fails
         if NOT :ValidateName(nameRef) then
            // if so, add it to array of failures
            AddArraySlot(failedPaths, path);
      end;
      else begin
         if NOT :ValidatePager(nameRef) then
            AddArraySlot(failedPaths, path);
      end;
   end;
   // return failed paths or empty array
   failedPaths;
end;
```

`pickerDef.OpenEditor(tapInfo, context, why)`
The arguments and return value are as per `OpenDefaultEditor`. However, you need to use this instead of `DefaultOpenEditor`.

```
pickerDef.OpenEditor := func(tapInfo, context, why)
begin
   local valid = :Validate(tapInfo.nameRef, tapInfo.editPaths) ;
   if (Length(valid) > 0) then
      // if not valid, open the editor
      // NOTE: returns the edit slip that is opened
      GetLayout("editor.t"):new(tapInfo.nameRef,
         tapInfo.editPaths, why, self, 'EditDone, context);
   else
   begin
      // the item is valid, so just toggle the selection
      context:Tapped('toggle);
      nil;                              // Return <nil>.
   end;..
end;
```

The example above assumes that the layout "editor.t" has a `New` method that will open the editor and return the associated View.

The editor can be designed to fit your data. However, we suggest that you use a `protoFloatNGo` that is a child of the root view created with the `BuildContext` function. You are also likely to need a callback to the pickderDef so it can appropriately update the edited or new item. Finally, your editor will need to update your data soup uing an "Xmit" soup method so that the listPicker will update.

In the `OpenEditor` example above, the last three arguments are used by the editor to send a callback to the pickerDef from the `viewQuitScript`. The design of the callback function is up to you, here is an example:

```
pickerDef.EditDone := func(nameRef, context)
begin
   local valid = :Validate(tapInfo.nameRef, tapInfo.editPaths) ;
   if (Length(valid) > 0) then
   begin
   // Something failed. Try and revert back to original
      if NOT :ValidatePager(nameRef) AND
         self.('[pathExpr: savedPagerValue, nameRef]) = nameRef
         then
         nameRef.pager := savedPagerValue.pager;

      context:Tapped(nil);    // Remove the checkmark
   end;
   else
      // The nameRef is valid, so select it.
      context:Tapped('select);

   // Clear the saved value for next time.
   savedPagerValue := nil;
end;
```

Picker List is Too Short (4/29/96)

Q: I have items in my picker list with different heights that I set using the `fixedHeight` slot. When I bring up the picker, it is not tall enough to display all the items. Worse, I cannot scroll to the extra items. What is going on?

A: The `fixedHeight` slot is used for two separate things. Any given pick item can use the `fixedHeight` slot to specify a different height. This works fine.

However, the code in Newton 2.0 OS that determines how big the list should be also uses the `fixedHeight` slot of the first pick item (in other words, `pickItems[0]`) if it exists. It is as if the following code executes:

```
local itemHeight := kDefaultItemHeight;
if pickItems[0].fixedHeight then
   itemHeight := pickItems[0].fixedHeight;
local totalHeight := itemHeight * Length(pickItems);
```

This total height is used to figure out if scrolling is required. As you can see, this can cause problems if your first item is not the tallest one. The solution is to make sure the first item

in your `pickItems` array has a `fixedHeight` slot that is sufficiently large to make scrolling work correctly. This may be fixed in future revisions of the NewtonOS.

Note that there will be similar problems if your pick items contain icons. The system will use the default height unless you specify a `fixedHeight` slot in your first item. The default height is not tall enough for most icons. In other words, if you have icons in your pick items, you must have a `fixedHeight` slot in the first item that is set to the height of your icon.

Tabs Do Not Work With ProtoTextList (5/8/96)

Q: I tried to use tabs to get columns in a `protoTextList` but they do not appear. How do I get columns?

A: The text view in `protoTextList` is based on a simple text view which does not support tabs. If you want scrolling selectable columns you can use shapes to represent the rows. If you need finer control, use the `LayoutTable` view method.

How to Avoid Problems with ProtoNumberPicker (8/23/96)

Q: I am thinking of using `protoNumberPicker` for input. (or) I have used `protoNumberPicker` and have encountered a bug/misfeature/problem. What should I use?

A: `protoNumberPicker` has several instabilities and bugs. We recommend that you use the DTS sample code "protoNumberPicker_TDS". It provides all of the features of `protoNumberPicker` with none of the bugs. It also provides additional functionality that is not in `protoNumberPicker`. See the sample code for more detail.

Single Selection in ProtoListPicker-based Views (9/20/96)

Q: How do I allow only one item to be selected in a `protoListPicker`, `protoPeoplePicker`, `protoPeoplePopup`, or `protoAddressPicker`?

A: The key to getting single selection is that single selection is part of the picker definition and not an option of `protoListPicker`. That means that the particular class of nameRef you use must include single selection. In general, this requires creating your own subclass of the particular name reference class.

The basic solution is to create a data definition that is a subclass of the particular class that your `protoListPicker` variant will view. That data definition will include the `singleSelect` slot. As an example, suppose you want to use a `protoPeoplePopup` that just picks individual people. You could use the following code to bring up a `protoPeoplePopup` that only allowed selecting one individual at one time:

```
// register the modified data definition
RegDataDef('|nameref.people.single:SIG|,
{_proto: GetDataDefs('|nameRef.people|), singleSelect: true});
```

```
// then pop the thing
protoPeoplePopup:New('|nameref.people.single:SIG|,[],self,[]);

// sometime later
UnRegDataDef('|nameref.people.single:SIG|);
```

For other types of `protoListPickers` and classes, create the appropriate subclass. For example, a transport that uses `protoAddressPicker` for emails might create a subclass of `'|nameRef.email|` and put that subclass symbol in the `class` slot of the `protoAddressPicker`.

Since many people are likely to do this, you may cut down on code in your `installScript` and `removeScript` by registering your dataDef only for the duration of the picker. That would mean registering the class just before you pop the picker and unregistering after the picker has closed. You can use the `pickActionScript` and `pickCanceledScript` methods to be notified when to unregister the dataDef.

How to Change Font or LineHeight in ProtoListPicker (9/20/96)

Q: How do I set a different font for the items in the `protoListPicker`?

A: There is a way to change the font in the Newton 2.0 OS, however, we intend to change the mechanism in the future. Eventually, you will be able to set a `viewFont` slot in the `protoListPicker` itself and have that work (just like you can set `viewLineSpacing` slot now). In the meantime, you need a piece of workaround code. Warning: you must set the viewFont of the listPicker AND include this workaround code in the `viewSetupDoneScript`:

```
func()
   begin
      if listBase exists and listBase then
         SetValue(listBase, 'viewFont, viewFont) ;

      inherited:?viewSetupDoneScript();
   end;
```

This will set the `viewFont` slot of the `listBase` view to the `viewFont` of the `protoListPicker`. You cannot rely on the listbase view always being there, hence the test for its existence.

Note that you can use the same code to modify the `lineHeight` slot of the listPicker. Just substitute `lineHeight` for `viewFont` in the code snippet. The one caveat is that the `lineHeight` must be at least 13 pixels.

How to Preselect Items in ProtoListPicker (9/20/96)

Q: If I put name references in the `selected` array of a `protoListPicker`, it throws a `-48402` error. How do I preselect items?

A: In the MessagePad 120/130 units it is not possible to preselect items in the listPicker and have it work correctly. We recommend that you use the "protoSlimPicker" DTS Sample Code instead.

ProtoDigit Requires a DigitBase View (9/24/96)

Q: I get an exception concerning an undocumented `digitbase` slot in `protoDigit`. The slot is not documented in the current release of the documentation. How can I make `protoDigit` work?

A: `protoDigit` is not really designed to be used independently. You should use the `protoNumberPicker_TDS` sample code for input like this.

If you really need to use `protoDigit, remember that` it expects to be contained in a view that has a `declareSelf` slot whose value is the symbol `digitBase`. To solve the problem, draw out a `clView`, give it a `declareSelf` slot with a value of `'digitBase` and draw your `protoDigits` inside that view. You are responsible for propagating carries and other information to all `protoDigits`. You are also responsible for animation and the flip digit look. Unfortunately, the dotted line picture is not available.

As of 2/6/96, the Newton 2.0 Platform file also gives a `protoDigit` a default `digitBase` slot of the number type. This slot must be removed.

How to Get ProtoSoupOverview Selections (10/3/96)

Q: How do I get the selected items in `protoSoupOverview`?

A: The final documentation inadvertantly left out the following documentation on the `selected` slot:

`selected` - Required. Initially set to `nil`; it is modified by `protoSoupOverview` as the user selects and deselects overview items.

This proto is an array of aliases to the selected items when the overview is closed. For example:

```
[[alias: NIL, 66282812, 84, "Names"],
 [alias: NIL, 66282812, 85, "Names"]]
```

Dynamically Adding to ProtoTextList Confuses Scrolling (1/15/97)

Q: I am adding items to a `protoTextList` after it is displayed. I add an item and scroll to highlight that item. However, the state of the scroll arrows does not correctly get updated. Sometimes it will indicate that there are more items to scroll when it is really at the end of the list.

A: There is a problem with Newton 2.0 OS devices (although not Newton 2.1 OS devices) that causes the `protoTextList` to reset the scroll distance when you update the listItems array. The workaround is to always scroll the list to the top before calling `SetupList` when you add items. Then you can scroll the list to where you want it. Note that this workaround is safe to use in Newton 2.1 OS as well. In other words, if you are adding items

to a `protoTextList`, use this workaround unless your application is Newton 2.1 OS-specific. This method will add a single item to the `protoTextList`, set the highlighted item to the new item and scroll if required. It will also make sure the item is unique.

