Soup's On

by Mike Engber
Apple Computer, PIE Technical Support
Copyright © 1993 - Michael S. Engber

This article was published in the November 1993 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

With the myriad of different soup and store functions available on the Newton, it's hard to tell which functions are important and how they should be used. This article discusses how an application should properly create and use soups. This article is not a tutorial on the topic of soups. This article assumes that the reader is familiar with the chapter on soups in the Newton Programmer's Guide and has some experience using NTK to write Newton applications.

Union Soups

The first rule of using soups on the Newton is "Use union-soups." Union-soups handle the details of whether you should be writing data to the internal store or an external store, a PCMCIA card. The Newton user should be in control of where his/her data goes. When the user inserts a card or taps the Card item in the extras drawer, the Card Storage dialog comes up. This has a checkbox titled "Store new items on card" with which the user controls where data goes. By using union-soups your program automatically respects the user's wishes.

You can think of union-soups as grouping together the soups with a given name on the various available stores. Currently, that is just one or two soups, but future Newtons may change that. Later in this article I discuss how to properly create union-soups.

There are three basic calls that should handle 99% of your soup related code. First, there's GetUnionSoup. It takes a soup name (a string) as its parameter and returns a union-soup. Next there's the union-soup method, AddToDefaultStore, which does exactly what its name says. It stores a new entry to the store the user has chosen as the default. And finally, there's EntryChange, which you use to make sure any changes you make to a soup entry are written out to the storage device.

Stores and Soups

Stores can be likened to volumes and soups can be likened to databases. A store can contain many soups just as a volume can contain many databases. A soup is comprised of entries just like a database is composed of records. Soups can maintain indexes on their entries. These indexes can be specified when the soup is created or added or deleted later.

There are a variety of soup and store methods. They are documented and in practice you rarely need to use them, so I won't say much about them here. The GetStores() global function returns an array of the available stores. You can assume that the first element of the array is Newton's internal store. Currently, the second element of the array, if it exists, is always the PCMCIA card, but this assumption won't necessarily hold for future Newtons. Consider the problem of a Newton with multiple slots.

When debugging from NTK's Inspector it's sometimes useful to use the GetSoupNames method. For instance, `GetStores()[0]:GetSoupNames()` returns an array of the soup names from the internal store.

Entries

Soups are comprised of entries. As you access the soup, frames are created for the entries. Normally, you can just think of soups as consisting of these frames. In reality, these frames are just caching information from the soup into RAM. That's why it's necessary to call EntryChange to write the information from an entry's frame in memory back out to the soup. Otherwise, the change would be lost when the Newton is restarted.

Deciding when to call EntryChange is sometimes difficult. For instance, if you have an input line, you wouldn't want to call EntryChange from its viewChangedScript. If the user was entering data into the line with the keyboard, your viewChangedScript gets called after every key press. Calling EntryChange with every key press would be noticeably slow.

In some situations it's obvious when to call EntryChange. For example, a natural time to call EntryChange is when the user dismisses a dialog box. Other cases may not be so clear cut. The Notepad uses a viewIdleScript to call EntryChange about every five seconds. That's why the Newton user manual suggests waiting 5-10 seconds before removing a PCMCIA card or you may lose data. If you manage to corrupt the Notepad or its soup while hacking around, you may get error dialogs every five seconds, each time the Notepad tries to write to its soup.

There isn’t much structure imposed on soup entries. For instance, there’s no requirement they all have the same slots. All entries have a _uniqueId slot which contains an integer uniquely identifying the entry within its

soup. The system creates this slot for you, so you don't need to worry about it. In fact, the _uniqueId slot is only unique within a soup, not within a union-soup. Since applications normally use union-soups, the _uniqueId slot isn't of much practical use.

You're free to put any slots you want in your entries and to vary them from entry to entry. The only slots you have to worry about are the slots that are indexed. When you create a soup, you decide what slots are used to build indexes. If you specify that an index should be built on slot foo and that foo contains a string, then it's important that you make sure that the foo slot in every entry contains a string. It's permissible for an entry not to have a foo slot at all, in which case it won't participate in queries on the foo index. The essential point is that if an entry has an indexed slot, the slot contains data of the right type.

