Newton 2.0 OS Q&A's

—

Introduction

This document addresses Newton 2.0 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.

TABLE OF CONTENTS:

Controls and Other Protos

Text and Ink Input and Display

Stroke Bundles

Recognition

Data Storage (Soups)

Drawing and Graphics

Sound

System Services, Find, and Filing

Intelligent Assistant

Built-In Apps and System Data

Localization

Application Design

Optimizing Root 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 root view and the lookup time to find it is slowing me down. 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();
```

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

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)

Note that in NewtonScript, global functions and variables are true globals. This means that you might get name clashes with other possible globals, and 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 this inside the base view. 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.

For instance, we might have the following function declared inside the base view:

```
|MyBaseView:MySIG| := {
   viewClass: clView,
   declareSelf: 'base,
   TestThisView := func()
   begin
      // blah blah blah
   end
};
```

If you now call the function from another template that is not a child of the base view, you might do this:

```
|MyBaseView:MySIG|:TestThisView();
```

Preventing an Application From Opening (6/9/94)

Q: I do not want my application to open, for example because the screen size is too small, or because the OS is the wrong version. 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.

NEW: Creating 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 applicationÕs 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.)

NEW: Responding 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.

NEW: Testing 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 from with app frontmost
3.8. Find data in app using Find All from paperroll
3.9. Find data in all user enterable 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.)
6.2. Check store memory and frames heap, install application, check store memory and frames heap. Do this several times and check for consistency
6.3. Do 6.2. and also check store and frames memory after removing application. Is all/most of the memory restored?
6.4. Check frames heap. Launch & use application. Check heap. Close application. Check heap.
6.5. Does the application add anything to the Preferences App?
6.6. Does the application add Prefs and Help to the "i" icon?
6.7. Does the application add anything to Assist, How Do I?
6.8. Launch with pager card installed
6.9. Check layout issues on MP100 vs. MP110 screen sizes (if application runs in 1.x.)
6.10. 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.
6.11. Open, use, and close the application many times. Check frames heap afterward to check for leaks.
6.12. If application has multiple components and components can be removed separately, verify that application does the right thing when components are missing.

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.
    7.3.3. Make sure that bringing up dialogs or other BuildContext views works correctly.

Views

Saving clEditView Contents to a Soup (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.

Declaring Multiple Levels (6/9/94)

Q: Call the main application view viewA. ViewB is a child of viewA and is declared to viewA. ViewC is a child of viewB and is declared to viewB. ViewB and ViewC are both initially invisible. This causes the ViewC slot in viewB to be nil when the application is first run. Is there any way to access viewC without first opening and then then hiding it?

A: The built-in declare mechanism will not work without opening the view. The declared view frames are not created until the view they are declared to is opened. You may consider trying to declare viewC to viewA, but this will actually illustrate a problem with the declare mechanism--it can get confused in this case because viewC's parent (viewB) may not have been created when the view frame for viewC needs to be allocated.

Depending on what sort of access you need to viewC, you could choose alternative such as

- promoting the shared data from viewC to viewB, where it can be accessed.
- writing your own equivalent of the declare mechanism, with a slot called myViewC in viewB. Have viewC's viewSetupFormScript copy data from myViewC into the view frame being created.

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 editView 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 editView 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. (See the Notarize sample code for additional relavant information.)

The way viewStationery is handled will change in future Newton versions, and we cannot guarantee that the above code will continue to work.

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 OS 2.0, 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 following Q&As for important information on those methods.)

NEW: Arguments to AsyncConfirm and ModalConfirm (12/12/95)

Q: The Newton Programmer's Guide says that I can pass a symbol as the 2nd argument to `ModalConfirm` and `AsyncConfirm`, but doesn't say what symbols to use. What symbols can I use?

A: `ModalConfirm` and `AsyncConfirm` are actually very flexible. You can pass three different things as the 2nd argument (the list of buttons.) These things are:

- a symbol - Supported symbols are `'okCancel` or `'yesNo`.
- an array of strings - for example `["Three", "Two", "One"]`
- an array of frames - each frame has two slots, `'value` and `'text`.
  - `text` - holds the label for the button, a string
  - `value` - holds the result that tapping the button generates.

In `ModalConfirm`, the function returns the result of the user's choice. In `AsyncConfirm`, the call-back function provided as the 3rd argument is called with the result. The result varies depending on what was passed as the 2nd argument.

If a symbol was used, the result is non-`NIL` for the "OK" and "Yes" buttons, and `NIL` for the "Cancel" and "No" buttons. If an array of strings was passed, the result is the index into the array of the item that was chosen. If an array of frames was passed, the result is the contents of the `value` slot for the item that was chosen.

NEW: 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 madeÑthe 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;
```

