Newton Q&A: Ask the llama

This column first appeared in volume 24 of DEVELOP (the Apple technical journal for developers).

Q *The online discussion groups for Newton developers have a lot of references to compatibility these days. My application works fine on the 120, 110, and 100 models. Does that mean I'm compatible?*

A Good question. Compatibility doesn't mean your application works now, but that it's written in such a way that it will work on future Newton devices and operating systems. There are several APIs and methods for doing things on the 120, 110, and 100 that will work with them but are not necessarily compatible with future releases of the OS.

There are two main points to observe for the sake of compatibility:

- If it's not documented, don't use it.
- Catch exceptions; they *can* occur (especially if you ignore the first point).

Since compatibility is such an important question, it will be the focus of this column. The rest of the column will cover the most common breaches of compatibility. Where applicable, there will be an example of the incompatible and compatible ways of doing things. After reading this and making copious notes (especially where you find yourself saying "Oh dear" and "Oh no!"), you'll be in a position to make your code compatible. We also recommend that you try out your application with the Compatibility App Package (which is on this issue's CD and is available from various online services).

Note that we refer often to the Newton Toolkit platform file functions. The Toolkit comes with documentation and platform files can come with release notes. Both the documentation and release notes describe functions that are provided in lieu of future APIs. You should use these platform file functions where applicable. Call the code directly and don't modify it. That is, use the **call/with** syntax; don't place the code in a slot in your application and use message sending.

UNDOCUMENTED GLOBAL FUNCTIONS

There are four common offenders here: CreateAppSoup, SetupCardSoups, MakeSymbol, and GetAllFolders.

The function kRegisterCardSoupFunc in the platform file replaces both CreateAppSoup and SetupCardSoups. It's much simpler to use than the undocumented functions:

```
// RIGHT way
constant kSoupName := "MySoup:MYSIG";
constant kSoupIndices := '[];
constant kAppObject := '["Item", "Items"];
call kRegisterCardSoupFunc with
   (kSoupName, kSoupIndices, kAppSymbol, kAppObject);

// ******* WRONG way ****** Bad, naughty, nasty, skanky, way *****
CreateAppSoup(kSoupName, kSoupIndices, EnsureInternal([appSymbol]),
   EnsureInternal(kAppObject));
AddArraySlot(cardSoups, kSoupName);
AddArraySlot(cardSoups, kSoupIndices);
SetupCardSoups();
```

The fix for MakeSymbol is to call the Intern function; it does the same thing and it is documented. There's no replacement function for GetAllFolders; just don't call it.

UNDOCUMENTED GLOBAL VARIABLES

The three most common misused global variables are **cardSoups**, **extras**, and **userConfiguration**.

There are two uses of **cardSoups**: one is to register a card soup; the other to unregister it. Registering is taken care of with kRegisterCardSoupFunc (see above). Unregistering is done with another platform file function, kUnRegisterCardSoupFunc:

```
// RIGHT way
call kUnRegisterCardSoupFunc with (kSoupName);

// ******* WRONG way ****** Bad, naughty, nasty, skanky, way *****
SetRemove(cardSoups, kSoupName);
SetRemove(cardSoups, kSoupIndices);
```

You should never access the **extras** global variable. Not only is it undocumented, but so is the format of the. Both are subject to major revisions. The platform file function kSetExtrasInfoFunc is provided for setting information about items in the extras drawer. The most common use of this function is to give your application a different icon (see the ExtraChange DTS sample code on the CD).

There are also platform file functions to manipulate **userConfiguration**: kGetUserConfigFunc gets a slot from the **userConfiguration** soup entry; kSetUserConfigFunc lets you set user configuration information; and kFlushUserConfigFunc should be called when you've changed user configuration information.

```
// RIGHT way
local userName := call kGetUserConfigFunc with ('name);
if userName then
begin
   if StrEqual(userName, "Doctor") then
      call kSetUserConfigFunc with ('name, "The Doctor");
   call kFlushUserConfigFunc with ();
end;

// ******* WRONG way ****** Bad, naughty, nasty, skanky, way *****
if userConfiguration.name AND
      StrEqual(userConfiguration.name, "Doctor") then
   userConfiguration.name := "The Doctor";
```

UNDOCUMENTED SLOTS AND METHODS

This is a broad category of violations. The most common problem is **keyboardChicken** in the root view. But there are others, like **cursor.current**, **paperRoll.dataSoup**, **dockerChooser** in the root view, **UnionSoup:Add**, and anything in a built-in application. Unfortunately, there is no right way to access most of these. The exceptions are **cursor.current** and **Add**:

```
// RIGHT way
local currentEntry := cursor:Entry();
myUnionSoup:AddToDefaultStore(anEntry);

// ******* WRONG way ****** Bad, naughty, nasty, skanky, way *****
```

```
local currentEntry := cursor.current;
myUnionSoup:Add(anEntry);
```

Also, don't rely on the routing slips, such as **mailSlip** and **printSlip,** being in the root view. You can, however, still use those symbols in your routing frame.