To create an entry, you should create a new frame and pass it as the parameter to an AddToDefaultStore message. The frame is destructively modified. You'll see that it now has a _uniqueId slot. When a frame is added to a soup, a deep copy of it is made. That is, all its slots are followed and everything those slots reference is followed and copied, and so on, until everything is copied into the soup. This is necessary because soups persist across restarts, so they can't contain references to objects in RAM. Because of this you need to think carefully about the data structures you use for soup entries. For instance, a view whose _parent chain ultimately leads to the root view would be a very bad candidate for a soup entry.

There are two exceptions to this copying rule. Pointers into ROM are not followed. Since the ROM is persistent across restarts, pointers into ROM are safe to save in soups. The second exception is _proto slots. A frame's _proto slot is not followed when creating a soup entry, nor is a _proto slot created for the entry. The _proto slot is not removed when a frame is added to a soup, so it may appear that you have a soup entry with a _proto slot. However, the frame is only a temporary cache of the entry's data. Next time you retrieve the entry from the soup, you see it does not have a _proto slot. If you need it, you will to create it every time you retrieve the entry.

Cursors

Cursors are used to navigate through soups. You obtain a cursor as the result of a call to Query. This simplest possible query is:

```
curs := query(uSoup,{type: 'index})
```

I use this technique in the NTK Inspector to get a "quick and dirty" cursor for peeking at entries in a soup. I say "quick and dirty" because this code is sloppy. The argument specifies a query type of index, but it doesn't specify an indexPath. This works because all soups have an index on the _uniqueId slot which is used as the default indexPath.

There are a wide variety of options for specifying queries, as well as a wide variety of cursor functions. They are all documented in the *Newton Programmer's Guide*, so I won't go into them here. Here are a few points about using cursors.

- Cursor:Entry() can return nil if the cursor runs off either end of the soup. It can return the symbol 'deleted if the entry it references was deleted. Be sure your code can handle these cases.
- Traversing your soup creates a frame for each soup entry as it is encountered. (Yes, these frames are cached, so you don't pay a penalty for visiting the same entry twice.) Avoid traversing every entry in the soup. Use indexes and startKeys to limit your search. Using validTests doesn't help with this problem. The entry frame has to be created in order to pass it to the validTest.
- Don't leave unnecessary references to entries lying around. For instance, building up an array of entries ties up a lot of memory. As soon as you're done with the array, delete any references to it so it can be garbage collected, which in turn allows all the entry frames it contains to be garbage collected.

Creating Your Application's Soup

The benefits of using a union-soup depend on one crucial point we've overlooked so far. The individual soups which comprise a union-soup must exist on all available stores so the user can direct his/her data there. There is a global array, CardSoups, which is used by the system to handle this. The elements of CardSoups are used in pairs. The first element of the pair is the soup name, a string. The second element is an array of index specifications (the frames you use to specify indexes when creating a soup). Whenever a card is inserted, the system uses this array to make sure each soup in Card-Soups exists on the card, creating soups as necessary.

You need to use the function CreateAppSoup when creating soups. Call CreateAppSoup to make sure your soup

exists on the internal store. CreateAppSoup creates it only if necessary. CreateAppSoup takes four arguments: the soup name, an array of soup indices, a one element array containing your application's appSymbol, and your application's appObject. (appObject is an array of two strings, such as ["entry", "entries"], that describes the singular and plural of the data in that soup. Filing expects the appObject slot of your base

view to contain an array like this.) The first two argument are used to create the soup; see CreateSoup's documentation for all the details on index specs. The last two arguments are added to the soup's info frame as an annotation describing who uses the soup and what's in it. Due to a bug in the SetInfo soup method, you need to use EnsureInternal to make sure these second two arguments reside in internal memory.

There are conventions for naming your soup. If your application creates only one soup, you should use your package name for the soup name. Your package name is formed by concatenating your Extras drawer name with your unique signature, for instance "foo:myCompany". If your application creates multiple soups, use your package name as a suffix to a descriptive soup name ("soup1:foo:myCompany").