NewtApp

NEW: Creating Preferences in a NewtApp-based Application (01/31/96)

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 template for your preferences slip there 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
```

In the preferences frame you will find a slot called `internalStore`. Setting this slot to `true` will force the NewtApp framework to save all new items on the internal store.

NEW: Creating an About Slip in a NewtApp-based Application (01/31/96)

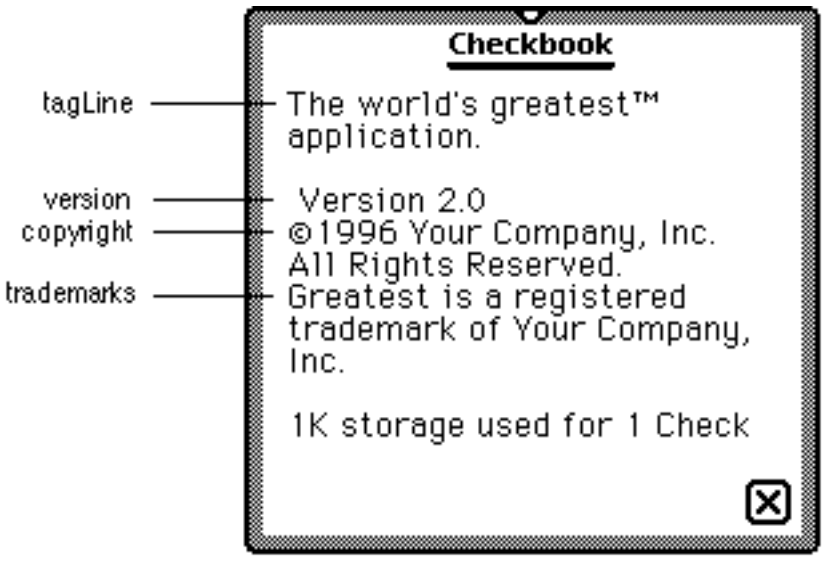
Q: How do I create my own About slip in a NewtApp-based application?

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.

Here is 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`. Then use the NTK `GetLayout` function to place a template of your view in that slot. A view will be created from that template and opened when the user selects "About" from the Info button's popup.

NEW: Customizing Filters with Labelled Input Lines (2/5/96)

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.

`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.

NEW: Creating a Simple NewtApp (2/7/96)

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".

Create the NewtApp base view:

1) Create a layout file.
2) Draw a `newtApplication`.
3) Remove the following slots:
   `afterScript, allDataDefs, allViewDefs, superSymbol`.
4) Set the following slots to the following values:

```
allLayouts:     {
   default: GetLayout("default.t"), // see step 9 in the next section.
   overview: GetLayout("overview.t")} // see step 4, overview section.
allSoups:       {
   mySoup: {
      _proto: newtSoup,
      soupName: "SoupName:Signature",
      soupIndices: [],
      soupQuery: {}   }  }
appAll:         "All items"
appObject:      ["item", "items"]
appSymbol:      kAppSymbol
title:          kAppName
```

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

```
menuLeftButtons:  [newtInfoButton]
menuRightButtons: [newtActionButton, newtFilingButton]
```

8) Save the layout file as `"main"` 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 (necessary only until platform file is updated).
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.

NEW: 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 calls `FillNewSoup` if the soup on the internal store is empty.

The `'userName` slot 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;
```

Stationery

Pickers, Popups and Overviews

CHANGED: ProtoDigit Requires a DigitBase View (2/6/96)

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 `protoNumberPicker` for input like this.

If you really need to use `protoDigit` then 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.

NEW: Single Selection in ProtoListPicker-based Views (12/5/95)

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 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.

NEW: Using Icons withProtoLabelPicker (1/3/96)

Q: How do I successfully specify an initial icon for my `protoLabelPicker` and change the value of the icon programatically?

A: There are two relevant methods of `protoLabelPicker` that did not appear in early documentation:

`IconSetup()`

Returns an icon to use initially (like `TextSetup`). The default script will use the icon associated with the first item in the labelCommands array.

`UpdateIcon(newIcon)`
Set the icon to the `newIcon`.