UNDOCUMENTED MAGIC POINTERS

If you use one of these, you know it. Just think what would happen if the magic pointer changed from a view to a string: you would get some pretty bad behavior. Note that most of this could be dealt with by catching exceptions.

STORE AND SOUP ASSUMPTIONS

All you can assume is that store 0 is the internal store. You can't rely on there being only one other store, nor can you rely on the position of a store in the array returned by GetStores. Also, don't assume that another store is a card or even that there is just one store per card.

If you support moving or copying items between stores, you shouldn't find the title of the store. Use the constant ROM_cardAction as provided in the platform file:

```
// RIGHT way
routingFrame := {
   print: ...
   ...
   card: ROM_cardAction
}
```

In addition, don't assume that your soup will exist on every store. Currently, if you register your union soup, it's automatically created on every store that enters the Newton; however, this may change in the future:

```
// RIGHT way
GetUnionSoup(kSoupName):AddToDefaultStore(anEntry);

// ******* WRONG way ****** Bad, naughty, nasty, skanky, way *****
aStore:GetSoup(kSoupName):Add(anEntry);
```

Remember that AddToDefaultStore or Add could throw exceptions. Wrap your calls to these functions in exception handlers.

Finally, if you support the soup change mechanism, don’t assume that the change is adding or deleting an entry. It could be something else, such as a soup being created or removed from a store.

SCREEN SIZE

Don’t assume the screen is any particular size. It could be larger or smaller than current devices. It could also be wider than it is tall. Your application size setup routine (usually in the viewSetupFormScript) should take this into account. Have maximum and minimum sizes. Close your application if it can’t handle the current screen size:

```
// Code to close your application
constant kUnsupportedScreenSize :=
   "WiggyWorld does not support this screen size";

DefConst('closeMeFunc, func(x) x:Close()) ;

:Notify(kNotifyQAlert, EnsureInternal(kAppName),
   EnsureInternal(kUnsupportedScreenSize));
AddDeferredAction(closeMeFunc, [self]);
```

UNDOCUMENTED FEATURES OF DATA TYPES

Only rely on the features and details of built-in data types that are documented. There are three common problem areas: order of slots in a frame, precision of integers, and implementation of strings.

The order of slots in a frame is undefined. It just so happens that in the current implementation the first 20 slots are returned in the order added. This is not a documented feature, so don’t rely on it.

Integers are documented as having at least 30 bits of precision. This doesn’t mean they’ll always be 30 bits; they could be wider (as anyone who has used compiled NewtonScript can tell you). Note that compiled NewtonScript integers may not be 32 bits; they also follow the “at least 30 bits” rule.

The biggest offender is assumptions about how strings are implemented. Don’t rely on strings being null terminated or being composed of two-byte Unicode characters. The practical upshot is that you should use StrLen to find the length, and StrMunger (or &) for length changes. Don’t use Length, SetLength, or

BinaryMunger with strings. Do not set a string using the array accessor. You can check a character, but do not set a character.

MISCELLANEOUS BITS

Don't send messages directly to the IOBox; use the kSendFunc platform file function. Nor should you read the items in the IOBox soups.

Also note that there are platform file functions to register and unregister for Find that you should use.

Always use SetValue when you're changing the view or other system values.

Use only the **body** slot in items that you route. Don't rely on slots other than **body** surviving the routing process. On a related note, do not rely on **category** slot of **fields** in your SetupRoutingSlip method either.

Don't rely on the closing order of views in the viewQuitScript. If you need to do some ordered cleanup, you can initiate your own message (for example, myViewQuitScript) from the view that first receives the viewQuitScript.

Replace system functions and messages at your peril. It's conceivable for them to support other data types in the future (for example, to take NIL now where before they only took a string).

Don't assume anything about the built-in applications. Don't assume that they exist, or that their soups are there, or that the view structure will stay the same. If you do need to use a system feature (for example, a particular prototype, global function, or root method), test your assumptions:

```
local cardFileExists := GetRoot().cardfile;

if cardFileExists then
begin
   local cardFileSoup := GetUnionSoup(ROM_cardfilesoupname);
   if cardFileSoup then
      ...
end;
// :-0
if GetRoot().keyboardChicken then
...
end;
```

Current Newtons have two levels of Undo; this may change. There could be more or fewer levels and it could change to Undo/Redo. It's safest to call AddUndoAction from inside your undo action; this will support Undo/Redo if we implement it, but will do nothing if we do not.

The llama is
the unofficial mascot of the Developer Technical Support group in Apple's Newton Systems Group. Send your Newton-related questions to NewtonMail or eWorld DRLLAMA, or AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt.

Thanks
to our Newton Partners for the questions used in this column, and to jXopher Bell, Henry Cate, Bob Ebert, David Fedor, Stephen Harris, Jim Schram, Maurice Sharp, James Speir, and Bruce Thompson for the answers.

Have more questions?
Need more answers? Take a look at Newton Developer Info on AppleLink.