Basic Methods

Putting all this information together into a simple recipe, I recommend you define two methods in your application's base view, RegisterCardSoup and UnRegisterCardSoup. Here is their code.

```
//constants defined in "Project Data"
constant kAppSymbol := '|llama:PIEDTS|;
constant kPackageName := "llama:PIEDTS";
constant kAppObject := '["llama","llamen"];

constant kSoupName := kPackageName;
constant kSoupIndexes := '[{structure: slot,                    path: name.first, type: string}];

//2 base view methods

RegisterCardSoup: func(soupName,soupIndexes,appSymbol,appObject)
 //returns a union-soup for your app to use
 begin
  //first check for system provided function
  if functions.RegisterCardSoup then
   return RegisterCardSoup(soupName,soupIndexes,
                           appSymbol,appObject);
  CreateAppSoup (soupName, soupIndexes,
                 EnsureInternal([appSymbol]),
    EnsureInternal(appObject));

  //ensure your soup will exist on stores
    //which later become available
  AddArraySlot(CardSoups,soupName);
  AddArraySlot(CardSoups,soupIndexes);

  //ensure your soup exists on all
    //currently available stores
  local store;
  foreach store in GetStores() do
   if NOT store:IsReadOnly() AND NOT
            store:HasSoup(soupName) then
    store:CreateSoup(soupName,soupIndexes);
  GetUnionSoup(soupName);
 end,

UnRegisterCardSoup: func(soupName)
 begin
  //first check for system provided function
  if functions.UnRegisterCardSoup then
   return UnRegisterCardSoup(soupName);
```

```
local pos := ArrayPos(CardSoups,soupName,0,
      func(x,y) ClassOf(y)='String AND
      StrEqual(x,y));
if pos then ArrayRemoveCount(CardSoups,pos,2);
```

```
end;

//in your viewSetUpFormScript, call
:RegisterCardSoup(kSoupName,kSoupIndexes,
        kAppSymbol,kAppObject);

//in your viewQuitScript, call
:UnRegisterCardSoup(kSoupName);
```

For now, you should define RegisterCardSoup and UnRegisterCardSoup as methods in your base view. In future Newtons, these functions may be provided in the ROM. The methods defined above allow your application to take advantage of this future functionality by first checking to see if these global functions are available.

Your application should call RegisterCardSoup when it starts up. RegisterCardSoup works by first making sure your soup exists in the internal store. Next, it registers your soup with the CardSoups global variable. Then it ensures your soup exists on any cards already inserted. Finally, it calls GetUnionSoup and returns that value. Your application can use the value returned by RegisterCardSoup as an alternative to calling GetUnionSoup directly.

Your application should call UnRegisterCardSoup when it quits. UnRegisterCardSoup simply unregisters your application from the CardSoups global variable. Your application only needs to worry about its soup existing on all available stores while it's actually open. If a new card is introduced while your application is closed, it is handled automatically when your application gets opened–RegisterCardSoup is run and creates the soup. When your application quits, it should also take care to remove any references it has to entries, soups, or cursors. This usually means setting to nil any slots in which you stored soup or cursor references.

Sharing Soups With Others

Newton programmers are encouraged to reuse existing soups for their own needs. The format of the standard soups is documented in the Newton Programmer's Guide. This section discusses how to properly handle the sharing of data.

Registering With SoupNotify

Applications can be notified when a particular soup gets changed. For instance, maybe your application has developed a following of third party utilities which manipulate your soups, or more likely, you're using a built in soup, like "Names." If you implement the Find feature, the FindAll overview notifies your application of changes it makes to your soup using this same mechanism. It's generally a good idea for your application to set itself up for notification.

To register for notification all you have to do is add two entries to the SoupNotify global array. Your application should register in its viewSetupDoneScript and unregister in its viewQuitScript. The elements of SoupNotify are used in pairs. The first element in the pair is the name of the soup. The second element is the appSymbol of the application to be notified. Below is some code to register an application for changes to the System Soup. This code assumes that a constant, kAppSymbol, has been defined for the application's appSymbol:

```
//register w/ SoupNotify in viewSetupDoneScript
AddArraySlot(SoupNotify,ROM_SystemSoupName);
AddArraySlot(SoupNotify,kAppSymbol);

//unregister with SoupNotify in viewQuitScript

local soupNotifyPos := ArrayPos(soupNotify,
        kAppSymbol,0,nil);
ArrayRemoveCount(soupNotify,soupNotifyPos-1,2);
```

Notice that the code for unregistering looks for the appSymbol instead of the soup name. This is essential since other applications may have requested notification for the same soup. Also note that this code can easily be generalized to unregister you from multiple soups:

```
//unreg all SoupNotify entries in viewQuitScript
```

```
local soupNotifyPos;
```

```
while soupNotifyPos := ArrayPos(soupNotify,
        kAppSymbol,0,nil) do
 ArrayRemoveCount(soupNotify,soupNotifyPos-1,2);
```

Receiving Notifications

Once your application is registered it receives soupChanged messages when soups change. You need to define a soupChanged method in your base view which accepts one argument, the name of the soup that changed. Your application should respond to this in whatever manner is appropriate. Normal applications have no need to respond unless they're open. That's why I recommend you register when your application opens and unregister when your application closes.