NEW: Preselecting Items in ProtoListPicker (1/16/96)

Q: If I put name references in the `selected` array of a `protoListPicker`, it throws a `-48402` error. How do I preselect items?

A: You are probably setting up the `selected` array in your `viewSetupFormScript` or `viewSetupChildrenScript`. Use the `viewSetupDoneScript` to setup the `selected` array, then send the `Update` message to `protoListPicker` to tell it to update the display.

NEW: Picker List is Too Short (2/5/96)

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. Note that this may be fixed in future revisions of the Newton OS.

NEW: 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
```

NEW: 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;
```

NEW: How To Use ProtoOverview (2/6/96)

Q: I can't figure out how to use `protoOverview`, even after reading the NPG 2.0 First Edition (beta) docs. How does it work?

A: The most recent documentation does not contain the current information on `protoOverview`. Below is some interim documentation on how to use it. This information is also in a DTS sample called "protoOverview".

`protoOverview` was really set up as the basis for `protoSoupOverview`. Because of that, you need to do some extra work to use just the `protoOverview`.

The easiest way to use the overview is to encapsulate your data in a "cursor"-like object that supports the methods: Entry, Next, Clone. Since your data is probably in an array, you can use a "cursor" object like this:

```
{  items: nil,

   index: 0,

   Entry: func()
   begin
      if index < Length(items) then
      items[index];
   end,

   Next: func()
      if index < Length(items)-1 then
      begin
         index := index + 1;
         items[index];
      end,

   Move: func(delta)
   begin
      index := Min(Max(index + delta, 0), kNumItems-1) ;
      items[index];
   end,

   Clone: func()
      Clone(self),

   GetIndexEntry: func(theIndex)
      items[theIndex]}
```

You need to define the following methods in your `protoOverview`:

`Abstract(item, bbox)`
`item` - data item returned by your
`bbox` - bounding box of the shape you should draw
Return a shape that represents the item. Must fit in the bounding box specified by bbox.

`HitItem(hitIndex, xcoord, ycoord)`
`hitIndex` - index of item relative to top of displayed items
`xcoord` - x coordinate of the tap relative to item that was tapped
`ycoord` - y coordinate of the tap relative to item that was tapped
Called when an item is tapped. If checkboxes are enabled, you should check if the x is less that the `selectIndent`. If so, call the inherited `HitItem`, otherwise your item has been tapped on.

Note: `hitIndex` is relative to the displayed items, not the total items. You will need to track what the real "top" index is.

An example is:

```
func(hitIndex, xcoord, ycoord)
begin
   if xcoord < selectIndent then
      inherited:HitItem(hitIndex, xcoord, ycoord) ;
   else begin
      hitIndex := hitIndex + saveIndex;
      print("hit item: " & hitIndex) ;
      :Dirty(); // refresh the view
   end ;
end
```

`IsSelected(entry)`
`entry` - the "entry" that is tapped on
Return true if the entry is selected (the checkbox is checked in the overview). Note that selection is different from highlighted or hit.

`Scroller(dir)`
`dir` - direction to scroll
Implements the code necessary to scroll the contents that will be displayed.

Typically, this will update some sort of saved index and any highlight tracking an then redo the children of the view.

`SelectItem(hitIndex)`
`hitIndex` - index of item relative to top of displayed items
Perform whatever record keeping is required to toggle the selected state of the item at `hitIndex`. `SelectItem` is called each time the checkbox for an item is tapped.

Note: `hitIndex` is relative to the displayed items not the total items. You will need to track what the real "top" index is.

`viewSetupChildrenScript()`
You must provide this method. You must send the `SetupAbstracts` message from this script. Note that `SetupAbstracts` is expecting a cursor object. If you use the cursor object given above, this method will work correctly.

You also need to define the following slot in your `protoOverview`:

`cursor`
The cursor object based on the above object. This should contain the encapsulated array you are displaying.

In addition to the above methods and slot, you must provide a mechanism to find an actual data item given an index of a displayed item. In general, you need some sort of saved index that corresponds to the first displayed item.

You also should provide a mechanism to track the currently highlighted item. This is distinct from a selected item.

Controls and Other Protos

Text and Ink Input and Display

NEW: ProtoPhoneExpando Bug (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: 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.

NEW: 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 2.0 System NPG in the "Built-In Applications and System Data" chapter, in the section on "Notes" 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.

NEW: 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.