```
AddListItem := func(newItem)
begin
    // Insert the item if not already in Array
   local index := BInsert(listItems, item, '|str<|, nil, true);

   // item must be in the array and index will point to the item.

   if NOT index then
   begin
      :Notify(kNotifyAlert, kAppName, "Duplicate entry.");
      return nil;
   end;

   // workaround a bug in 2.0 that causes the scroll arrows to get
out of sync
   // do this by scrolling to the top
   :DoScrollScript(-viewOriginY) ;

   self:SetUpList();

   // Setting the selection slot will highlight the item
   selection := index;

   // scroll to show the new item
   if index >= viewLines then
      :DoScrollScript((index - viewLines + 1) * lineHeight) ;

   self:RedoChildren();

   return true;
end ;
```

Recognition

Custom Recognizers (2/8/96)

Q: Can I build recognizers for gestures and objects other than those built into the Newton system?

A: In Newton 2.0 OS, theres no support to add custom recognizers using the Newton Toolkit. Stay tuned for more information concerning this.

Some recognition engines can work in a window separate from the edited text. For instance, writing a "w" in a special view might causes "w" to appear in the currently edited text view (the key view.) This type of recognition system can be implemented as a keyboard. If you want to use this approach, you might want to use a function in the Newton 2.0 Platform file that allows your keyboard to replace the built-in alphanumeric "typewriter" keyboard. See the Platform File Notes documentation for more information on the `RegGlobalKeyboard` function.

How to Save and Restore Recognition Settings (4/9/96)

Q: Can I capture a user's recognition settings, which then may later be restored?

A: Yes, the global functions `GetUserSettings`, `SetDefaultUserSettings` and `SetUserSettings` allow you to manipulate recognition-related user preference data. These functions can allow an application to keep and manage recognition settings for multiple users. These functions only manage information about the recognition settings, and no other user preference settings.

`GetUserSettings()`
This function returns a frame of the current user recognition settings; this frame is the argument for `SetUserSettings`. Do not modify the frame this function returns. Do not rely on any values, as the frame may change in future releases.

`SetDefaultUserSettings()`
This function sets recognition-related user preference settings to default values.

`SetUserSettings(savedSettings)`
savedSettings - Recognition preferences frame returned by `GetUserSettings`.
Sets user preferences for recognition as specified.

CHANGED: Opening the Corrector Window (3/17/97)

Q: I want the corrector window available for the user at specific times, can I open it from within my application?

A: Yes, below is the code you should use to open the corrector window. For compatibility, you should always make sure the corrector exists. The corrector itself requires that a correctable key view exists.

```
local correctView := GetRoot().correct;
if correctView and (GetCaretBox() or GetHiliteOffsets()) then
   correctView:Open();
```

Note: An older version of this Q&A (from 12/8/95) showed using `GetKeyView` as a test to make sure a correctable view was the key view. With the changes to the OS with the Newton 2.1 release that allow any view to be a key view, this is no longer a reliable test. The corrector will fail to open (generating a -48204 "bad path" error) if the key view does not support the caret or a selection. Calling `GetCaretBox` and `GetHiliteOffsets` is a more reliable test to see if a correctable view is available.

Routing

Printing Resolution 72DPI/300DPI (2/8/94)

Q: I've tried to print PICT resources; the picture was designed in Illustrator and copied to the clipboard as a PICT. The picture printed correctly but at a very low resolution. Is there any way of printing PICTs with a higher resolution?

A: Currently the only supported screen resolution for PICT printing is 72dpi. This may change in future platforms, so stay tuned for more information.

Not all Drawing Modes Work with a PostScript Printer (3/8/94)

Q: It seems that not all drawing modes work with printing. Is that true?

A: Yes. PostScript behaves like layers of paint: you can not go back and change something. Anything that uses an invert mode (like XOR, and possibly ModeNot* (to be tested)), will not work.

Note: If you want to get the effect of white text on a black/filled background, use bit clear mode for drawing the text.

PICT Printing Limitations (6/9/94)

Q: My large pictures cannot print on my LaserWriter. Is there a maximum size Newton picture?

A: The current PostScript printing system in the Newton ROMs is unable to print extremely large individual bitmap frames, the kind of pictures created using the NTK Picture editor or the GetPictAsBits routine. This is because in order to print these, the Newton must copy the bitmaps into an internal buffer. Thus the GetPictAsBits case fails (current limitation is a 168K buffer, but do not rely on a specific number for other Newton devices).

Using the `GetNamedResource(..., 'picture)` routine, you can use PICT resources to be drawn in clPictureViews. MacOS PICT resources often contain multiple opcodes (instructions). For single-opcode PICTs, compression is done for the whole picture. You can check *Inside Macintosh* documentation for specifications of the PICT format. If you are using very large bitmaps which you will print, you should use PICT resources composed of many smaller 'bitmap copy' opcodes because they will print much faster and more reliably on PostScript printers. This is because very large PICT opcodes printed to LaserWriters must be decompressed on the printer. The printer's decompression buffer is sometimes too small if the opcodes represent large bitmaps. Check your MacOS graphics application documentation for more information on segmenting your large PICTs into smaller pieces. For some applications, you might have two versions of the PICTs, one for displaying (using `GetPictAsBits` for faster screen drawing), and a large tiled PICT for printing.

Starting with N2 OS, color PICTs (PICT 2) are supported. Colors will be interpolated into gray values.

Printing Fonts with a PostScript Printer (7/26/94)

Q: When printing from my application on the Newton to a PostScript Laser printer, I noticethat the fonts are being substituted. Printing always looks fine on a QuickDraw printer like the StyleWriter.

A: Yes, this is true.The additional System font (Espy Sans) or any custom Newton font created with the Newton Font Tool is not printed directly to a LaserWriter because the fonts are missing in the PostScript font versions. Just printing Espy Sans (Newton system fonts) is currently not possible on the LaserWriter, but is possible on faxes and bitmap printer drivers, since the rendering for those is done inside the Newton.

For the built-in Espy font, the troublesome characters are the Apple-specific ones, starting with Hex FC. The filled diamond is one of these characters, the specific tick box arrow is another.

For printing, you might need to include bitmaps for special characters or words in your application in order to print them (that is, if the normal LaserWriter fonts are unacceptable)

Note that if you want a monospaced font, check out the Monaco DTS sample. That includes a font which will print as the monospaced Courier font.

Printing Does Not Have Access to My Application Slots (11/27/95)

Q: Why can't I find my application slots from my print format?

A: Print format does not have direct access to your application context because it is not a child of your application, so it cannot rely on the parent inheritance chain. All viewDefs should be designed so that they do not rely on your application being open or rely on state-specific information in your application. The application may be closed, or the user may continue to work in your application while the print/fax transport is imaging.

Print format does have access to the `target` variable (it will contain the "body" of the data sent; don't use `fields`.body). Note that if mulitiple items are sent, the value of `target` will change as the print format iterates over the list. Try to put the real "data" for the routing in the target using the view method `GetTargetInfo`.

If, for some reason, you need to access slots from your application, you can access them using `GetRoot().(yourAppSymbol).theSlot`.

How to Open the Call Slip or Other Route Slips (12/19/95)

Q: How do I open the call slip (or other Route Slips) programatically?

A: Use the global function `OpenRoutingSlip`. Create a new item with the transport's `NewItem` method and add routing information such as the recipient information in the `toRef` slot. For the call slip, the transport symbol will be `'|phoneHome:Newton|`, but this approach will work for other transports. (For transports other than the call transports, you will also provide the data to route in the `item.body` slot.)

Determining the value of the toRef slot

The `toRef` slot in the item frame should contain an array of recipients in the form of nameRefs, which are the objects returned from `protoPeoplePicker` and other `protoListPicker`-based choosers. Each nameRef can be created from one of two forms: a cardfile soup entry, or just a frame of data with minimal slots. (The required slots vary depending on the transport. For instance, the current call transport requires only phone, name, and country.)

1. Cardfile entry:

```
entry := myCursor:Entry();
```

2. Create your own pseudo-entry:

```
entry := {
   phone:"408 555 1234",
   name: {first: "Glagly", last: "Wigout"},
   country: "UK",
};
```

Make the entry into a "nameRef" using the nameRef's registered datadef -- an object which describes how to manipulate nameRefs of a specific class. Note that every transport stores its preferred nameRef class symbol in its `transport.addressingClass` slot. (Examples are `'|nameRef.phone|` and `'|nameRef.email|`).