Sending Notifications

If your application makes changes to a shared soup, it should use the BroadcastSoupChange function to notify other applications of the change. BroadcastSoupChange takes a single argument, the name of the soup that changed. Unfortunately, the current version of the Prefs application doesn't use BroadcastSoupChange. So if you try to register with the System Soup, don't be surprised if Prefs doesn't notify you.

Making Changes to Other Application's Soups

In addition to using BroadcastSoupChange, there is another convention you should following when modifying soups not owned by your application. If your application needs to add its own slots to entries in another application's soup, you should create only a single slot. Use your appSymbol as the slot name. This avoids name conflicts, keeps the entries from getting cluttered up, and allows for easy removal of your data.

Cleaning Up Spilled Soup

You may have been wondering "Doesn't this mean that while my application is open, it creates soups on every card inserted into the Newton?". The answer is "yes." However, the soups will be empty unless the card is the default store and the user creates data with your application.

You may also have been wondering "If the user inserts lots of cards in the process of using my application, can't his/her data be spread out all over the place?". Again, the answer is "yes." There's not much you can do about this. Hopefully Newton users will learn to organize their data on cards, just like computer users learn organize their data on floppy disks.

The issue I haven't discussed is deleting your soups. Similar issues are deleting your preferences from the System Soup and deleting modifications you've made to other applications' soups. Analogous problems exist in the Macintosh world. How do you clean up your Preferences folder or get those out of date icons out of the Desktop Database?

I don't have a good answer. You certainly don't want to do it in your viewQuitScript. You probably don't want to do it in your RemoveScript. Remember, your RemoveScript is run when the card containing your application is removed, as well as when the user chooses Remove Software from the Prefs or Card applications. There is no straightforward way to determine which event is happening.

I suspect the eventual outcome will be one, or more, of the following:

- A reliable method to distinguish having your card removed and Remove Software will be documented. Then, when your application detects the Remove Software case, it can prompt the user to see if they really want all traces of the application removed. You wouldn't want to do this without prompting the user. Suppose the user is simply deleting an extra copy of your software from a card.
- Users will explicitly do the cleanup. Some applications will provide this option to users–as a button, part of their Prefs panel, or as a utility application. Alternately, if applications follow the guidelines for naming their soups and preferences data, it should be possible to write a cleanup application.
- Chaos will reign. Soups will build up until the ozone layer is depleted. Life, as we know it, will cease.

Preferences–Exception to Always Using Union Soups

You may recall that I said applications should always use union-soups. There's one exception–the SystemSoup, where preferences are stored. It doesn't make much sense to have your preferences spread out on PCMCIA cards. If you know your application is always run from a particular card, it might make sense, but in general, using the internal store seems to be the right thing.

Since everyone has to share the SystemSoup and the Newton's internal store, be considerate and don't keep large amounts of data there. Below is some code that retrieves this preferences information and creates it if necessary.

```
local sysSoup := GetStores()[0]:GetSoup
```

```
        (ROM_SystemSoupName);
local cursor := Query(sysSoup,{type: ‘index,
        indexPath: ‘tag, startKey: PackageName});
local prefsEntry := cursor:Entry();
if NOT (prefsEntry AND
    StrEqual(prefsEntry.tag,kPackageName)) then
  prefsEntry := sysSoup:Add({tag: PackageName,
            appSpecificSlot: nil});
```

Conclusion

Your application should allow the user flexibility in storing data. The user might keep your application internally and store its data on PCMCIA cards (even if your application is too big to fit in the internal store of the current Newton, don’t assume this will be the case for future Newtons.) Alternately, your application might be kept on a PCMCIA card along with its data. On a two slot Newton, your application might be stored on one PCMCIA card and its data on another. You should design your application with enough flexibility to handle any configuration. Union-soups provide an easy way to do this. π