Stroke Bundles

Recognition

Opening the Corrector Window (12/8/95)

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 valid keyView exists.

```
local correctView := GetRoot().correct;
if correctView and GetKeyView() then
   correctView:Open();
```

CHANGED: Custom Recognizers (2/8/96)

Q: I would like to build recognizers for gestures and objects other than those built into the Newton system.

A: Currently thereÕs 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 for more information on the RegGlobalKeyboard function.

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:PIEDTS", []);
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.
```

CHANGED: 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 Kit cannot handle entries larger than 32K.

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: Like QuickDraw in the MacOS operating system, the drawing package in the Newton OS supports no calls for rotating text. 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 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% 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 3D and 50% 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: Like the MacOS QuickDraw architecture, 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.

Difference Between LockScreen and RefreshViews (6/17/94)

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 all at once.

The Newton Bitmap Format (11/27/95)

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

A: The Newton OS provides routines for creating and manipulating bitmaps at runtime. Here is a description of the format of a bitmap object.

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

<raw bitmap data> - class 'bits
 Binary object
 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 should should normally be in agreement as to

the size of the bitmap, or other functions (for instance, hit testing) won't work properly. It can occasionally be useful to change the bounds slot (only) to get `CopyBits` to do scaling.

Sound

System Services, Find, Filing

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 available in the Newton 1.x OS or Newton 2.0 OS. You can either accept the extra idle script calls, or use some other non-`clParagraphView` based view to implement your idle functions.

NEW: Functions to Create Application-specific Folders (12/5/95)

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 some 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, nil for system-wide
    result - symbol, the folder symbol of the newly added folder.
```

`AddFolder` takes a folder name and creates a new folder for the application or system, broadcasting as necessary. It should commonly only be used to add application-specific folders. It is a violation of the UI guidelines to create system-

wide folders without user direction.

The function 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.

`RemoveFolder(folderSym, appSymbol)`
`folderSym` - symbol, the folder symbol of the folder to remove
`appSymbol` - symbol, the application for which to remove the folder
`result` - symbol / NIL, the removed folder or NIL if no folder was removed.

`RemoveFolder` can be used to remove a folder from the available list for an application. It also should commonly be used only for application-specific folders.

NEW: 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 DTS Q&A "Creating Custom Finders" for more information.

NEW: 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 on page 12-15 of 2.0 NPG System guide (version a0.5)

`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.

Intelligent Assistant

Built-In Apps and System Data

NEW: 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 just 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.

NEW: 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.

NEW: 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.

Localization

Utility Functions

NEW: 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.

Errors

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.

Routing

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.

Note that the PICT2 (color) picture format is not currently supported by the Newton drawing system.

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 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.

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 the `fields` variable.) 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`.

NEW: 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;