```
local class := '|nameRef.phone|;
local nameRef := GetDataDefs(class):MakeNameRef(myData, class);
```

Setting up the targetInfo Frame

Your `GetTargetInfo` view method should return a `targetInfo` frame, consisting of `target` and `targetView` slots. Alternatively, you can create a frame consisting of these slots and pass it to `OpenRoutingSlip`. As a workaround to a ROM bug, you must also supply an `appSymbol` slot in the `targetInfo` frame containing your appSymbol. Note that `targetInfo.target` could be a multiple item target (see the `CreateTargetCursor` documentation for more info.)

Opening The Slip

You can use `OpenRoutingSlip` to open the slip after setting up slots such as `toRef` and `cc` within the item. You can use code such as the following:

```
/* example using Call Transport */
local item, entry, class, nameRef;

// just for testing, get an Name...
entry := GetUnionSoup("Names"):Query(nil):Entry();

item := TransportNotify('|phoneHome:Newton|, 'NewItem, [nil]);
if item = 'noTransport or not item then
   return 'noTransport;
```

```
class := '|nameRef.phone|;
nameRef := GetDataDefs(class):MakeNameRef(entry, class);
item.toRef :=  [nameRef];
targetInfo := {
   targetView: getroot(),
   target: {}/* for non-CALL transports, add your data here! */,
   appsymbol: kAppSymbol
   };

// returns view (succeeded), or fails: nil or 'skipErrorMessage
OpenRoutingSlip(item, targetInfo);
```

Routing Multiple Items (5/15/96)

Q: How can my application route multiple items at one time?

A: The target must be a "multiple item target" created with the `CreateTargetCursor` function.
For instance, your application could use a `GetTargetInfo` method like:

```
func(reason)
begin
   local t := CreateTargetCursor(kDataClassSymbol, myItemArray);
   local tv := base; // the targetView

      return {target: t, targetView: tv};
end;
```

The first argument to `CreateTargetCursor` is used as the class of the target, which is used to determine what formats and transports are available. You must register formats on that data class symbol in your part's `InstallScript` function.

The item array passed to `CreateTargetCursor` can contain any items, including soup entries or soup entry aliases. If you include soup entry aliases, they will automatically be resolved when accessing items using the `GetTargetCursor` function.

Print formats that have their `usesCursors` slot set to `nil` will automatically print items on separate pages -- print formats must use the target variable to image the current item. To print multiple items, set the format `usesCursors` slot to `true` and use `GetTargetCursor(target, nil)` to navigate through the items.

If either the format (the `usesCursors` slot) or the transport (the `allowsBodyCursors` slot) does not support cursors, the system will automatically split the items into separate Out Box items.

When to Call Inherited ProtoPrintFormat ViewSetupFormScript (1/6/97)

Q: Does it matter when I call the inherited method in my `protoPrintFormat:viewSetupFormScript()`?

A: Yes, you must call the inherited method before doing anything else in the `viewSetupFormScript`.

Among other things, the inherited method sets up the page size. After calling the inherited method, you can call `self:LocalBox()` and get the correct page size. Note that you cannot rely on the `protoPrintFormat.viewBounds` slot value. To position subviews within the print format centered or "full" width or height, use view justifications like centered, right, and full, or use `theEnclosingView:LocalBox()` to determine the exact size of the enclosing view.

Limitations with NewtOverview Data Class (1/8/97)

Q: I want to use code like `CreateTargetCursor('newtOverview, myItemArray)` in my application to simplify my code which handles overviews. Why would my print format throw an exception when I use this method?

A: There are limitations to using the `'newtOverview` symbol as your data class with `CreateTargetCursor`. The biggest limitation is that it requires you to support exactly the set of of datatypes: `['frame, 'text, 'view].` In other words, you must register a `protoFrameFormat` (by default, it handles `'frame` and `'text` dataTypes) and a `protoPrintFormat`. However, there are two other limitations not mentioned in the final documentation: the system does not guarantee that it will call your print format's `formatInitScript` method or a format's `SetupItem` method.

This means that if your print format's `viewSetupFormScript` (or other code in the print format) assumed that the `formatInitScript` has been called, it could cause errors and/or exceptions. The workaround to this would be to set a flag in the `formatInitScript`; if it was not set at the beginning of `viewSetupFormScript`, send your format the `formatInitScript` message. Other problems could occur with `SetupItem`, but you'd probably not see any errors or exceptions until you tried to beam/mail a frame to another device and then tried to Put Away the item.

About the default overview class: when you use `CreateTargetCursor` to prepare a "multiple item target", you may be able to use this special `'newtOverview` symbol as your data class. If your application prints every item on separate pages (in other words, not multiple items on one page) and you want to split beam and mail items into separate items in the Out Box, this might be useful to you. For more information, see the Newton Programmers Guide (not reference) in the Routing chapter "Using the Built-in Overview Data Class" section and the "Move It!" article in the Newton Technology Journal 2.02. Also, check out the MultiRoute DTS sample.

Sound

Finding and Adding Alert Sounds (1/23/97)

Q: Is there a way to add a new sound to the list of available alert sounds?

A: Yes. There is a registry API for alert sounds that is not mentioned in the Newton 2.0 OS final documention. Alert sounds are sound frames with an additional `userName` slot. This slot contains a string that will show up in the alert sound picker.

`RegSound(soundSymbol, alertSoundFrame)`
`soundSymbol` - a unique symbol identifying the sound that incorporates your signature
`alertSoundFrame` - see above
Registers the sound in `alertSoundFrame` with the alert sound picker.

`UnRegSound(soundSymbol)`
`soundSymbol` - symbol of the sound to unregister
Unregisters the sound frame that was registered with the symbol specified by `soundSymbol`.

`SoundList()`
Returns an array of currently registered alert sounds that can be used to construct a popup menu. It returns an array of frames, such that each frame is of the format: `{item: soundName, soundSymbol: theSoundSymbol}`

`GetRegisteredSound(soundSymbol)`
`soundSymbol` - symbol of the sound to return
Returns a sound frame that can be passed to the sound playing functions (for example, the global function `PlaySound`). `soundSymbol` is the symbol used to register the sound.

Stationery

Limits on Stationery Popups (4/30/96)

Q: If I add stationery to Notes, Names, or my application and it is off the bottom of the popup in the new button, I am unable to scroll to it in the stationery popup. Why?

A: There is a problem in the MessagePad 120 and 130 with Newton 2.0 OS constructing popups that contain icons. See the "Picker List is Too Short " Q&A in the Pickers, Popups and Overviews section.

Properly Registering a ViewDef (1/3/97)

Q: When I add a viewDef using `RegisterViewDef` on Newton 2.0 OS, I get the "grip of death" alert ("The package 'MyApp' still needs the card you removed...") when the card the viewDef is on is removed. I don't get the grip of death alert when using Newton 2.1 OS. How can I keep the grip of death alert from appearing?

A: In Newton 2.0 OS, you must `EnsureInternal` the second argument to the `RegisterViewDef` global function. The `RegisterViewDef` global function was changed in Newton 2.1 OS to automatically `EnsureInternal` the second argument.

Note: `EnsureInternal`-ing something that has been `EnsureInternal`-ed is a very fast operation so you don't need to worry about checking which platform you are running on.

System Services, Find, and Filing

Preventing Selections in the Find Overview (2/5/96)

Q: When I use `ROM_compatibleFinder` in Newton 2.0, the overview of found items contains checkboxes for each item, allowing the user to attempt to route the found items. Since my found items are not soup items, various exceptions are thrown. How can I prevent the checkboxes?

A: What you do depends on how you want to handle your data. There are basically two cases. The first case is when you want no Routing to take place (Routing refers to Delete, Duplicate, and the ability to move the data using transports like Beam or Print). The second case is when you want some or all of the Routing to occur.

The first case is easy. Just add a `SelectItem` slot to the result frame, set to `nil`. For example:

```
AddArraySlot(results,
     {_proto: ROM_compatibleFinder,
      owner: self,
      title: mytitle,
      SelectItem: nil,   // prevents checkboxes
      items: myresults});
```

The second case is more complex. The problem is that there are many variants. The best strategy is to override the appropriate methods in your finder to gain control at appropriate points. This may be as simple of overriding `Delete` to behave correctly, or as complex as replacing `GetTarget` and adding appropriate layouts. See the Newton DTS Q&A "Creating Custom Finders" for more information.

Creating Custom Finders (2/5/96)

Q: My application uses more than one soup, so `ROM_soupFinder` is not appropriate, but `ROM_compatibleFinder` seems to throw many exceptions. Which should I use?

A: The answer depends on how much modification you will make. What you need is documentation on how they work and what you can override:

Each of the finder base protos (soupFinder and compatibleFinder) are magic pointers, so can create your own customizations at compile time.

So to do a `soupFinder` based item you could do something like:

```
DefConst('kMySoupFinder, {
   _proto: ROM_soupFinder,

   Delete: func()
   begin
      print("About to delete " & Length(selected) && "items") ;
      inherited:Delete() ;
   end,
}) ;
```

Most of these routines are only callable by your code. They should not be overwritten. Those routines that can be safely overriden are specified.

Some of methods and slots are common to both types of finders:

`finder.selected`
An array of selected items stored in an internal format. All you can do with this array is figure out the number of selected items by taking the Length of this array.

`finder:Count()`
Returns an integer with the total number of found items.

`finder:ReSync()`
Resets the finder to the first item.

`finder:ShowFoundItem(item)`
Displays the item passed. item is an overview item that resides in the overview's items array.

`finder:ShowOrdinalItem(ordinal)`
Display an item based on the symbol or integer passed in ordinal:
`'first` - the first found item
`'prev` - the previous item
`'next` - the next item
`<an-integer>` - display the nth item based on the integer.

Under no circumstances should you call or override:

```
finder:MakeFoundItem
finder:AddFoundItems
```

ROM_SoupFinder

SoupFinder has the following methods and slots:

All the documented items from the simple use of soupFinder as documented in the Newton Programmer's Guide 2.0.

`soupFinder:Reset()`
Resets the soupFinder cursor to the first found entry. In general, you should use the ReSync method to reset a finder.

`soupFinder:ZeroOneOrMore()`
Returns 0 if no found entries, 1 if one found entry or another number for more than one entry.

`soupFinder:ShowEntry(entry)`
causes the finding application to display entry. This may involve opening the application and moving it to that item.
This does not close the findOverview.

`soupFinder:SelectItem(item)`
mark the item as selected.

If this method is set to `nil` in the soupFinder proto, items will not have a checkbox in front of them (not selectable).

`soupFinder:IsSelected(item)`
Returns true if the item is selected.

`soupFinder:ForEachSelected(callback)`
Calls callback function with each selected item. The callback function has one argument, the entry from the soup cursor.

`soupFinder:FileAndMove(labelsChanged, newLabel, storeChanged, newStore)`
File and/or move the selected items.
`newLabel` is the new label if and only if `labelsChanged` is `true.`
`newStore` is the new store if and only if `storeChanged` is `true.`

Developers can override this, though they may want to call the inherited routine to do that actual work. Note that `FileAndMove` can be called even if no items are selected. If you override this method you MUST check if there are selected items by doing:

```
   if selected then
      // do the work
```

`soupFinder:FileAs(labels)`
Deprecated. Do not use.

`soupFinder:MoveTo(newStore)`
Deprecated. Do not use.

`soupFinder:Delete()`
Deletes all selected items from read/write stores.

Developer can override. Note: if you override this, the crumple effect will still happen. There is no way to prevent the ability to delete the items or prevent the crumple effect at this time.

`soupFinder:GetTarget()`
Returns a cursor used by routing.

The following methods should not be called or modified:
`soupFinder.MakeFoundItem`
`soupFinder.AddFoundItems`

ROM_CompatibleFinder

`compatibleFinder:ShowFakeEntry(index)`
Show the `index`'th item from the found items. Note that items will likely be an array of the found items.

`ShowFakeEntry` should behave just like `ShowFoundItem`. In other words, it should open the application then send a `ShowFoundItem` to the application.

`compatibleFinder:ConvertToSoupEntry(item)`
Return a soup entry that corresponds to the item. item is an item from the found items array.

The following methods are defined to be the same as the soupFinder:

```
FileAs, MoveTo, Delete, IsSelected, SelectItem,
ForEachSelected, GetTarget, FileAndMove
```

Note that this causes problems in some cases: most notably, the `ForEachSelected` call is expected to return an array of soup entries. The chances are you will need to override most of those methods. See `soupFinder` for a description of what the methods are supposed to do.

How to Interpret Return Value of BatteryStatus (5/6/96)

Q: I am trying to determine whether the Newton device is plugged in and to obtain other battery status information. Many slots have a `nil` value in the frame returned by the `BatteryStatus` global function. How do I interpret these values?

A: A value of `nil` is returned if the underlying hardware cannot determine the correct information. Some hardware is limited in the amount of information that it can return. Future hardware may fill in more slots with authoritative non-nil values.

How to Create Application-specific Folders (5/14/96)

Q: I would like to programatically create folders so that they are available as soon as the application is open. What is the best approach to add application-specific folders?

A: You can use the global functions `AddFolder` and `RemoveFolder` to modify the folder set for a given application.

```
AddFolder(newFolderStr, appSymbol)
newFolderStr - string, the name of the new folder
appSymbol - symbol, application for local folder
result - symbol, the folder symbol of the newly added folder.
```

`AddFolder` takes a folder name and creates a new folder for the application.

`AddFolder` returns the symbol representing the tag value for the new folder. Please note that the symbol may be different from the value returned by using `Intern()` on the string. In particular, folder names with non-ASCII folders are supported. If a folder with the name already exists, the symbol for the pre-existing folder is returned and a new folder is not created.

There is a limit on the number of unique folders an application can support. If the limit is exceeded, `AddFolder` returns `NIL` and a new folder is not added. With the Newton 2.0 OS, the current limit is twelve global folders and twelve local folders.

```
RemoveFolder(folderSym, appSymbol)
folderSym - symbol, the folder symbol of the folder to remove
appSymbol - symbol, the application for which to remove the folder
result - undefined; do not rely on the return value of this function.
```

`RemoveFolder` can be used to remove a folder from the available list for an application. If items exist in a folder that is removed, the only way users can see the items is by selecting "All Items" from the folder list.

Changing the ProtoStatusButton's Text in ProtoStatusTemplate (1/15/97)

Q: I am using a `protoStatusTemplate`-based view and am trying to rename the primary button through the `protoStatusTemplate`'s setup frame. After doing this, I get an exception when I tap on the renamed button. What am I doing wrong?

A: You are not doing anything wrong. There is a bug in `protoStatusTemplate` which will cause the primary button to function incorrectly if you do not include a `buttonClickScript` in the setup frame.

When you specify a frame in the `primary` slot of the values frame of the setup, the primary button uses the `text` slot and the `buttonClickScript` slot of that frame to initalize itself. Unfortunately, it does not check to see if either of those slots exist before trying to use them. The result is that an exception is thrown when you tap the button.

To work around this bug you must add a `buttonClickScript` to the primary frame. From that method you will typically call your base view's `CancelRequest` method.

Here is a code example:

```
// Add a buttonClickScript method which just calls the application's
CancelRequest method.
local viewSetValues :=  {
   primary:
      {
      text: "Stop",
      buttonClickScript: func()
GetRoot().(kAppSymbol):CancelRequest('userCancel)
      }
   };

local viewSet := {
      appSymbol: kAppSymbol,
      name: "The Name",
      values: viewSetValues
      };