entry := query(getunionsoup("Names"), {type:'index}):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);
```


Transports

NEW: 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.

Endpoints & Comm Tools

Maximum Speeds with the Serial Port (3/8/94)

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,
                                'char,
                                'char,
                                'boolean,
                                'boolean,
                                'boolean,
                                'boolean,
                          ],
                    },
},

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

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

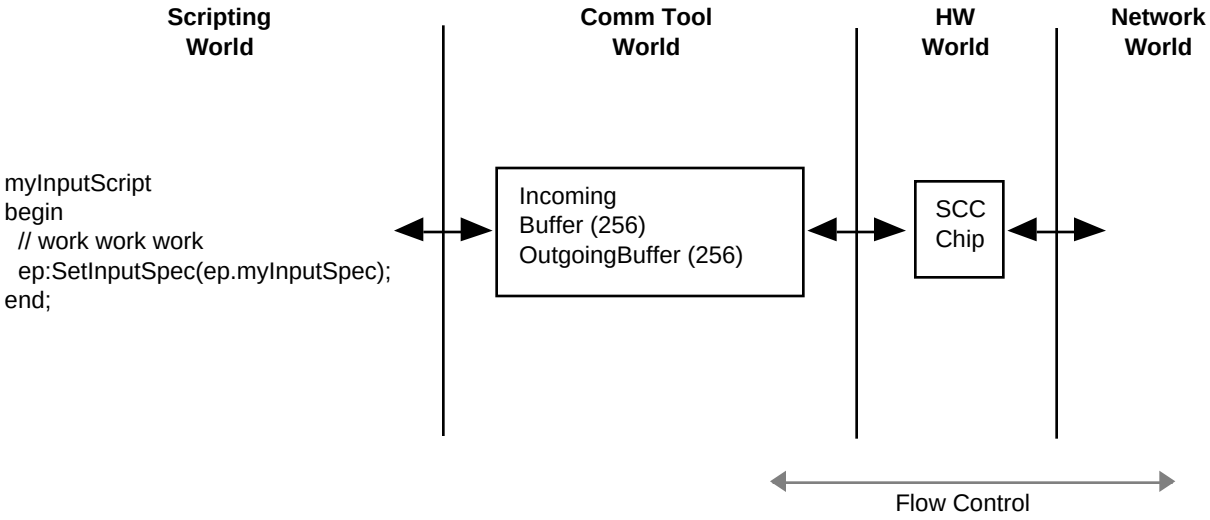
Q: What is error code -18003?

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.

Here's 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.

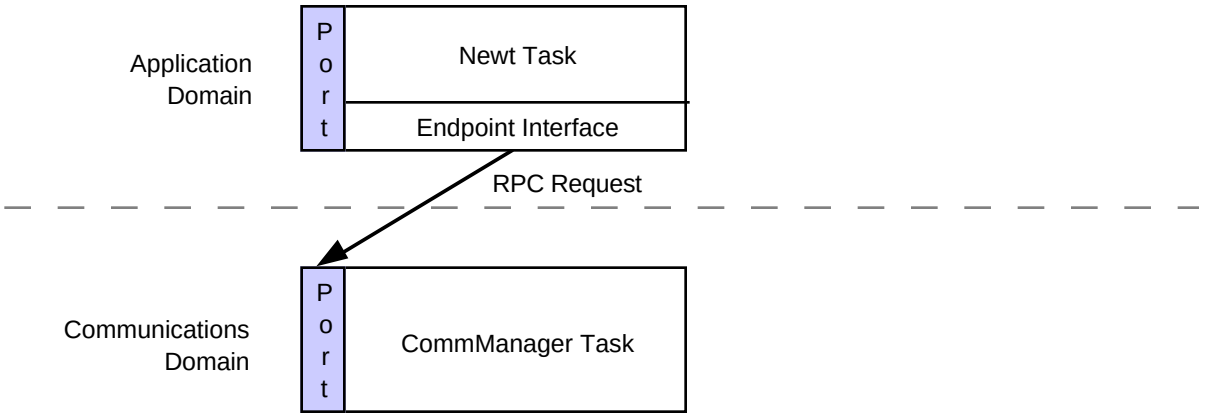
What Really Happens During Instantiate & Connect (6/14/94)

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

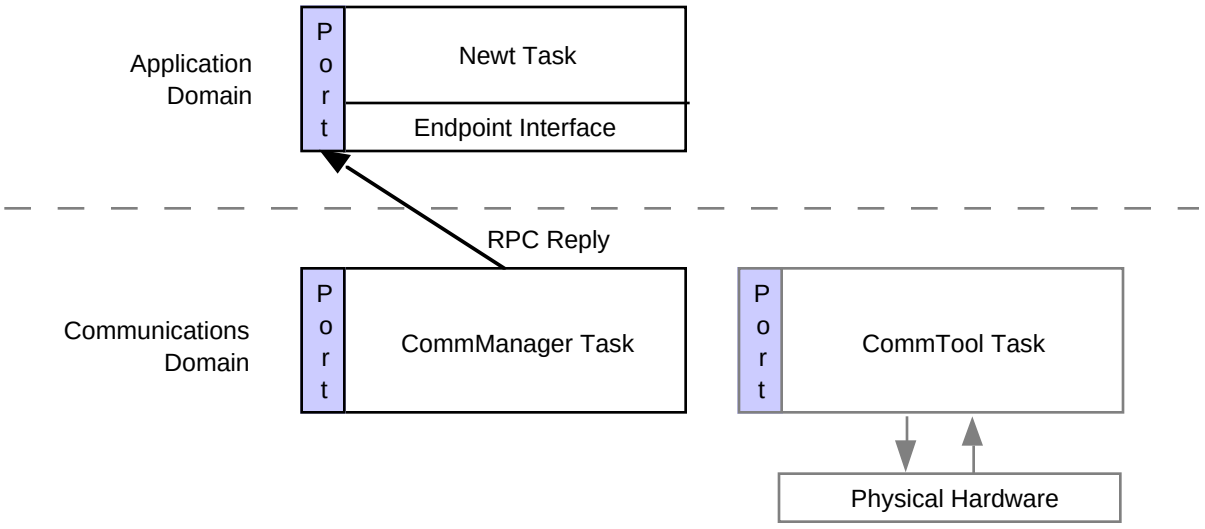
A: Exactly what happens depends on the type of endpoint being used. In general:

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

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

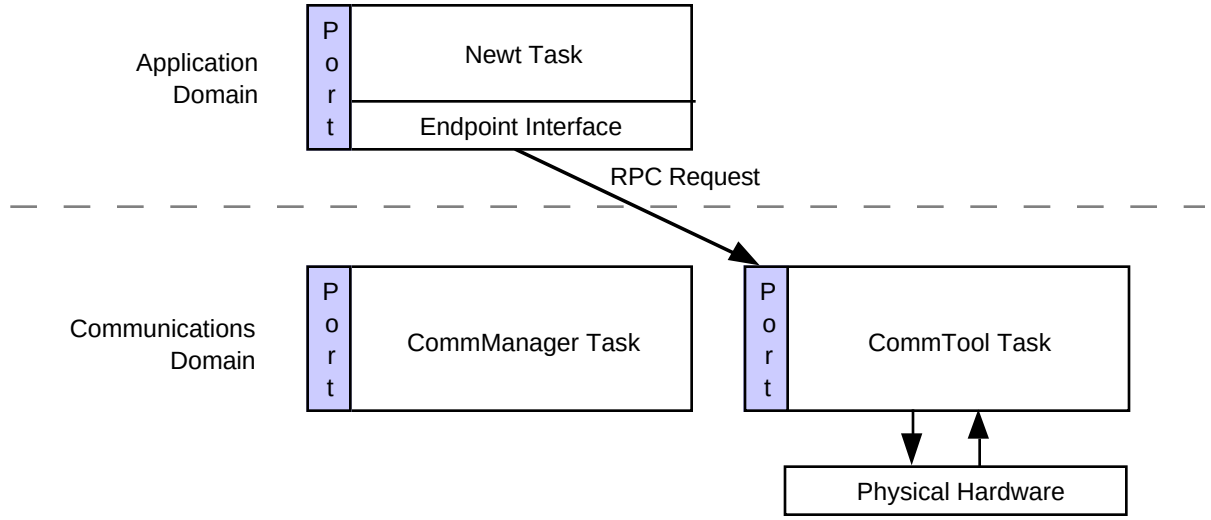


The CommManager task creates the appropriate CommTool task(s) and replies to the communications service request. Each CommTool task initializes itself . In response to the `Bind` request the CommTool acquires access to any physical hardware it controls, such as powering up the device. The endpoint is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

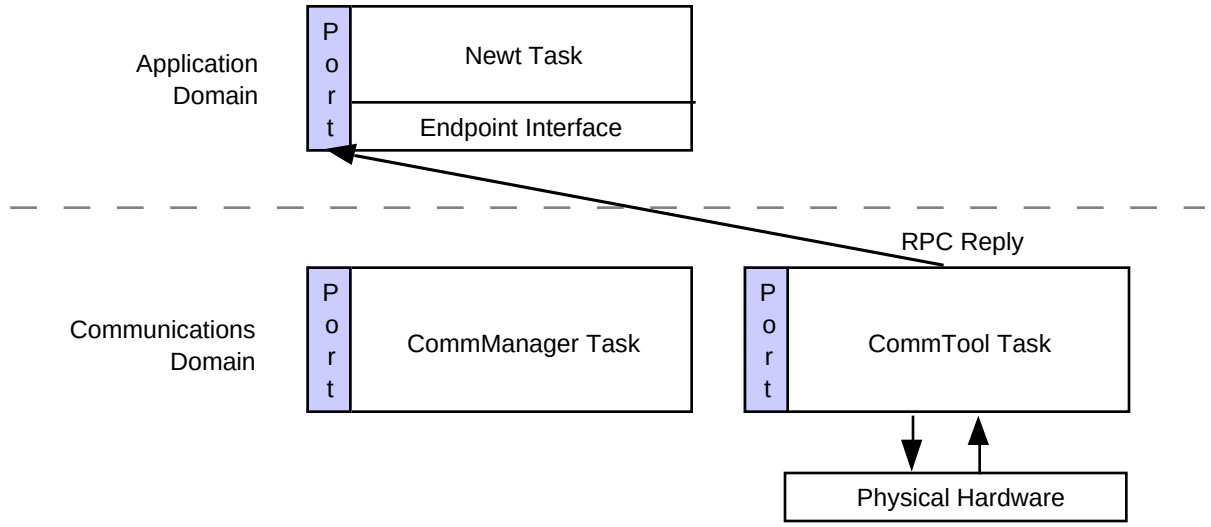


An endpoint may use multiple CommTool tasks, but there will be a single NewtonScript endpoint reference for them.

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.



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.



`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.

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 MessagePadÊ100Êupgrade 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).

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 GetNamedResource (see global data) 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 AppleLink, ftp.apple.com (Internet) and on the Newton Developer CDs.

The following sites have more information about other infrared protocols:

nada.kth.se:home/d89-bga/hp/remote/remotes (Internet, ftp)
flash.ecel.uwa.edu.au (Internet, ftp)

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 function if necessary to determine the exact execution time of apartialScript.

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
};
```