// Setup the status template
statusView:ViewSet( viewSet );
```

Text and Ink Input and Display

ProtoPhoneExpando Bug in Setup1 Method (2/6/96)

Q: I am having a problem using `protoPhoneExpando` under Newton 2.0 OS. Something is going wrong in the `setup1` method. Is this a known bug?

A: Yes, this is a known bug. `protoPhoneExpando` (and the entire expando user interface) have been deprecated in the Newton 2.0 OS, and are only supported for backward compatibility. If possible, you should redesign your application to avoid the expandos.

The problem seems to be that the expando shell is sending the `setup1` and `setup2` messages to the template in the `lines` array. These methods in `protoPhoneExpando` rely on information that isn't created until the view is actually opened.

We're investigating solutions to this problem. You can usually hack around the problem by placing a `labelCommands` slot in the template which has an array of one element, that element being the label you want to appear in the phone line. For example: `labelCommands: ["phone"]`.

This hack works only if your `protoPhoneExpando` doesn't use the `phoneIndex` feature. If it does, you'll have problems that are harder to work around.

Pictures in clEditViews (2/6/96)

Q: Is there a API or procedure that allows an application to write objects such as shapes, PICTs, or bitmaps to a note in the Notes application?

A: There is no API for Notes specifically. The Notes "Note" view is basically a plain old `clEditView`, and `clEditViews` can contain pictures (in addition to ink, polygons, and text) in the Newton 2.0 OS.

The Newton Programmer's Guide 2.0 (in the "Built-In Applications and System Data" chapter) contains a description of the types of children you can create in the Notes application.

This is really a description of the frames you need to put in the `'viewChildren` slot of a `clEditView` to create editable items. `'para` templates are text and ink text, `'poly` templates are drawings and sketch ink, and `'pict` templates are images.

To add a picture to a `clEditView`, you need to create an appropriate template and then add it to the `viewChildren` array (and open the view or call `RedoChildren`) or use the `AddView` method to add it to an existing view (then `Dirty` the view.) See the item "Adding Editable Text to a clEditView" elsewhere in the Q&As for details.

The template for `'pict` items needs to contain these slots:

`viewStationery`: Must have the symbol `'pict`
`viewBounds`: A bounds frame, like `RelBounds(0,0,40,40)`
`icon`: A bitmap frame, see `clPictureView` docs

For other slots, see the documentation for the `clPictureView` view class.

Horizontal Scrolling, Clipping, and Text Views (2/7/96)

Q: I want to draw 80 columns in a `clParagraphView` that's inside a smaller view and be able to scroll back and forth. When I try this, it always wraps at the bounds of the parent. How can I create a horizontal scrolling text view?

A: Normal paragraph views are written so that their right edge will never go beyond their parent. This is done to avoid the circumstance where a user could select and delete some text from the left part of a paragraph in a `clEditView`, leaving the rest of it off screen and unselectable.

What happens is the `viewBounds` of the `clParagraphView` are modified during creation of the view so that the view's right edge is aligned with the parent's right edge. After that, wrapping is automatic.

The so-called "lightweight" text views do not work this way. You can force a paragraph to be lightweight by: 1) Making sure the `viewFlag` `vReadOnly` is set, 2) making sure `vCalculateBounds` and `vGesturesAllowed`, are off, and 3) not using `tabs` or `styles`. Lightweight text views are not editable, but you can use `SetValue` to change their `text` slots dynamically.

If you must use an editable `clParagraphView` or if tabs or styles are required, there is another workaround. The code to check for clipping only looks one or two levels up the parent chain, so you could nest the paragraph in a couple of otherwise useless views which were large enough to prevent clipping, and let the clipping happen several layers up the parent chain.

How to Intercept Keyboard Events (5/6/96)

Q: How do I intercept hardware keyboard events or "soft" keyboard events?

A: You can implement view methods that are called whenever the user presses a key on software or external (hardware) keyboards.. There are two keyboard-related methods associated with views based on the `clParagraphView` view class:

the `viewKeyDownScript` message is sent when a key is pressed.
the `viewKeyUpScript` message is sent when a key is released.

Both methods receive two arguments: the character that was pressed on the keyboard and a keyboard flags integer. The keyboard flags integer encodes which modifier keys were in effect for the key event, the unmodified key value, and the keycode. The layout of the keyboard flags integer is shown in the section below, "Keyboard Flags Integer". The modifier key constants are shown in the section "Keyboard Modifier Keys".

`ViewKeyUpScript` and `ViewKeyDownScript` are currently called using parent inheritance. Do not rely on this behavior: it may change in future ROMs.

If you want the default action to occur, these method must return `nil`. The default action for `ViewKeyDownScript` is usually to insert the character into the paragraph. (There may be other default actions in the future.) If you return a non-nil value, the default action will not occur.

You must include the `vSingleKeyStrokes` flag in the `textFlags` slot of your view for the system to send the `ViewKeyDownScript` or `ViewKeyUpScript` message for every key stroke. If you do not specify `vSingleKeyStrokes`, keyboard input may be dropped if a lot of key strokes are coming in.

The hard keyboard auto repeats with the following event sequence:

keydown -- keydown -- keydown -- keydown...

The soft keyboard auto repeats with this sequence:

keydown -- keyup -- keydown -- keyup -- keydown -- keyup...

Do not rely on this order, it may change in future ROMs.

ViewKeyDownScript

`ViewKeyDownScript(char, flags)`
This message is sent to the key view when the user presses down on a keyboard key. This applies to a hardware keyboard or an on-screen keyboard.

`char` The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be 0.

`flags` An integer that specifies which modifier keys were pressed, the unmodified key value, and the keycode. The modifier key constants are shown in the section "Keyboard Modifier Keys".

ViewKeyUpScript

`ViewKeyUpScript(char, flags)`
This message is sent to the key view whenever the user releases a keyboard key that was depressed. This applies to a hardware keyboard or an on-screen keyboard.

`char` The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be 0.

`flags` An integer that specifies which modifier keys were pressed, the unmodified key value, and the keycode. The modifier key constants are shown in the section "Keyboard Modifier Keys".

Keyboard Flags Integer

Bits	Description
0 to 7	The keycode.
8 to 23	Original keycode. The 16-bit character that would result if none of the modifier keys were pressed.
24	Indicates that the key was from an on-screen keyboard. (kIsSoftKeyboard)
25	Indicates that the Command key was in effect. (kCommandModifier)
26	Indicates that the Shift key was in effect. (kShiftModifier)
27	Indicates that the Caps Lock key was in effect. (kCapsLockModifier)
28	Indicates that the Option key was in effect. (kOptionsModifier)
29	Indicates that the Control key was in effect. (kControlModifier)

Keyboard Modifier Keys

You use the keyboard modifier key constants to determine which modifier keys were in effect when a keyboard event occurs.

Constant	Value
`kIsSoftKeyboard`	(1 << 24)
`kCommandModifier`	(1 << 25)
`kShiftModifier`	(1 << 26)
`kCapsLockModifier`	(1 << 27)
`kOptionsModifier`	(1 << 28)
`kControlModifier`	(1 << 29)

How to Keep Multiple Keyboards Open (8/30/96)

Q: I want my `protoKeyboard`-based keyboard to be open at the same time as other keyboards. When my keyboard opens, it seems like any other keyboard closes. How do I keep multiple keyboards open?

A: When a `protoKeyboard`-based view opens, it closes the last-opened `protoKeyboard`-based view. However, you need not use `protoKeyboard`.

Instead, you can base your keyboard on a different view type (for instance, `protoDragger`) and use the `RegisterOpenKeyboard` view message to register the keyboard with the system. Using `RegisterOpenKeyboard` will ensure that the caret is set up properly and allows you to track the caret changes with the `viewCaretChangedScript` view message if desired.

Adding a Local Keyboard to a ProtoKeyboardButton-based Button (1/14/97)

Q: I have an application-specific keyboard that I would like to have appear only in my application's `protoKeyboardButton`-based keyboard list. Is this possible?

A: Yes, `protoKeyboardButton` has a method called `SetKeyboardList` that lets you do this. `SetKeyboardList` takes two arguments. The first argument is an array of keyboard symbols to add to the list. The second argument is an array of keyboard symbols to remove from the list. Note that the keyboard symbols of the built-in keyboards are listed on pages pages 8-26 and 8-27 of the Newton Programmer's Guide.

To create a local keyboard, your keyboard view must be declared either to the keyboard button view or to a view within in its parent view chain. It is common to declare the keyboard view in your application's base view. When you declare the keyboard view, it must be declared using the keyboard's `keyboardSymbol`.

There are three additional slots that your keyboard template must have:
1) a `preallocatedContext` slot with the symbol of the keyboard
2) a `userName` slot with the name that will appear in the `protoKeyboardButton` popup
3) a `keyboardSymbol` slot with your keyboard's symbol

The `preallocatedContext` slot and the `keyboardSymbol` slot must be the same symbol. Note that the `keyboardSymbol` slot is required, but the `preallocatedContext` slot is additionally necessary to avoid exceptions on devices prior to Newton 2.1 OS.

Next, in the `viewSetupDoneScript` of the `protoKeyboardButton`-based view, send the button a `SetKeyboardList` message with your keyboard's symbol. For instance, you might have the following `viewSetupDoneScript`:

```
viewSetupDoneScript: func()
   begin
      :SetKeyboardList( [kMyKeyboardSymbol], nil );

      // Be sure to call the inherited viewSetupDoneScript method!
      inherited:?viewSetupDoneScript();
   end;
```

If you want to dynamically change the keyboard list, you can also override the `buttonClickScript`. You must first call `SetKeyboardList`, then call the inherited `buttonClickScript`.

All additions and subtractions are removed from the list when your `protoKeyboardButton`-based view is closed.

Getting Digital Ink to the Desktop (1/17/97)

Q: I want to get ink (for instance, a signature) from my Newton device to a desktop machine. How do I do that?

A: The easiest way to get digital ink to the desktop is to convert it into a bitmap on the Newton device, and send the bitmap up to the desktop via the Desktop Integration Libraries (DILs). Another common technique is to convert the ink into an array of (x,y) points and send that array to the desktop for it to convert into whatever format is suitable.

Take a look at the DTS Sample Code projects, "InkForm" and "InkTranslate". They offer some pointers on how to do this. Depending on how and when you want to do the translation, you'll either want to use the view method `ViewIntoBitmap`, or the global function `GetPointsArray`, or a set of functions from the Recognition chapter, particularly `GetStroke` and `GetStrokePointsArray`.

If you need DIL sample code, the DTS Sample Code project "SoupDrink" may be helpful to you.

CHANGED: Constraints on Keyboards Sizing to the View (4/7/97)

Q: I am having a problem with dynamically adjusting the size of keyboards. According to the documentation, adjusting the size of my keyboard view should cause the keys to size correctly to the bounds of the view. This does not happen. If I set the viewbounds of the keyboard (a full alphanumeric keyboard) to anything less than 224x80, the keys scrunch up only taking up about half the view (horizontally). They seem to size fine vertically. This happens even if I set the viewbounds to 222 (only 2 pixels shorter.) What is going on?

A: It turns out the the documentation does not give the full story. The final size of the keys in a keyboard is constrained by the smallest fractional key unit width you specify in the keyboard. To understand key units and key dimensions, read the "Key Dimensions" section of the Newton Programmer's Guide (pages 8-35,6). You can also find this information in the "Key Descriptor Constants" section of the Newton Programmer's Reference.

In addition to calculating the size (in key units) of the longest key row, the `clKeyboardView` also finds the smallest key unit specified in the keyboard and uses this to constrain the final horizontal size. It calculates a minimal pixel size for the keyboard and makes sure that the final keyboard size is an integral multiple of this value. For example, if the smallest size is 10 pixels, then the final keyboard can be 10 pixels or 20 pixels, but not 15 pixels. If the view is 15 pixels, the keyboard will be 10 pixels.

The calculation for this minimal size is:

```
   m = w * (1/s)

m - minimal size
w - width of the longest keyboard row in key units
s   - numeric equivelent for smallest keyboard unit specified in the
keyboard:
   (keyHUnit = 1,  keyHHalf = 0.5, keyHQuarter = 0.25, keyHEighth =
0.125)
```

For the built-in ASCII keyboard in current ROMs, the longest row is 14 key units, the smallest key unit used is `keyHQuarter`, so the minimal width for the ASCII keyboard is:

```
   m = 14 * (1 / 0.25) = 14 * 4 = 56 pixels.
```

The keyboard will always be an integral multiple of 56 pixels in width. Note that 224 pixels is exactly 4 * 56. By changing the width to 223, the keyboard now becomes 168 pixels wide.

Transports

Adding Child Views to a ProtoTransportHeader-based View (1/19/96)

Q: How can I add child views to a `protoTransportHeader`-based view?

A: First, you need to specify an `addedHeight` slot. The height of the transport header will be increased by this amount.

Next, add the following code to the `viewSetupFormScript` method of your `protoTransportHeader` view. This works around a bug with `protoTransportHeader`:

```
self.stepChildren := SetUnion( self._proto.stepChildren,
     self._proto._proto.stepChildren, true );
```

Finally, use NTK as you normally would to create the child views.

How to Omit Default Transport Preference Views (5/6/96)

Q: I want to omit some transport preferences that appear automatically. If I specify `nil` for the `sendPrefs`, `outboxPrefs`, or `inboxPrefs` slots in my transport preferences template, opening the slip throws -48204. What is going wrong?

A: The documentation states if you donÕt want to include `sendPrefs`, `outboxPrefs`, or `inboxPrefs` in your preferences dialog to set those slot to `nil`. Due to a bug in the cooresponding views for those preference items, -48204 is thrown when an attempt is made to open the views. This will be fixed in a future ROM.

How to Stop ProtoAddressPicker Memory (9/20/96)

Q: How do I stop `protoAddressPicker` from remembering the last people picked?

A: `protoAddressPicker` has a slot called `useMemory` that was left out of the documentation. If this slot is set to `nil`, the memory mechanism will be disabled.

ReceiveRequest Requests Incorrect After Using RemoveTempItems (10/1/96)

Q: If my transport calls the owner method `RemoveTempItems` when items are selected, the `ReceiveRequest` message sometimes has bogus request arguments (for instance, the `cause` is set to `'remote`). Is this a known bug?

A: Yes, this is a known bug in `RemoveTempItems`. The call to `RemoveTempItems` does not clear out the cache of selected items correctly. This will be fixed in a future version of the Newton OS.

To workaround the problem, assume the following: If your transport's communications channel is not currently connected and you receive a request with its cause slot set to `'remote`, assume that the cause slot is actually `'user` and act accordingly. If you receive a request with its `'cause` slot set to `'remote` and your transport's communications channel is connected then perform the appropriate action for receiving remote items.

Filing Sent Entries in the Out Box (1/14/97)

Q: If a user has selected to file sent entries into a folder that has been deleted, my transport throws an exception when it calls ItemCompleted. Why is this problem occuring?

A: This is caused by a bug in `ItemCompleted`. To work around this, you should check to make sure the folder exists before calling `ItemCompleted`. If it does not exist, then set the transport's `'outboxFiling` preference to `nil`. Here is a code example:

```
// This code assumes that the current receiver (self) is your
transport
if NOT GetFolderStr( :GetConfig( 'outboxFiling ) ) then
```

```
:SetConfig( 'outboxFiling, nil );
```

Documentation on the InboxFiling Preference (1/15/97)

Q: The documentation on In Box filing appears to be incorrect. It says that incoming items will be filed when they are received, however it appears that they are actually filed when they are read. Is the documentation incorrect?

A: Yes, the documentation is incorrect. Items are filed when they have been read, not when they have been received. The default for `inboxFiling` is `nil`.

Note that if the In/Out Box is not the backdrop application, filing does not occur until you close the In/Out Box. If the In/Out Box is the backdrop application, filing occurs when the user switches between the overview and the main view.

Utility Functions

What Happened to FormattedNumberStr (2/12/96)

Q: The Newton 1.x documentation and OS included a `sprintf`-like function for formatting numbers called `FormattedNumberStr`. The Newton Programmer's Guide 2.0 First Edition (beta) says this function is no longer supported. How do I format my numbers?

A: You may continue to use `FormattedNumberStr`. Here is the `FormattedNumberStr`API that is supported. `FormattedNumberStr` should be considered to have undefined results if passed arguments other than those specified here.

`FormattedNumberStr(number, formatString)`
Returns a formatted string representation of a real number.

`number` A real number.
`formatString` A string specifying how the number should be formatted.

This function works similar to the C function `sprintf`. The `formatString` specifies how the real number should be formatted; that is, whether to use decimal or exponential notation and how many places to include after the decimal point. It accepts the following format specifiers:
`%f` Use decimal notation (such as "123,456.789000").
`%e` Use exponential notation (such as "1.234568e+05").
`%E` Use exponential notation (such as "1.234568E+05").

You can also specify a period followed by a number after the % symbol to indicate how many places to show following the decimal point. (`"%.3f"` yields `"123,456.789"` for example.)

Note: `FormattedNumberStr` uses the current values of `GetLocale().numberFormat` to get the separator and decimal characters and settings. The example strings above are for the US English locale.

Known Problems

Other specifiers
Do *not* use other `formatStrings`. Previous releases of the documentation listed `%g` and `%G` as supported specifiers. The behavior of these specifiers has changed with the Newton 2.0 OS. Given the similarities to the `sprintf` function, it may occur to you to try other sprintf formatting characters. Specifiers other than above have an undefined result and should be considered undocumented and unsupported.

Large numbers
`FormattedNumberStr` does not work properly for numbers larger than `1.0e24`. If the number is very large the function can cause the Newton device to hang.

Small numbers or long numbers
If more than 15 characters of output would be generated, for example because you are using `%f` with large number or a large number of digits following the decimal, `FormattedNumberStr` has undefined results, and can cause the Newton device to hang.

Rounding
`FormattedNumberStr` does not guarantee which direction it will round. In the Newton 2.0 OS, it rounds half cases down rather than up or to an even digit. If you need a precisely rounded number you should use the math functions `Ceiling`, `Floor`, `NearbyInt`, or `Round` with suitable math.

Trailing decimals
In early releases of the Newton 1.0 OS, there was a bug in `FormattedNumberStr` that caused a trailing decimal character to be added when zero decimal positions was specified. That is, `FormattedNumberStr(3.0, "%.0f")` resulted in `"3."` not `"3"`. To properly test for and remove this unwanted extra character you must be sure to use the character specified in the Locale settings and not assume the decimal character will be a period.

Backlight API (4/19/96)

Q: What is the API to check for and use the backlight?

A: There are three relevant pieces of information:

Checking for the backlight

To check if the backlight is there, use the `Gestalt` function as follows:

```
// define this somewhere in your project
// until the platform file defines it (not in 1.2d2)
constant kGestalt_BackLight := '[0x02000007, [struct, boolean], 1];

local isBacklight := Gestalt(kGestalt_BackLight);

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

Status of the backlight

To find the current state of the backlight, use the following function:

```
BackLightStatus()
```
return value = nil (backlight is off) or non-nil (backlight is on)

Changing backlight status

To turn the backlight on or off, use:

```
BackLight(state)
```
return value - unspecified
state - nil (turn backlight off) or non-nil (turn backlight on)

Unusual Sort Order/Case Sensitivity in Swedish Locale (1/16/97)

Q: When I set the unit to the Swedish locale and use the `StrPos` global function to search for a ':' character, it finds other characters such as '.' or ';'. Isn't that a bug? How can I reliably search for these characters in any locale?

A: The global function `StrPos` and many of the other string functions are case insensitive - they treat upper and lowercase letters as being identical. In other languages, characters such as accented letters may be considered as different cases of the base letter, so they are treated as identical as well. In the Newton OS model, the concepts 'same case' and 'same position in the sorting order' are not distinguished, so all cases of a letter will sort to the same position. Going backwards, all characters that sort identically are considered to be different cases of the same letter. Well, in the Swedish sort order, many punctuation characters are defined to sort to the same place, and so the case insensitive functions in the Newton device treat the characters as identical. Many special and punctuation characters are grouped this way, but perhaps the most surprising set is ? ¡ : , . ; ¿ and !, which all sort to the same position and so are treated as identical in Swedish by `StrPos` and other case insensitive functions.

To search a string for a particular character using a case sensitive search, use the `CharPos` function instead of `StrPos`.

NEW: Time Zones, GMT, Daylight Savings, and Newton Time (3/4/97)

Q: There don't seem to be any functions in the Newton OS for converting between standard time values, such as finding the time in a different time zone, or GMT time. I know it's possible because the built in Time Zones application does it. How can I do this in my own application?

A: The Newton OS doesn't actually have the concept of time zones. Instead, for each city if keeps track of the offset (in seconds) from GMT for that city. You can find this in the `'gmt` slot of a city entry, which can be gotten with the `GetCityEntry` global function. See the "Built In Apps and System Data" chapter of the Newton Programmers Guide for details. Note that the docs incorrectly say the `gmt` slot contains the offset in minutes, when it is actually specified in seconds. The current location is available in the `'location` slot of the user configuration frame. Use `GetUserConfig('location)` to access it. The global function `LocalTime` can be used to convert a time to the local time in a distant city.

A simple way to get the local time from a GMT time would be to create a city entry representing GMT (gmt offset 0, no daylight savings) and then use `LocalTime` to compute the delta between the current city and the GMT city, then add the delta to the given GMT time. `LocalTime` can be used directly to go the other way--getting the GMT time from the local time.

`LocalTime(`*`time`*`,` *`where`*`)`
`time` - a time in minutes in the local (Newton device) zone, for example as returned from the `Time` function
`where` - a city entry, as returned from `GetCityEntry`
`result` - a time in minutes in the *where* city, adjusted as necessary for time zone and daylight savings.

`LocalTime` tells you the local time for the distant city, given a time in the current city. For example, to find out the time in Tokyo:

```
Date(LocalTime(time(), GetCityEntry("Tokyo")[0]))
#C427171  {year: 1997, month: 2, Date: 22, dayOfWeek: 6,
    hour: 8, minute: 1, second: 0, daysInMonth: 28}
```

Because the Newton OS doesn't have time zones, it can't keep track of daylight savings time by changing zones (for example, from Pacific Standard Time to Pacific Daylight Time). Instead, it uses a bunch of rules that tell it when to set the time ahead or back, and by how much. The global function `DSTOffset` can be used to find out how much these daylight savings time rules have adjusted a given time for a given city.

`DSTOffset(`*`time`*`,` *`where`*`)`
`time` - a time in minutes in the where city
`where` - a city entry, as returned from `GetCityEntry`
`result` - an integer, number of minutes that daylight savings adjusted that time in that city.