Sharp IR Protocol (12/2/94)

(Distilled from source dated 10/14/1992)

1 Serial Chip Settings

Baudrate	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) require 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 device 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 is to be transmitted, it is 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 bit 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.

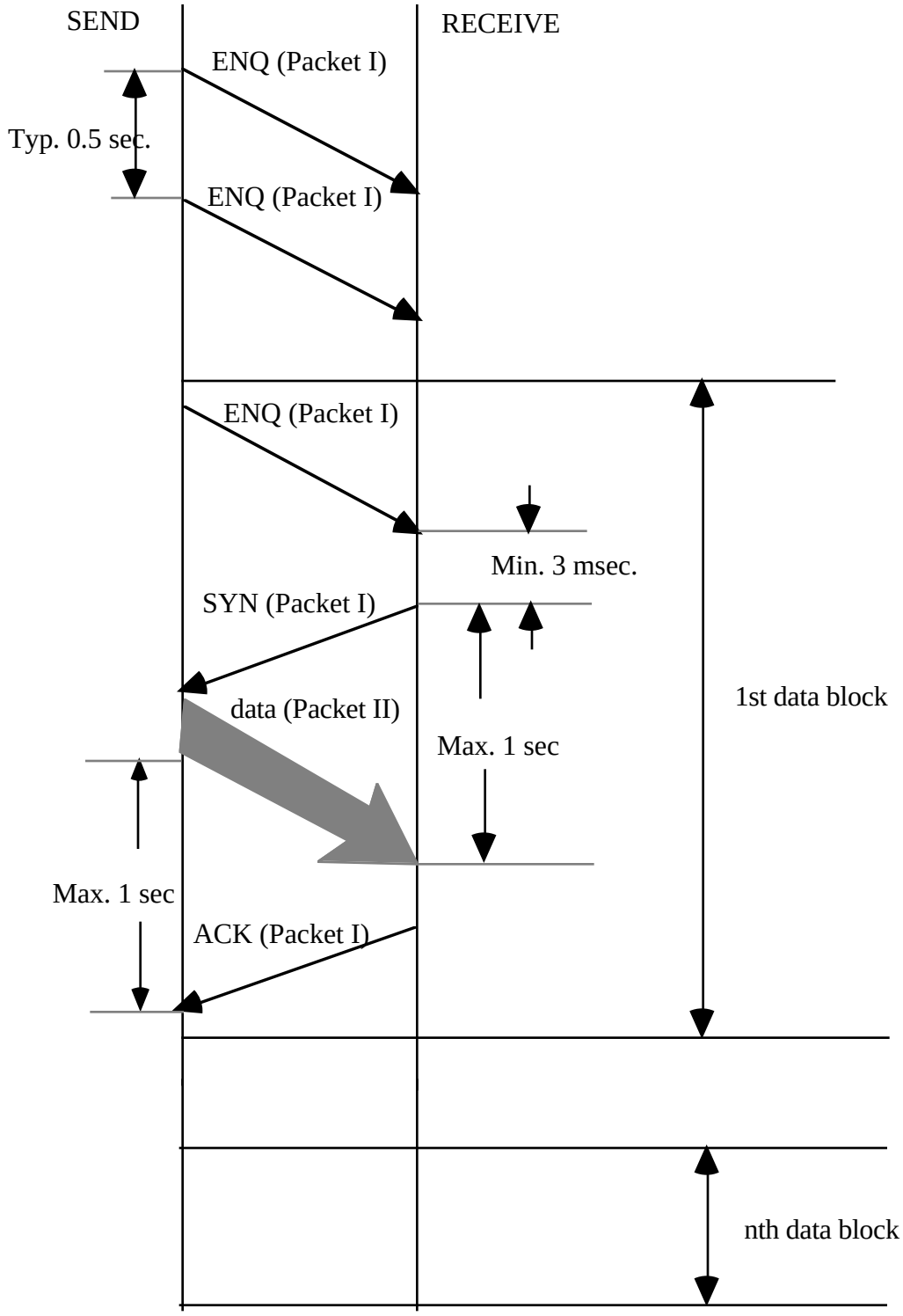
4.2.2 When (B) receives an ENQ packet, (B) transmits either a SYN packet to continue the session or a CAN packet to terminate the session.

4.2.3 When (B) receives a valid type II packet (eg. 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.



NEW: 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);
```

NEW: 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.

```
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.

Modem Setup

Desktop Connectivity (DILs)

NEW: 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.

NEW: 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.

NEW: 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.

User Interface

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")

IR Hardware Info (9/6/94)

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

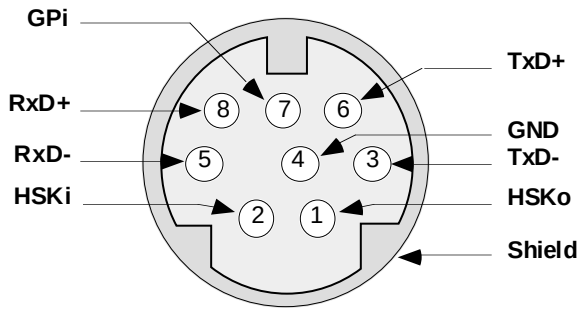
A: This information is hardware dependent, and is only valid for the Original Message Pad, Message Pad 100, and Message Pad 110 products.

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.

Serial Port Hardware Specs (6/15/94)

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

A: In the Apple MessagePad 100, 110, and 120, 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):



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)

All inputs are:	Vih = 0.2V	Vil = -0.2V	Ri = 12k ohms
All outputs are:	Voh = 3.6V	Vol = -3.6V	Rl = 450 ohms

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.

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.

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
```

NewtonScript

NewtonScript Object Sizes (6/30/94)

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 memory, in pseudo-ROM memory, inside the package or in ROM. In MessagePad platforms, these objects are aligned to 8-byte boundaries. 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

Immediates (integers, characters, TRUE and NIL) 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.

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
            /* actually creates an undefined slot */
            foo :=  myEncloser:GetImportantSlot(); // MAY WORK

            importantSlot := 12;                   // WON'T WORK
            myEncloser.importantSlot := 12;        // MAY WORK

         end
      }
   };

   myEncloser.nestedSlot:GetTheValue();
```

The workaround 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:= {}; tSlotName := "Slot_1";
```

At this point is there a way to then create this ?:

```
MyFrame.Slot_1
```

A: There is a function called Intern, that takes a string and makes a symbol. There is also a mechanism called path expressions (see the NewtonScript manual), 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

tSlotName := "Slot_1";
MyFrame.(Intern(tSlotName)) := 7;

// myFrame is now {x: 6, Slot_1: 7}
```

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.

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.

Debugging NewtonScript

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 ally in determining the heap used by your applications.

NEW: 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.

Newton ToolKit

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

Q: How can I use a PICT in ROM from the Picture slot handler in NTK?

A: You have to 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). Something like this in the `AfterScript`:

```
thisView.icon := ROM_outboxbitmap;
```

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

Q: When I have the Inspector window open and I debug the 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 in the Newton, which uses a buffer taken from a space 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 many functions, like `StrCompare`, are present in NTK at compile time, they should not be considered documented or supported unless explicitly defined in the Newton ToolKit User's Guide or other material from Apple Computer.

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

// function to compare strings (only page 0 characters)
// with the same order as Newton 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);
```

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 labels 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 miltifinder 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 Macintosh dialog 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 68K 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.

NEW: 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.

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

Q: When I build my project which 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 makeÑeach 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" (e.g., 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".

NEW: 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 a simple

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>
}
```

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

NEW: Current Versions of MessagePad Devices (2/7/96)

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 February 2nd, 1996 the latest versions are:

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)

English Newton 2.0 OS

MessagePad 120	2.0 (515299)

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)