`DSTOffset` tells you what the daylight savings component is of a given time in a given location. This component would need to be subtracted from the result of the global function `Time` to get a non-daylight-adjusted time for the current location.

```
// it's currently 2:52 PM on 3/4/97, no DST adjustment
DSTOffset(Time(), GetCityEntry("Cupertino")[0]);
#0        0

// but during the summer, DST causes the clocks to "spring
forward" an hour.
DSTOffset(StringToDate("6/6/97 12:34"),
GetCityEntry("Cupertino")[0]);
#F0       60
```

NEW: Square Root of Negative Number Bug (3/4/97)

Q: When I call `Sqrt` with a negative number on the Newton, or use `Compile` in the NTK Inspector, I get a strange result. However, if I just type `sqrt(-2)` into the listener I get a different strange result. What's going on?

```
call compile("sqrt(-2)") with ()
#4412F2D  -1.79769e+308

sqrt(-2)
#440DE05  1.00000e+999
```

A: There is a floating point library bug in `Sqrt` on the Newton OS. When passed a negative number, the large positive value is returned instead of a not-a-number value. You can work around it using `Pow(x, 0.5)` instead of `Sqrt(x)` if there is no way to guarantee that the value passed to `Sqrt` is non-negative, or simply check and see if the argument is less than 0 and return a not-a-number constant.

The reason `sqrt(-2)` works differently when you type it into the NTK Inspector is because of a compiler process known as constant folding. `Sqrt` can be evaluated at compile time if you pass it a constant argument. So what's really happening is that NTK is evaluating the `Sqrt` function during the compile phase and passing the resulting floating point number (or rather, not-a-number) to the Newton device where it's promptly returned. An NTK real number formatting limitation displays non-a-number values and infinities as `1.00000e+999` rather than as some other string. You can use `IsNAN` to determine if a real number is a not-a-number value.

You can avoid constant folding and force evaluation on the Newton device by using a variable. For instance:

```
x := -2;
y := sqrt(x);
#C4335B1  -1.79769e+308
```

Also, note that `FormattedNumberStr` does not properly handle not-a-number values. (it returns "Number too small.")

NEW: Making Use of the Serial Number Chip (4/3/97)

Q: I would like to get the serial number from the units that support it, as either an integer or real number. How can I do this?

A: You probably don't really want to do this. The serial number is an 8-byte binary object, so you could use `ExtractByte` or `ExtractWord` or possibly `ExtractLong` to get the bytes out in integer form, then do something with them. However, keep in mind that NewtonScript integers are only 30 bits wide, whereas the serial number is 64 bits wide, so you'll never be able to put all the information contained in the serial number into a single integer. (3 integers would be required.)

That is, let us suppose you added up the value of all the bytes in a serial number. You would get a single NewtonScript integer, but it would also be possible for a different serial number to produce the same integer. (Just swap the positions of two of the bytes.) Same goes for XOR or any checksumming scheme. There's just no way to reduce 64 bits of information to 30 bits without allowing loss of uniqueness. (If you come up with a way, let us know, it'd make a great compression algorithm!)

Real numbers aren't suitable either, for much the same reason. It's true that in NewtonScript reals are 8 bytes wide, but they use the IEEE 64-bit real number specification, and so not all combinations of 8 bytes are considered unique. That is, you might think about taking the serial number result and using `SetClass` to change it's class to `'real`, which would effectively "cast" the 8-byte object to a real number. This is a bad idea, because real numbers are interpreted using bitfields with special meanings, and it's possible for two real numbers to have different binary representations and still evaluate as equal using the '=' operator. (Any two not-a-number values will do this.)

Serial numbers are best treated as strings or as 8-byte binary objects, so that no data is lost. `StrHexDump` is the best way to format the serial number object for humans to read. If you want to break it up to make it more easily readable, you could do something like this:

```
local s := StrHexDump(call ROM_GetSerialNumber with (), 2);
StrReplace(s, " ", "-", 3);
```

Which produces this string (on my unit):

```
"0000-0000-0154-8423 "
```

Please note that the serial number provided by the chip does NOT match the serial number that Apple Computer and other Newton device manufacturers may put on the outside of the case. When supporting a device, Apple and its licensees will most likely request the user-visible serial number, typically found on a sticker on the case. Please be sure that you present data from the internal chip-based serial number in such a way as to ensure the user will not be confused. (This is the reason the chip-based serial number is not displayed by any software built into the device.)

NEW: Programmatically Cancelling a Confirm Slip (4/3/97)

Q: During an operation, I bring up a slip to ask the user if they really want to abort the operation. Before they answer, the operation may complete or be aborted anyway. I would like to remove the slip if this happens, much like the "Remount" slip is removed when a gripped card is ejected, reinserted, and then re-ejected.

A: There's no way to dismiss a `ModalConfirm` slip, because your code is paused waiting for the result. You can, however, remove an `AsyncConfirm` slip. The return value from `AsyncConfirm` (which is documented in the Newton Programmer's Guide as "unspecified") is actually a reference to the confirm view. Sending that view a `Close` message dismisses the slip. The callback function will not be called if this slip is removed in this way, so make sure your program handles that case.

Views

How to Save the Contents of clEditView (10/4/93)

Q: How can I save the contents of a `clEditView` (the children paragraph, polygon, and picture views containing text, shapes, and ink) to a soup and restore it later?

A: Simply save the `viewChildren` array for the `clEditView`, probably in the `viewQuitScript`. To restore, assign the array from the soup to the `viewChildren` slot,

either at viewSetupFormScript or viewSetupChildrenScript time; or later followed by RedoChildren.

You shouldn't try to know "all" the slots in a template in the viewChildren array. (For example, text has optional slots for fonts and tabs, shapes have optional slots for pen width, and new optional slots may be added in future versions.) Saving the whole array also allows you to gracefully handle templates in the viewChildren array that don't have an ink, points, or text slot. In the future, there may be children that represent other data types.

Adding Editable Text to clEditViews (6/9/94)

Q: How can I add editable text to a clEditView? If I drag out a clParagraphView child in NTK, the text is not selectable even if I turn on vGesturesAllowed.

A: clEditViews have special requirements. To create a text child of a clEditView that can be selected and modified by the user (as if it had been created by the user) you need to do the following:

```
textTemplate := {
   viewStationery: 'para,
   viewBounds: RelBounds(20, 20, 100, 20),
   text: "Demo Text",
};
AddView(self, textTemplate);
```

The view must be added dynamically (with AddView), because the clEditView expects to be able to modify the contents as the user edits this item. The template (textTemplate above) should also be created at run time, because the clEditView adds some slots to this template when creating the view. (Specifically it fills in the _proto slot based on the viewStationery value. The _proto slot will be set to protoParagraph) If you try to create too much at compile time, you will get -48214 (object is read only) errors when opening the edit view.

The minimum requirements for the template are a viewStationery of 'para, a text slot, and a viewBounds slot. You can also set viewFont, styles, tabs, and other slots to make the text look as you would like.

TieViews and Untying Them (6/9/94)

Q: What triggers the pass of a message to a tied view? If I want to "untie" two views that have been tied with TieViews, do I simply remove the appropriate slots from the viewTie array?

A: The tied view's method will be executed as a result of the same actions that cause the main view's viewChangedScript to be called. This can happen without calling SetValue, for example, when the user writes into a view that has recognition enabled, the viewChangedScript will get called.

As of Newton 2.0 OS, there is no API for untying tied views. It may be wise to first check for the existance of an `UntieViews` function, and call it if it exists, but if it does not, removing the pair of elements from the tied view's `viewTie` array is fine.

Immediate Children of the Root View Are Special (11/17/94)

Q: In trying to make a better "modal" dialog, I am attempting to create a child of the root view that is full-screen and transparent. When I do this, the other views always disappear, and reappear when the window is closed. Why?

A: Immediate children of the root view are handled differently by the view system. They cannot be transparent, and will be filled white unless otherwise specified. Also, unlike other views in Newton 2.0 OS, their borders are considered part of the view and so taps in the borders will be sent to them.

This was done deliberately to discourage tap-stealing and other unusual view interaction. Each top level view (usually one application) is intended to stand on its own and operate independently of other applications.

So-called "application modal" dialogs can and should be implemented using the technique you describe with the transparent window as a child of the application's base view.

You can make system modal dialogs with the view methods `FilterDialog` and `ModalDialog`. (See the Q&A "FilterDialog and ModalDialog Limitations" for important information on those methods.)

ViewIdleScripts and clParagraphViews (8/1/95)

Q: Sometimes a `clParagraphView`'s `viewIdleScript` is fired off automatically. (For example, an operation which results in the creation or changing of a keyboard's input focus within the view will trigger the viewIdleScript.) Why does this happen and what can I do about it?

A: The `clParagraphView` class internally uses the idle event mechanism to implement some of its features. Unfortunately, any `viewIdleScript`s provided by developers also execute when the system idle events are processed. Only the "heavyweight" views do this, "lightweight" paragraph views (in other words, simple static text views) do not.

There is no workaround in the Newton 1.x OS or Newton 2.0 OS while using clParagraphView `viewIdleScript`. You can either accept the extra idle script calls, or use another non-`clParagraphView` based view to implement your idle functions.

FilterDialog and ModalDialog Limitations (2/5/96)

Q: After closing a view that was opened with `theView:FilterDialog()`, the part of the screen that was not covered by the `theView` no longer accepts any pen input. `theView` is a `protoFloatNGo`. Is there some trick?

A: There is a problem with `FilterDialog` and `ModalDialog` when used to open views that are not immediate children of the root view. At this point we're not sure if we'll be able to fix the problem.

You must not use `FilterDialog` or `ModalDialog` to open more than one non-child-of-root view at a time. Opening more than one at a time with either of these messages causes the state information from the first to be overwritten with the state information from the second. The result will be a failure to exit the modality when the views are closed.

Here are some things you can do to avoid or fix the problem with `FilterDialog`.

Redesign your application so that your modal slips are all children of the root view, created with `BuildContext`. This is the best solution because it avoids awkward situations when the child of an application is system-modal. (Application subviews should normally be only application-modal.)

Use the `ModalDialog` message instead of `FilterDialog`. `ModalDialog` does not have the child-of-root bug. (`FilterDialog` is preferred, since it uses fewer system resources and is faster.)

Here is some code you can use to work around the problem much like a potential patch would. (This code should be safe if a patch is madethe body of the if statement should not execute on a corrected system.)

```
view:FilterDialog();
if view.modalState then
   begin
      local childOfRoot := view;
      while childOfRoot:Parent() <> GetRoot() do
         childOfRoot := childOfRoot:Parent();
      childOfRoot.modalState := view.modalState;
   end;
```

This only needs to be done if the view that you send the `FilterDialog` message to is not an immediate child of the root. You can probably improve the efficiency in your applications, since the root child is ususally your application's base view, which is a "well known" view. That is, you may be able to re-write the code as follows:

```
view:FilterDialog();
if view.modalState then
   base.modalState := view.modalState;
```

Using Proportional View Alignment Correctly (6/20/96)

Q: I am trying to use proportional view alignment but things don't seem to be working correctly. For instance, if I have a view which is full justified or center justified, proportional view alignment doesn't seem to work at all. Whats wrong?

A: Proportional justification only works if you are using left, right, top, or bottom justification. This is true for both sibling and parent justification.

Proportional justification is very similar to full justification. The view system needs some reference point at which to position the view. If you specify full or center justification, and you are also using proportional justification, the reference point is undefined.

Additionally, if you are using right or bottom justification, you will need to specify negative values for your proportional bounds. For instance, if you want a view to take up the right 30 percent of its parent, you would specify the following view bounds:

```
{left: -30, top: <top>, right: 0, bottom: <bottom>}
```

and the following view justification:

```
vjParentRightH + vjLeftRatio
```

Drag and Drop Caches the Background Bitmap (7/15/96)

Q: I am trying to implement drag scrolling. Although I can scroll the contents of the window, when I drag the item back into the window, it strips away the updated (scrolled) contents and leaves the original (unscrolled) contents behind. How can I get this to work?

A: Unfortunately, you have hit a design limitation of the Drag and Drop implementation. When you send the `DragAndDrop` message to a view, the bitmap for the pre-drag state is cached. Once the drag loop starts, you can not update that cached bitmap. When you "scroll" the view (probably using `RefreshViews`), you update the screen, but the cached bitmap is still there and is used by the `DragAndDrop` routine to update the screen as the dragged item is moved.

NEW: Default and Close Keys in Confirm Slips (2/28/97)

Q: Is there any way to put a keyboard default on a confirm dialog?

A: Yes. For both `ModalConfirm` and `AsyncConfirm`, the 2.0 Newton Programmer's Reference says you may pass three types of things as the buttonList argument: a symbol (`'okCancel` or `'yesNo`), and array of strings, or an array of frames with `'value` and `'text` slots.

In the Newton 2.1 OS, this API has been extended to allow for default key and close key behavior. There are four new symbols that are allowed: `'okCancelDefaultOk`, `'okCancelDefaultCancel`, `'yesNoDefaultYes`, and `'yesNoDefaultNo`. They do the obvious thing, setting the default key as specified and the close key to Cancel or No if those aren't the default. However, using these symbols on a Newton 2.0 OS device will result in the "OK" and "Cancel" buttons always being displayed, even if you specify `'yesNoDefaultYes` or `'yesNoDefaultNo`.

The array-of-frames flavor for the `buttonList` argument allows an additional slot, called `'keyValue`. Supported values for this slot are the symbols `'default` and `'close`, or NIL/not present. `'default` makes the button the default key, and `'close` makes the button activate with the close key. Any other value will cause a problem in the current Newton 2.1 implementation. The `keyValue` slot is ignored on the Newton 2.0 OS.

For compatibility, we recommend avoiding the `'yesNoDefaultYes` and `'yesNoDefaultNo` symbols if you intend to run on both Newton 2.0 and 2.1 devices. Instead, use one of these specifiers:

```
   '[{text: "Yes", value: TRUE, keyValue: default}, {text: "No",
value: NIL, keyValue: close}]
   '[{text: "Yes", value: TRUE}, {text: "No", value: NIL, keyValue:
default}]
```

NEW: Screen Rotation and Linked Views or BuildContext Slips (3/10/97)

Q: I've got a linked view open, and I'm trying have it rotate with `ROM_DefRotateFunc`. When I rotate the screen the base view rotates properly, but the linked view closes. Do I need to add a `ReorientToScreen` slot to the linked view?

A: When the user requests a screen rotation, the OS first checks each immediate child of the root view to see if it will still operate on the rotated screen. Having a `ReorientToScreen` slot in the view tells the OS that this view is OK, so the slot is used first as a token ("magic cookie") to tell the OS that this view knows about rotation. Later during the rotation operation, the `ReorientToScreen` message is sent to your application's base view and to other views that are immediate children of the root view. That method then performs its second function, which is to completely handle resizing the view and its children for the new screen size. (Even views which are small enough so that no resizing is necessary need a `ReorientToScreen` method. That method may need to move the base view to ensure that it remains on-screen after rotation.)

It's convenient to use the `ROM_DefRotateFunc` for this script, since it fills the magic cookie requirement and handles resizing for most views. `ROM_DefRotateFunc` is very simple: it send a close and then an open message to the view. Since well-written applications take screen size into account when they open, this works fine in most cases. However, applications that keep track of internal state that isn't preserved when the app is closed can't use `ROM_DefRotateFunc`, because when the app reopens on the rotated screen, it will look different. Opening a linked subview is one example of this; it doesn't usually make sense to remember that a slip is open, since it's usually closed when your application is closed.

Adding a `ReorientToScreen` method to your linked views wouldn't help; since they are descendents of your base view and not children of the root view, the OS wouldn't handle these views. (It's up to your application to keep its kids under control.) You could change your application so that it kept track of whether the linked views were open or closed, and restored them to the same state when it was reopened. However, this might be confusing to users who closed your app and then opened it again much later.

A better workaround is to implement your own `ReorientToScreen` method, which either resizes the views so they fit on the new screen, or which closes and reopens the views such that the floaters also re-open. By using the `ReorientToScreen` message to handle the special case, you get to do something different during rotation versus during opening at the user request (for example, after tapping on the Extras icon.)

Slips created with `BuildContext` also must be handled carefully during rotation. Because they are themselves children of the root view, they'll each need their own `ReorientToScreen` method or the screen may not be rotatable when they are open or they

won't reopen after rotation. If you use `ROM_DefRotateFunc`, the slip itself will be closed and reopened, and care may need to be taken to ensure the slip properly handles being reopened, and that its connection to its controlling application is not lost.

NEW: How to Get Data From a ProtoTXView's Externalized Data (4/3/97)

Q: I'm using the `protoTXView` text engine in Newton 2.1 OS. How can I get text, styles, pictures, etc. out of the object returned by `protoTXView`'s `Externalize` method without either a) instantiating a `protoTXView` or b) digging in the data structure?

A: You must instantiate a view to get data from the externalized object that `protoTXView` produces. The data structures in that object are not documented or supported. You may be tempted to do this anyway, since it looks as though the data structure is obvious. Don't, it isn't. `ProtoTXView` actually uses several different data formats depending on the complexity and storage destination for the data.

It's actually very easy to instantiate a view based on `protoTXView` to get at the data. Here's one way:

```
local textView := BuildContext(
   {
      _proto: protoTXView,
      viewBounds: SetBounds(0, 0, 0, 0),
      viewFlags: 0,
      ReorientToScreen: ROM_DefRotateFunc,
   });
textView:Open();
textView:Internalize(myExternalizedData);
```

You can now use all the `protoTXView` APIs to get the data from the `textView` object. Don't forget to clean up with `textView:Close()` when you're done.

NEW: Extracting All Text from a ProtoTXView Object (4/3/97)

Q: How can I get all the text out of the `protoTXView` data stored in a soup entry, for example to get the text for sending in email?

A: First, instantiate a dummy view with the data from the soup, as described in the Q&A "How to Get Data From a ProtoTXView's Externalized Data". The `protoTXView` method `GetRangeData` always allocates its storage from the NewtonScript heap, so you will need to copy the data into a destination VBO in chunks. Here's some code to do that. (This code uses a 4K block size, you may wish to adjust that as necessary, add error checking, etc.) `StringFilter` is used to remove the `protoTXView` graphic indicator.

```
   constant kChunkSize := 0x1000; // 4K chunks
   local start := 0;
   local theText := GetDefaultStore():NewVBO('string, length(""));
   BinaryMunger(theText,0,nil, "", 0, nil);    // make VBO into a
proper string
```

```
   while numChars-start > kChunkSize do
      // strip out graphics characters
      StrMunger(theText,start, nil,
               StringFilter(
                  textView:GetRangeData({first: start, last: start :=
start + kChunkSize}, 'text),
                  "\u2206\u", 'rejectAll),
               0, nil);
   // copy remainder
   if start < numChars then
      StrMunger(theText,start,nil,
               StringFilter(
                  textView:GetRangeData({first: start, last:
numChars}, 'text),
                  "\u2206\u", 'rejectAll),
         0, nil);
   // theText now holds plain text from the protoTXView
```

For clarity, the code above does not use `ClearVBOCache` as mentioned in the Q&A, "How to Avoid Resets When Using VBOs". If you are having problems with large VBOs during code like that mentioned above, see that Q&A for more information.